PROGRAM PROTECTION MANUAL FOR THE C - 64 VOLUME II

LEARN THE SECRETS OF SOFTWARE PROTECTION.

COPYRIGHT 1985

BY CSM SOFTWARE, INC.

ALL RIGHTS RESERVED

INTRODUCTION

This manual is designed for the computer user or programmer who has some background in programming, machine language and program protection. We are not going to assume a high level of expertise. We only expect that the reader has read and become familiar with the information presented in the PROGRAM PROTECTION MANUAL FOR THE C-64 (VOLUME I).

The best way to get the most out of this book is to keep the PROGRAM PROTECTION MANUAL FOR THE C-64 (VOLUME I) handy for reference. One cannot be expected to remember all the techniques described in the first manual, so feel free to refer back to it for information when needed.

The information presented herein will be for illustrative purposes only. The routines featured in this manual are original and contain code similar to that in actual use. Don't be surprised if you see some programmers using our routines in the near future, they've done it before.

The first few chapters are a review of some very important aspects of computer software. If the information contained in these chapters seems familiar, that's because it is mainly from the first manual on program protection. Please take the time to re-read this information. It is very important!

The rest of the manual contains all new information, presented in a logical manner. Read this book from front to back, first chapter to last. The information presented in the earlier chapters is used as building blocks for the later chapters. Take your time when reading the chapters, try to understand each and every concept before going on. It has taken months to compile the information contained in this manual so don't feel bad if you don't understand all of it the first time through.

We have called upon many different experts to help us write this manual. We would like to give special credit to these fine folks for all their help. Without their help this manual could not have been written.

SPECIAL THANKS TO THE FOLLOWING PEOPLE FOR CONTRIBUTING TO THIS MANUAL - YOU FOLKS DID A GREAT JOB!!

BILL MELLON

DAVE JOHNSON

CAYE GIRGENTI

P. J. MYERS

PHIL SLAYMAKER

MIKE POWERS

T. N. SIMSTAD

P.S. Thanks to my wife and kids for putting up with me while writing this.

COPYRIGHT NOTICE

PROGRAM PROTECTION MANUAL FOR THE C-64 VOLUME II
COPYRIGHT 1985 (C) BY CSM SOFTWARE INC
ALL RIGHTS RESERVED

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 CSM SOFTWARE INC.

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 CSM SOFTWARE INC.

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 CSM SOFTWARE INC.

WARRANTY AND LIABILITY

Neither CSM SOFTWARE INC., nor any dealer or distributor makes any warranty, express or implied, with respect to this manual, the disk 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 will CSM SOFTWARE INC. 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. CSM SOFTWARE INC. will not assume any other warranty or liability. Nor do they authorize any other person to assume any other warranty or liability for them, in connection with the sale of their products.

UPDATES AND REVISIONS

CSM SOFTWARE INC. reserves the right to correct and/or improve this manual and the related disk at any time without notice and without responsibility to provide these changes to prior purchasers of the program.

IMPORTANT NOTICE

THIS PRODUCT IS SOLD SOLELY FOR THE ENTERTAINMENT AND EDUCATION OF THE PURCHASER. IT IS ILLEGAL TO SELL OR DISTRIBUTE COPIES OF COPYRIGHTED PROGRAMS. THIS PRODUCT DOES NOT CONDONE SOFTWARE PIRACY NOR DOES IT CONDONE ANY OTHER ILLEGAL ACT.

TABLE OF CONTENTS

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 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 less than a minute. With todays high speed copy machines, programs may be duplicated within a minutes or less. This will include the time necessary to verify the disk.

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 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 for a software pirate to copy?

Two methods currently exist to protect the program from unauthorized copying. Both 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 or 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 lawful owner of the program to make a copy for archival purposes. The law also provides for the lawful owner of a program to adapt (modify) the program if the adaptation (modification) is essential to the use of the program.

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. Anytime 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 anyone 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.

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, gotten 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, tape or computer chip 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 arvchival 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. Other companies offer the legitimate purchaser the right to obtain an archival copy for a nominal fee, thereby keeping the honest people honest.

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 buyer 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 many copy programs that will copy almost any disk, errors and all. They take 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 programs 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 information, 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 program keep you from having the copy you need.

In 1976 Congress passed the current copyright law. This law is refered to as 'TITLE 17, USC, COPYRIGHTS'. A copy of this law is not hard to find; all law libraries have one. Check with your local court house or university libraries for a copy of the act. Prior to 1976 the copyright laws could be found in the 'COPYRIGHT ACT OF 1909'. The 1909 law did not specifically address computer software (for a good reason - there weren't any computers in 1909). It was not until 1976 that computer programs were specifically mentioned in the copyright law and then only briefly. On December 12, 1980 the Congress revised Section 117 of the 1976 copyright act to specifically include computer software. The revision was known as 'COMPUTER SOFTWARE ACT OF 1980'.

The 1980 act provided a definition of a 'computer program' (17 USC 101). Computer software is defined as: 'A COMPUTER PROGRAM IS A SET OF STATEMENTS OR INSTRUCTIONS TO BE USED DIRECTLY OR INDIRECTLY IN A COMPUTER TO BRING ABOUT A CERTAIN RESULT.'

This definition is hardly startling or revolutionary; it is, however, the first definition of computer software in a U.S. copyright law.

The 1980 act is a revision of section 117 of the 1976 act. The new section 117 provides some specifics relating to backing up programs. It gives the lawful owner of a program the right to copy or adapt a copyrighted program as long as the copying is essential to using the program (117 USC 117):

'NOTWITHSTANDING THE PROVISIONS OF SECTION 106, IT IS NOT AN INFRINGEMENT FOR THE OWNER OF A COPY OF A COMPUTER PROGRAM TO MAKE, OR AUTHORIZE THE MAKING OF, ANOTHER COPY OR ADAPTATION OF THAT COMPUTER PROGRAM PROVIDED (1) THAT THE NEW COPY OR ADAPTATION IS CREATED AS AN ESSENTIAL STEP IN THE UTILIZATION OF THE COMPUTER OR (2) THAT THE NEW COPY OR ADAPTATION IS FOR ARCHIVAL PURPOSES ONLY AND THAT ALL ARCHIVAL COPIES ARE DESTROYED IN THE EVENT THAT CONTINUED POSSESSION OF THE COMPUTER PROGRAM CEASES TO BE RIGHTFUL.'

'ANY EXACT COPIES PREPARED IN ACCORDANCE WITH THE PROVISIONS OF THIS SECTION MAY BE LEASED, SOLD, OR OTHERWISE TRANSFERRED, ALONG WITH THE COPY FROM WHICH THE COPIES WERE PREPARED, ONLY AS PART OF THE LEASE, SALE OR OTHER TRANSFER OF ALL RIGHTS IN THE PROGRAM. ADAPTIONS SO PREPARED MAY BE TRANSFERRED ONLY WITH THE AUTHORIZATION OF THE COPYRIGHT HOLDER.'

The COPYRIGHT ACT OF 1980 allows the backing up of a copyrighted program for archival purposes provided that the archival copies ARE NOT retained by original purchaser after the purchaser sells the program. This means that you are allowed to keep backup copies of the program as long as you keep the original. In most cases there is only one occasion when you can no longer keep a back up copy. That's when you sell it to someone else.

The purchaser of a copyrighted program is allowed to make an adaptation (change or modify) as long as the adaptation is an essential step in the utilization of the program. The purchaser may not sell or transfer the adapted program without the authorization of the copyright holder. All adaptations of the original program must be destroyed upon sale of the original!

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 (header) including SYNC MARKS, ID numbers, CHECKSUM and TRACK and SECTOR numbers. 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 properly read the information contained in on the disk. 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 they cares more about protecting their program from pirates than they do 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 the 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 this, 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!

Some of the newer protection schemes rely on different forms of data on the disk. Rather than have the disk drive's normal DOS (Disk Operating System) read the data from the disk, many programmers are now writing their own routines to read this data from the disk. What may appear as an error to the normal DOS may be the programmers own form program protection. Programmers have just recently begun to read and write data to the disk at will. This is accomplished by writing their own ML routine that resides in the RAM memory of the disk drive. All the programmer has to do is execute this routine to read or write data from the disk. Since the disk drive is under the control of this new routine all of the error checking routines may be bypassed. This allows the programmer to read non-standard dats as it comes from the disk. The specific routines and alograthims will be explained in further detail in later chapters of this book.

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.

EVOLUTION OF PROTECTION SCHEMES

In this chapter we will try to cover the evolution of disk protection schemes for the C-64. We will give an overview of the schemes that have appeared on the majority of software during the past few years. The schemes discussed will be those that work and those that don't. Those that work will be covered with special emphasis. You may want to use this guide to help you understand the various types of protection.

IN THE BEGINNING....

In the beginning there were not any copy programs for the C-64. The owner of the C-64 had much diffilculty making an archival copy of his treasured software. It was a dark and desolate time for the owner of the C-64 computer. There was not much software available for this new and powerful machine. The software that was available was overpriced, poor quality and mostly written in BASIC.

When the proud owner of a new piece of software would get it home, he immediately tried to make an archival copy of the valuable new program. This required the user to LOAD and SAVE each and every file from the disk. If each file was a BASIC program file, it would not pose much of problem. All the programmer had to do was just use another type of file (seq, usr, rel or ML) on the disk and the user could no longer back up the original disk by just simply LOADing and SAVEing the files.

The early types of program protection that developed were simple, easy to implement and mostly ineffective. Most of the original software for the C-64 was written in BASIC. The original protection schemes were designed to prevent the user from simply loading and then saving the program directly from memory. This could be done in a number of ways. First (and most ineffective), is through the use of a few simple POKEs when the program was RUN. These POKEs would disable the RUN/STOP & RESTORE keys, thereby preventing the user from stopping the program. If the user could not stop the program, they could not save the program back to another disk. This type of protection was only effective if the user ran the program. All the user had to do was to LOAD the program into memory, and immediately SAVE the program back to another disk. The SAVE had to be performed prior to RUNning the program. It didn't take long before the end user was able to copy all his software by simply LOADing and SAVEing the files.

The first improvement in program protection came through the use of a boot or loader program. This boot was also written in BASIC. In order for the main program to operate properly it was necessary to load the main program from another program (the boot or loader program). The user would load and RUN the loader

program. The loader program would POKE a few values into memory then it would LOAD and automatically RUN the main program. The main program would then check (PEEK) to see if the first program had placed the proper values into memory. This way the user could not simply LOAD and SAVE just the main program, they also had to SAVE the boot. Occasionally the loader program would have hidden lines, some pointers reset or bogus line numbers (see the PPM volume I). All this was an attempt to confuse the inexperienced user and was still not very effective.

Then came the first real step forward in program protection, the use of a ML loader. Not just any old ML loader, but an auto-loader. An auto-loader is a ML program that resides in a special area of memory (see the chapter on auto-loaders). When the auto-loader is loaded into memory it will automatically execute (run). There is no need to tell the computer to RUN (nor SYS). As soon as the program is loaded into memory the program executes. Generally these auto loaders perform the same functions as the BASIC loader performs. They will store a few values in memory and then LOAD and execute the main program. The main program will check for these special values as it executes. If the special values are there the program will run, if not the program will crash. It was still possible to LOAD the BASIC program and SAVE it from memory. One problem though, if the user was not sophisticated enough to go into the BASIC program and modify it so that it would not require the special values in memory, it still would not run. When the user tried to LOAD the ML auto-loader it would immediately execute, preventing the user from copying the ML auto-loader. As long as the user did not know much about BASIC programming this was an effective technique. Even if the user could LOAD and LIST the BASIC program, they still had to have enough knowledge to modify it so it would properly execute without the loader.

Shortly thereafter, the BASIC programs began to be replaced by ML. Now the user could not LIST either the loader or the main program. Since the main program was written in ML, the user could no longer just LOAD and SAVE it. This posed quite a problem for the user; it became almost impossible to back up the disks that they purchased. Well, about this time, a copy program appeared on the market. Not just any old copy program, but 1541 BACKUP!! This would copy a whole disk in 30 minutes. I'm sure some of you 'old timers' remember this one, it was the copy program with the 'gas gauge'. The program was slow and limited in what it would do, but it worked! It was possible copy the whole disk, from track 1 to track 35, without even knowing what was on the disk. This program would copy all types of files, BASIC or ML, Program, Sequential, User or Relative. The only thing that the copy program would not do is make a copy of a disk with an error on it. The copy program would just 'give up' if it encountered an error.

Well, it didn't take long until someone came up with idea that they could stop this type of copy program. All they had to do was to create an error on the disk. They just intentionally placed an error on the disk so that when the copy program encountered the error it caused the program to give up. The idea here is to inhibit or prevent the user from making a copy; the error served no purpose other than keeping the user from copying the disk. Two general types of errors appeared on the scene. The first and most common type of error involved the use of modified data on the disk. All the programmer had to do is put some data on the disk that should not be there and the disk drive would interpret this as an error. The disk drive has a very sophisticated error checking routine that insures proper reading of the data. If as little as one byte is 'out of place' the drive will give an error condition. The most common type of errors to appear were 20, 21, 22, 23, & 27. These errors, while undesirable, were 'acceptable' (you'll see what I mean by 'acceptable'). When the disk drive read this type of error it would cause the READ/WRITE head to beat against its end stop. This beating, if it occurred repeatedly, would cause the drive to go out of alignment. Keep in mind that at this stage of program protection the program never read this error. The error's only purpose was to inhibit the user from copying the disk. This makes the error acceptable.

The second and the least common type of error is where the programmer would punch a hole in the disk. The programmer would save the program on the disk in such a manner that a certain track or group of tracks (usually the outer tracks) would be unused. The programmer would then use a paper punch to punch a hole in disk on these unused tracks. The original disk would never move the R/W head over this 'error' on the disk. The programmer would then place all kinds of warning labels on the program informing the user that if they tried to copy the disk their disk drive would be damaged. If the user tried to make a copy of this disk they would end up ruining their disk drive!!!! When the disk drive encountered the hole in the disk, the R/W head would usually become damaged beyond repair. This seemed like a pretty good way for the programmer to protect his program, right?? WRONG! The programmer overlooked one small fact. When the disk drive reads information from a disk the R/W head is left in last position that it read data from the disk. This means that if the user were to read another program that used the outer tracks, the disk drives R/W head would remain on the outer tracks. Now, just suppose what would happen to the disk drive if the disk with the hole were inserted and the user tried to LOAD the program. Instant disaster, because the R/W head had been left on the outer tracks from the last disk; now when the 'punched' disk is inserted, the R/W head will be directly over the hole. The legitimate user would have destroyed his disk drive, all because some programmer didn't take the time to properly protect his program.

About this time copy programs took a great step forward. The file copy program appeared. The file copy program would allow the user to copy any type of file from one disk to another. It was now possible to just copy all the files from the original disk and make a perfect copy. Well it didn't take long before the programmers came up with a method of preventing file copying a disk and getting a working copy.

The next 'little' evolution in program protection is the most disasterous change that could have occurred. Some 'bright' programmer came up with the idea that if they could put an error on the disk to inhibit anyone from copying the disk it might be possible have the program itself check for the error on the disk. This is where program protection began to stink. Remember that whenever the disk drive encounters an error the R/W head gets beat up against its end stop. Now every time that one of these 'protected programs' loads into memory the error gets checked. Every time the user loads one of his favorite programs into memory the protection scheme causes the R/W head to beat up against the end stop. After the R/W head bangs enough times, the drive gets pounded out of alignment.

We are going to introduce a very important concept here; take time to understand this. Let's look at the concept of program protection.

1). The programmer places non-standard data on the disk (data that would not normally be there, i.e. the error).

2). The program checks for the presence of this non-standard data (the error).

3). If the non-standard data is present (as on the original disk), the disk drive will pass a specific value(s) to the computer. If the non-standard data is not present (as on the copy disk) the disk drive will pass a different value to the computer.

4). If the proper value is returned from the disk drive the program will execute properly. If the incorrect value is returned from the disk drive, the program will crash.

This 'error checking' form of program protection is especially hazardous to the casual user of protected programs. Some programs go so far as to tell the user that 'NOISE AT THE END OF LOAD IS NORMAL'. The poor user, who does not know any better, will end up beating their disk drive to death while listening to the 'NORMAL' noise. Thousands of disk drives have been beat out of alignment by this 'NORMAL' noise.

This error checking caught on like wild fire. Before long every programmer was using this terrible form of program protection. The programmers somehow never gave any thought to what might happen to the users disk drive after repeated use of the protected program. Even worse than that is some progrmmers knew

the effects of these errors on the disk drive and used them anyway.

Generating these errors required a programmer to have a very thorough understanding of how data is stored on the disk and how this data could be manipulated. It was no simple task to write a program that would allow the user to duplicate these errors. In fact, for a short time this error checking was uncopyable. The programmers were smug in the belief that they had a form of program protection that was beneficial to the software industry. This error checking kept the user from making a working copy of the origial disk and thereby prevented software piracy.

Well, the user soon realized that his disk drive was getting beat to death by using these protected disks. In order for the end user to use the software without destroying their disk drive, it became essential to modify the program. If the user was to keep their disk drive in alignment, it was necessary to remove the need for the program to find the error on the disk. This is where the casual user found himself reading books on ML and trying to find out ways to modify the original program. When the user found out how simple was is to modify these protected programs, it wasn't long before everyone was doing it.

Then a company wrote a program (UNGUARD BY MICRO-W) that would allow the user to repoduce these errors on the copy disk (why anyone would want to reproduce these errors is beyond me). Now the user had a tool to make an archival copy of their valued program. UNGUARD would do errors 20, 21, 22, 23 & 27. This program was a real break through for the user who only wants to reproduce these errors.

It didn't take the programmers long to come up with a new type of error that could be used for program protection. This 'new' error was #29, I.D. mismatch. The way that programmers generated this error is by reformatting a single track with a different I.D. Error 29 is unique in that it does not cause the head to beat against the end stop. Error 29 was a pleasant change on the program protection scene. This type of protection offered the programmer some measure of security for their software without beating the user's disk drive to death.

It wasn't very long before the users found a method of producing the error 29. A simple BASIC program could be written that would allow the user to reformat any track with any I.D. desired.

We are now at the time where the endless cycle in program protection really becomes apparant. The programmers have found a way to protect their software. Shortly thereafter, a company writes a program that will copy those disks. Then the programmers change their protection scheme, making it uncopyable. In a few months some other company writes a copy

program that will allow the user to back up this new protection scheme. Then the programmers come up with a new protection scheme that can not be copied..... and so on and so on.

Now the level of expertise in program protection is at a new plateau. Programmers have become more sophisticated and so have the program protection schemes. Some companies have included an extra sector on tracks 18-24. This may seem like a new protection scheme, but it is really a very old technique. The extra sector is a hold over from the days of the 2040 & 3040 disk drives from Commodore. Commodore disk drives used to have one more sector than the 1541 does on tracks 18-24. All a programmer had to do is use the format from a 2040 or 3040 disk drive and copy his program on to it. Then the programmer checks for the presence of the extra sector, if it's there the program will execute properly. If the extra sector is not there, the program will crash.

Well, in just the past few months we have seen some very drastic changes and advancements in the field of program protection. Programmers have learned how to read and write data anywhere on the disk. The data can be written on the track or in between tracks (half-tracks). The data can be placed beyond track 35 (extra tracks). The data may be written at different speeds to the disk (modified density). The data can be written in a number of different ways to the disk. It is not important how the programmer choses to write the data on the disk. What is important is that the programmer must be able to verify that the non-standard data is present. Recall our discussion of the concept of program protection. Keep in mind that no matter what form of program protection is used on the disk, the same basic premise of checking for the non-standard data is followed.

We have made an attempt to bring the reader up to date in the field of program protection. From here it will be necessary to look at the specific way that data is stored on the disk, how this data can be modified and how a program can be written that will read this non-standard data. Up to now the information presented has been very straight forward and easy to follow. From here on out, in may take a little more concentration to fully understand the material. If you don't grasp everything we are telling you, don't worry about it. Take your time and re-read the following chapters if necessary.

THE FUTURE: A PERSONAL OPINION

We've stepped through the evolution of copy protection techniques and now it's time to look at the future. What is the direction for copy protection, and what does that mean to the user? Will the new protection schemes prohibit you from making an archival copy of your software?

Copy protection is becoming so sophisticated that many of the current copy programs are unable to handle the present schemes let alone those of the future. We have investigated the newest copy programs on the market and have found each one lacking in some way. Some are better than others, but none have been able to successfully overcome all of the protection schemes currently being utilized. This is not meant to be a condemnation of copy programs, but merely a statement of fact.

We are now approaching a level of protection that simply cannot be overcome. This is mainly due to the built in hardware limitations of the 1541 Disk Drive.

You may have noticed that the newest copy programs will only copy specific programs. A few months later, an update is offered that will copy a few more programs. This is the future folks! Most of the copy programs today include a routine to read the headers of the original disk. Once this information is read, the program goes into a routine to duplicate THAT specific disk. If it is not a disk that has been analyzed 'in house' and provided for through the copy program, you will probably end up with an unsuccessful copy attempt. With the introduction of non-standard sectors, altered density bits, extra sectors, and the like, it is becoming increasingly more difficult for a copy program to allow for and deal with all of these possibilities. We can copy a disk that utilizes all of these techniques, but where these errors are to be placed on the disk and which techniques will be used is the problem we face now. This does not include what we will encounter in the future. Copying some of these schemes will require extensive investigation of each track and sector of the original disk and even then we have the problem of duplication. Some of the schemes being developed today may, because of hardware limitations, prove impossible for the 1541 disk drive to duplicate.

Where does this leave the legitimate user? Although the programmers intent is to keep his work safe from the 'pirate', it is the legitimate user that comes up on the 'short end' in this never-ending saga. It would seem that every program you buy, requires that you also purchase an updated copy program before you can exercise your right to make an archival copy of your software. And what about those who own other types of disk drives? Are they to be kept from using the latest software because the protection scheme being utilized by the programmer can only be read from a 1541 disk drive? Copying some of these

schemes requires extensive investigation of each track and sector of the original disk and even then we have the problem of duplication.

Making a archival copy of your original program, still does not allow you to exercise all of your rights. You are allowed by law to make revisions to the programs that you own (provided that these revisions are essential the lawful use of the program). This may be a crucial factor in business software, or utility programs. The programs you buy may not satisfy all of your needs. You may purchase a program and find yourself wishing that it had one or two more features. If you could access the code, you could add those features! A perfect copy of the original disk will not allow you access to the code if the program is protected. Let's not forget that some of these programs still 'beat' your disk drive to death. A copy disk will not eliminate this problem for you either.

The material presented in this book, along with it's predecessor (PROGRAM PROTECTION MANUAL FOR THE C-64), is designed to offer you an alternative. You can take control of your software, or you can remain a passive victim. The road to control may seem rocky at first, but it is worth the time and effort. With the techniques and tools provided through these manuals, you will learn to create your archival copies and have the access necessary to alter the program code to suit your needs. With time, patience, effort, and careful study, you will no longer have to purchase those expensive 'updates' to exercise your rights.

Try out the techniques presented in this manual. If one doesn't work, try another. You'll find that what you learn from one program can be applied to another. Why not change a branch statement, or hunt for an entry point instead of taking out your check book to purchase the latest '99.999% EFFECTIVE COPY PROGRAM' on the market?. Just a thought on those '99.999% EFFECTIVE COPY PROGRAMS' - Why is it that every time I want to copy something, it falls in that 0.001% group???? With our methods the most you can lose is a little time, but we believe you'll prefer that to losing money.

Don't be surprised if you learn something along the way. This was one of our primary goals in the preparation of this manual. LOAD and RUN is not enough for those who wish to know WHY. Why do some programs run automatically? Why does the disk drive rattle with some programs? How does SYS 64738 perform a RESET? How can I make my programs re-start by pressing the RESTORE key? You will find many of the 'hows' and 'whys' addressed in these pages.

A great deal of material is included on the inner working of the disk drive. You will be able to see what the track and sector editors do not show.

It's up to you now. Take charge and learn something in the process. By the way, don't forget to have FUN!

INTRODUCTION TO MACHINE LANGUAGE

In this chapter, we will take an introductory look at MACHINE LANGUAGE (ML). We will use a machine language monitor to enter our programs. The monitor we have chosen is LOMON, which is on the program disk that accompanies PROGRAM PROTECTION MANUAL VOLUME II. This monitor resides at HEX $8000, and may be activated from BASIC with SYS 32768.

We do not present this chapter as the ultimate 'MACHINE LANGUAGE TEXTBOOK'. Our main objective is to get you started in the right direction. Simple applications will be presented, along with examples to help clarify what you learn. For those wishing to continue their study of machine language, CSM Inc. is planning to publish a text in the near future. Watch the NEWSLETTER for further details.

Programming in machine language requires careful attention to detail. A difference of one byte could easily lock-up your computer. A condition of this kind will not do any damage to your computer, but you may find that the only way to recover control of your computer is to use your RESET button. If you do not have one, you will have to power-down and start over. By all means, if you don't have RESET button, get one (see PPM Vol. 1)

WHAT IS BINARY?

You could go through life without ever needing to understand the BINARY number system. You can even program in machine language without a knowledge of BINARY. So why even look at it? First, it's nothing to be afraid of. Second, it is the microprocessor's native number system. Although we will use the DECIMAL system to help explain BINARY, our emphasis will be on the relationship between BINARY and HEXADECIMAL (HEX). HEX is important because this is how we will code our programs. It is not essential to know BINARY, so if this section confuses you, just skip it. You can always come back to it later.

One unit of memory is called a BIT. BIT stands for BINARY DIGIT, meaning a unit that can be switched one of two possible ways. Thus a BIT can have only two different values, ON (1) or OFF (0). If we have a set of eight BITS, called a BYTE, the total number of different combinations of 0's and 1's possible is 256 (count 'em!). This gives us 256 one-byte codes we can use to represent our program instructions, data, etc.

In DECIMAL (BASE 10), the rightmost digit is the least significant digit. The digit in this position stands for multiples of 1, which is called the place value. As we move left, the place value increases by a factor of 10 each time (this is what makes it a BASE 10 number). The second position has a place value of 10x1=10, the third position 10x10=100, the fourth position 10x100=1000, etc. The total contribution made

by a particular digit in a number is calculated by multiplying the digit itself times its place value. Let's look at the base 10 number 4321 as an example.

```
  PLACE VALUE  1000   100    10     1
x DIGIT USED      4     3     2     1
-------------------------------------
= TOTAL VALUE  4000   300    20     1
```

Interpreted in DECIMAL this set of digits represents a value of 4000+300+20+1 = 4321. This should come as no surprise.

In BINARY (BASE 2) the rightmost digit position also has a place value of 1. As we move left, however, the place value increases by a factor of 2 rather than 10 (see below). Also, in binary the only digits that can be used are 0 and 1, so multiplying the digit times its place value is very simple. If the digit is 1, include the place value in the number's total value; if the digit is 0, ignore it. Let's use the binary number %10110110 as an example (the % is used to indicate binary).

```
  PLACE VALUE  128  64  32  16   8   4   2   1
x DIGIT USED     1   0   1   1   0   1   1   0
-----------------------------------------------
= TOTAL VALUE  128   0  32  16   0   4   2   0
```

This set of BINARY digits represents a DECIMAL value of 128+32+16+4+2 = 182.

Now you try a couple.

```
  PLACE VALUE  128  64  32  16   8   4   2   1
x DIGIT USED     0   1   0   1   1   0   1   1
-----------------------------------------------
= TOTAL VALUE    0  64   0  16   8   0   2   1
```

The value returned is 64+16+8+2+1 = ?

```
  PLACE VALUE  128  64  32  16   8   4   2   1
x DIGIT USED     1   0   0   0   1   1   0   1
-----------------------------------------------
= TOTAL VALUE
```

The value returned is ?

That's all there is to it. Now if the programmer had to program in BINARY, it would be a real chore. After a while all those 0's and 1's start to dance around before your eyes. They are difficult to remember, and hard to type in.

HEX TO THE RESCUE!

Instead of BINARY we can use HEXADECIMAL (HEX). HEXADECIMAL is BASE 16. We know that there are 10 different digits (0-9) in DECIMAL and we've learned that there are only 2 different digits (0-1) in BINARY. In HEX, therefore, we have to have 16 different digits. Wait a minute, you say. We can use the regular digits 0-9 for the first ten HEX digits, but what do we do for the other six? Answer: we use the letters A through F to stand for the 'digits' 10 through 15.

The following chart should make the relationship clearer:

DECIMAL	BINARY	HEX
0	% 0000	$ 0
1	% 0001	$ 1
2	% 0010	$ 2
3	% 0011	$ 3
4	% 0100	$ 4
5	% 0101	$ 5
6	% 0110	$ 6
7	% 0111	$ 7
8	% 1000	$ 8
9	% 1001	$ 9
10	% 1010	$ A
11	% 1011	$ B
12	% 1100	$ C
13	% 1101	$ D
14	% 1110	$ E
15	% 1111	$ F
16	%10000	$10

Once again, the rightmost digit position in HEX has a place value of 1. As we move to the left, this time the place value increases by a factor of 16 each time. Let's look at how we determine the (DECIMAL) value of the HEX number $10A5 (the $ indicates HEX).

	PLACE VALUE	4096	256	16	1
x	DIGIT USED	1	0	A	5
=	TOTAL VALUE	4096	0	160	5

The DECIMAL equivalent of $10A5 is therefore 4096+160+5 = 4261. Note that the HEX digit 'A' stands for 10 as shown in the chart above.

The reason we use HEX instead of BINARY is that it is easy to convert from one to the other, and HEX numbers are easier to remember. To convert from BINARY to HEX, you divide the BINARY number into groups of four BITS (starting from the right end of the number). Each group corresponds to exactly one HEX digit, in fact the corresponding digit from the chart above. For instance % 0110 1100 is converted to HEX by substituting the HEX digit $6 for %0110 and HEX digit $C for %1100. Thus % 0110 1100 equal $6C. Pretty neat, huh?

Converting from HEX to binary is just as simple. Look up each HEX digit in the chart above and substitute the corresponding group of four bits. For example, $F2 = % 1111 0010.

Since one BYTE consists of eight BITS (two HEX digits), the largest value that can be stored in one BYTE is %1111 1111 = $FF = 255 DECIMAL. With two BYTES we have 16 BITS (four HEX digits), which allows us to store values up to %1111 1111 1111 1111 = $FFFF = 65535 DECIMAL. All locations in the Commodore 64's memory have a two-byte ADDRESS associated with them. Thus the highest address possible is $FFFF = 65535 DECIMAL. This number is called 64K (1K = $0400 = 1024 DECIMAL)

So much for BINARY-HEX. Let's move on to DECIMAL-HEX conversions. We've already seen how to convert from HEX to DECIMAL, but we need to be able to go the other way, from DECIMAL to HEX. This will be required on a regular basis in machine language programming.

Let's do an easy one. Very often you will see a SYS command in a program listing. This command will execute an ML routine located in the computer's memory. The number you see after the SYS is the DECIMAL equivalent for the ML routine's location. For example, you might see a SYS 2049 in a program. Since most ML monitors use HEX only, it would be your job to convert 2049 DECIMAL to its HEX equivalent before you could investigate the ML routine through a monitor. Let's do it.

Since 2049 is larger than we can store in one HEX digit (limited to 15 = $F), we know we'll need several HEX digits. Start by dividing 2049 by 16. The answer is 128 with a remainder of 1 (128x16=2048). The remainder 1 is taken as the least significant HEX digit (rightmost digit; one's digit). What about the other HEX digits? Since 128 is still larger than we can store in a single HEX digit, we have to divide 128 by 16 again. This gives us an answer of 8 with a remainder of 0 (8x16=128). The remainder 0 is taken as our second HEX digit. Since 8 CAN be represented by a single HEX digit, we also have found our third digit and can stop. Thus 2049 DECIMAL = $801. However, most ML monitors require HEX numbers to contain an even number of digits, so we'll pad our result with a 'leading' zero to give us $0801. This won't change the value, of course.

By the way, some ML monitors such as HESMON have built-in HEX-DECIMAL and DECIMAL-HEX conversion functions. This can greatly simplify your ML programming. Still, there is no substitute for actually knowing how to do these conversions yourself.

USING THE LOMON MONITOR

Load and execute LOMON with SYS 32768. Notice the number in the SYS command. Our monitor resides at HEX $8000. Since we are starting our monitor up from BASIC, we must tell the computer in DECIMAL where the monitor is located. The DECIMAL equivalent for HEX $8000 is 32768 (verify this yourself for practice).

Let's investigate the monitor. When you activate LOMON, you should see the following display:

```
B*
   PC  SR AC XR YR SP
.;803E 32 00 83 00 F6
```

The B*, that you see, indicates that we have entered the monitor by way of a BRK. This is similar to a STOP command in BASIC.

The second line contains the labels for the third line: PC (PROGRAM COUNTER), SR (STATUS REGISTER), AC (ACCUMULATOR), XR (X REGISTER), YR (Y REGISTER) and SP (STACK POINTER). In order to understand machine language, we must investigate these REGISTERS.

PROGRAM COUNTER
The program counter is a 16-bit register which contains the address of the next instruction to be executed. It is merely two 8-bit locations used together. After the program counter is used to get a byte from memory it is incremented by 1, pointing it to the next memory location to be used.

STATUS REGISTER
The status register is an 8-bit register that contains all the FLAGS. A flag is a one-bit value which is said to be SET if ON (=1) and CLEAR if OFF (=0).

```
7 6 5 4 3 2 1 0
N V - B D I Z C
```

N FLAG - Negative flag. Always equal to the leftmost bit of the most recently altered register. Also affected by BIT command.

V FLAG - Overflow flag. Affected by addition (ADC), subtraction (SBC) and bit test (BIT) commands. Mostly used for arithmetic in which the numbers are considered to be signed.

- - Bit five is not used. It is usually found to be SET (=1).

B FLAG - Break flag. SET to 1 after a BRK instruction is executed; CLEAR otherwise. This helps distinguish a BRK interrupt from an IRQ (see the chapter on interrupts).

D FLAG - Decimal mode flag. Changes operation of add and subtract instructions from BINARY (D=0) to DECIMAL (D=1). Always CLEARed on RESET in Commodore machines. Can be SET to 1 with SED (SEt Decimal mode) or CLEARed to 0 with CLD (CLear Decimal mode).

I FLAG - IRQ Interrupt disable flag. Prevents an IRQ interrupt signal from being recognized. SET to 1 with SEI (Set IRQ disable) and CLEARed to 0 with CLI (CLear IRQ disable).

Z FLAG - Zero flag. Used for comparisions, it will be SET to 1 if comparision is equal; otherwise it will be CLEARed to 0.

C FLAG - Carry flag. Tests for greater than or equal to conditions after comparisons with CMP, CPX or CPY. It will be SET to 1 if the register (A, X or Y) is greater than or equal to the compared value. CLEARed to 0 if the register is smaller than the value.

ACCUMULATOR
This is the busiest register. Most of our operations will use the accumulator.

X REGISTER
An index register. Used mainly as an offset for memory references. By incrementing or decrementing X you can step through memory conveniently.

Y REGISTER
Another index register. Similar in function to X.

STACK POINTER
Before we can understand the STACK POINTER, we must take a side trip into the workings of the STACK. The STACK is located in memory from $0100 to $01FF. Its main function is to preserve the return address during subroutine execution. This function is carried out automatically. When a subroutine is executed, the return address is pushed onto the STACK. The last address put on the stack is always the next one available to be pulled off. When the RTS is encountered (Return from Subroutine), the top address on the STACK is pulled off and used as the return address. If the stack has not been disturbed this address will be the correct one.

Think of the STACK as a stack of plates. When you add a plate to the stack, you will put it on top of the plates that are already there. When you need to remove a plate, you must take the top one off before you can safely get to the one below it. The same principle works with our computer's STACK. When we execute a subroutine with JSR, the computer places the return address on the STACK. This way it knows where to return to when it encounters an RTS. There are also commands available to us to manipulate the STACK directly. Care must be taken to

properly prepare for this action, before and after the operation. If we cause the wrong address to be pulled off the STACK, we may crash the operating system. The processor will try to return to the wrong location, which may contain invalid instructions.

Once retrieved from the stack, the return address is placed into the program counter and incremented by one. It now points to the next instruction after the JSR which called on the subroutine. The STACK is used backwards starting from $01FF down to $0100. The STACK POINTER keeps track of the next available location on the STACK. Since the high byte of the stack address is always assumed to be $01, the STACK POINTER holds only the low byte. For example, if the next available stack location was $01A0, the STACK POINTER would have the value $A0.

ENOUGH ALREADY - LET'S LEARN BY DOING!

The best way to learn is to work with a problem. We've presented a lot of 'heavy stuff', so we'll let that settle and take a programming break. Through this break, we will introduce some more concepts such as: a few of LOMON's commands, addressing modes, and machine language instructions.

We are going to begin by entering a machine language program. In order to do that, we must go into ASSEMBLY mode. We will place our program at $C000. We chose this area because there is no fear of our program being overwritten by BASIC. We will lead you through the program and then comment on the code.

1). If you have LOMON activated, your cursor should be blinking beside a '.'. TYPE A C000 LDA #$48 and press the RETURN key. If you did that correctly, you should see .A C002 on the next line with the cursor beside it. If you typed something that the monitor did not like, you will be prompted with a question mark (?). If you made an error, just hit the return key and repeat the first instruction. Your cursor is now blinking beside the .A C002. (How about that, automatic line numbers!)

2). TYPE JSR $FFD2 and press RETURN. Notice that we did not have to type the A again. The computer is now in Assembly mode and will stay that way until we press the return key to exit this mode. The computer will return with .A C005.

3). TYPE BRK and press RETURN

4). Now press RETURN to take us out of Assembly Mode.

5). Check that your code is correct. We will do this through the DISASSEMBLY MODE. Type D C000 C005 and press RETURN. This is the start and end locations of your code. You should see the following:

```
., C000 A9 48    LDA #$48
., C002 20 D2 FF JSR $FFD2
., C005 00       BRK
```

6). TYPE G C000 and press RETURN to execute the code.

If you did everything correctly, you should have been returned to the monitor after execution. The letter 'H' should appear above the B* from the monitor start-up display. Not too exciting, but it is a beginning. We will now explain how our H was printed. Refer to the disassembly in step 5 above. The first column of each line contains the memory address of the corresponding instruction. This is like line numbers in BASIC. The set of columns is the MACHINE CODE (HEX) for the instruction. The last section is the ASSEMBLY CODE (MNEMONIC) version of the instruction.

Here's what we did through the instructions we typed in.

., C000 A9 48	LDA #$48	We placed the value of $48 into the accumulator. This represents the ASCII letter H.
., C002 20 D2 FF	JSR $FFD2	Jumps to a built-in ROM subroutine that prints a character for us. Since we did not specify a device, the character will print to the screen. We could print to the printer, cassette, and disk drive also. As with all subroutines it ends with an RTS, which returns to the next instruction in our program.
., C005 00	BRK	This will stop program execution and jump back to the monitor. This is like a BASIC STOP command.

Let's look at the program in another mode and learn another command in the process. First press return twice to exit D mode. Now type M C000. You should see the following:

```
.:C000 A9 48 20 D2 FF 00 00 00
```

This display tells us what is stored in memory at C000, C001, C002, etc. The A9 is stored in memory location $C000, the 48 is stored in memory location $C001, and so on. Notice that the only difference between this display and our DISASSEMBLY is that the Assembly code is missing. Only the MACHINE CODE is presented in the MEMORY DISPLAY MODE. Our program occupies the

first six bytes. We placed 00's in the last two bytes, but they could be anything. They will contain whatever was left there upon power-up. It doesn't matter what these bytes contain, because our program will stop executing when it encounters the BRK at $C005. Remember, a BRK in machine language is like a STOP in BASIC.

Before we get into all that we have experienced, let's try one more thing. You should still be in M mode and your cursor should be on the second character (:) of the MEMORY DISPLAY .:C000 A9 48 etc. Using your cursor key, move over to the 48 and change it to a 49. When you press RETURN, the new value will be entered into memory. Exit M mode - Remember how? TYPE RETURN. Now type G C000. You should now see an I above the B*. By changing the 48 to a 49, we loaded the accumulator with the ASCII code for the letter 'I' instead of 'H'.

WHAT WE HAVE LEARNED

MONITOR COMMANDS:

A - ASSEMBLE command - This command allows us to enter a machine language program using ASSEMBLY LANGUAGE (MNEMONICS). This is the normal and most convenient method.

D - DISASSEMBLE command - We can check our code at any time using this command. If the listing is extensive, we can scroll up or down through the code with the cursor keys.

G - GO command - This command allows us to execute the program. You may begin execution at any location you wish by specifying the address after the G. This is particularly useful for checking a subroutine. A subroutine (JSR) will end with a return subroutine (RTS). If you place a BRK in the place of the RTS, you can execute the code in question with a G. When it encounters the BRK we will be returned to the monitor. If you type G with no address given, it will use the address shown for the PC in the REGISTER DISPLAY (see below). The G command is similar to a BASIC RUN command.

M - MEMORY COMMAND - This will display the HEX values in an area of memory, without any assembly code.

ADDITIONAL COMMANDS

C - Compare command - Will allow us to compare sections of memory and will return the addresses that contain a difference. TYPE C C000 C005 C100. The computer responds with: C004 C003 C002 C001 C000. This tells us that C104, C103, C102, C101 and C100 contain different values than those found at $C004, C003, C002, C001, and C000. The only address that was not listed was C005. This indicates that C005 and C105 both contain the same value. TYPE D C005. Now TYPE D C105. Both addresses should contain a $00 (BRK).

Your results may vary depending on what is left over in memory from previous operations.

F - The FILL command allows us to clean up memory. Upon power-up, we find 'garbage' throughout memory. We can clean this up with the FILL command. Type F C008 CFFF 00. We have just filled the memory from C008 through CFFF with 00 (BRK's). All of the 'garbage' has been replaced.

H - The HUNT command will allow us to search for a specific sequence of bytes in memory. Type H C000 CFFF A9 49. We are asking the computer to search through the area $C000-CFFF for the bytes A9 49 (LDA #$49). We must use the MACHINE CODE (HEX) version of the instruction when HUNTing. The computer responds with C000. This tells us that these bytes were found starting at C000. You can also search for the ASCII equivalent of bytes by putting a SINGLE quote (') before them. Try H C000 CFFF 'I to look for the I ($49). It should be found at $C001.

I - Interpret command - Displays the contents of memory in HEX values and ASCII characters side by side.

L - Load command - Allows us to load a program from disk or tape. For disk, you would type: L 'PROGRAM NAME',08

R - Register Display - Displays the current contents of the registers. TYPE R. You will find that as a result of our program, the registers have changed. The PC points to C005, SR has changed to 30 as a result of our BRK, and AC now contains a 49 (ASCII value for I). The 49 was loaded into the accumulator by our program. The other registers have stayed the same, because our program did not affect them.

S - Save command - Allows us to save a program. You would type: S 'PROGRAM NAME',08,C000,C006. We first give the device number (08) then the area of memory to save. Our program only extends from $C000 to $C005 but we have to give the ending address PLUS ONE (C005+1=C006). This extra byte is not saved; C005 would be the last byte saved. DON'T FORGET TO ADD ONE TO THE ENDING ADDRESS!

T - Transfer command - Allows you to make a copy of a section of memory to another area. TYPE T C000 C005 C100. Now cursor up and change the T to a C to compare the copy with the original. No addresses will be listed because these two section of memory are now identical. Check it with D C000 C005 and D C100 C105. As you can see, the code is identical. We will leave it up to you to clean up the C100-C105 area with the F command.

X - EXIT to BASIC - This command allows you to return to BASIC. If we were to EXIT now, the monitor and our program would still be in memory. Unfortunately, some info that BASIC needs may be gone, so we probably can't RUN a BASIC program. We can still execute a SYS 32768 to restart the monitor if desired.

Other monitors provide additional commands such as TRACE, VERIFY, and PRINTER OUTPUT and HEX-DECIMAL conversions. We recommend a cartridge-based monitor such as HESMON for the more sophisticated user. This type of monitor provides some options not available on a disk-based monitor.

ADDRESSING

IMMEDIATE ADDRESSING

Through the LDA #$49 instruction, we told the computer that we want to load a value into the accumulator. The data to be loaded into A was given directly in the next byte (49) after the LDA instruction (A9). This is called immediate addressing. We MUST include the pound sign (#) to distinquish between immediate and absolute addressing (see below). Failure to use # for immediate addressing is a common error when first learning ML programming. You can pronounce the # as 'with the value' as in 'Load A with the value $49'.

ABSOLUTE ADDRESSING

Rather than specifying the data for LDA directly as in immediate addressing, we can instead specify the LOCATION of the data. This is called absolute addressing. An example of this would be LDA $C020. In this case the CONTENTS of location C020 will be loaded into A, rather than C020 itself. Absolute addressing is actually much more common than immediate addressing, which is why no special symbol like # is used to indicate it. In our program, JSR $FFD2 utilized absolute addressing. The FFD2 was not an instruction itself but rather the LOCATION of an instruction. We instructed the computer to execute the instructions starting at memory location $FFD2.

ADDITIONAL ADDRESSING MODES

There are approximately 13 address modes used by the 6510 processor. Space will not permit the use or explanation of all of them in this chapter. Machine language books will contain a complete explanation of these modes.

OBJECT CODE

When we assembled our code, the computer converted the ASSEMBLY CODE (MNEMONICS)to OBJECT CODE (HEX). OBJECT CODE is called that because it's the whole 'object' of the assembly process. The idea is to allow us humans to deal with easily remembered commands (assembly mnemonics) like LDA and have the assembler convert them to HEX numbers like A9, which the computer understands.

Our program used a JSR instruction. JSR is actually a mnemonic (memory aid). The assembler converted the mnemonic to the corresponding HEX code, also called the operation code or opcode. Opcodes are always one HEX byte. For example, the opcode for JSR is 20. Let's examine the rest of the OBJECT code for our program.

```
C000 $A9   - The opcode for load the accumulator
C001 $49   - ASCII code for the letter 'I'
C002 $20   - The opcode for jump subroutine (JSR)
C003 $D2   - Low byte of the memory address $FFD2
C004 $FF   - High byte of the memory address $FFD2
C005 $00   - Opcode for BRK (BREAK)
```

SOME OF THE MORE COMMON MNEMONICS AND THEIR OPCODES

```
RTI = $40  - RETURN FROM INTERRUPT
JMP = $4C  - DIRECT JUMP
EOR = $4D  - EXCLUSIVE OR
RTS = $60  - RETURN FROM SUBROUTINE
SEI = $78  - SET THE IRQ DISABLE FLAG
CMP = $C9  - COMPARE REGISTER TO MEMORY
BNE = $DO  - BRANCH IF NOT EQUAL
```

The complete list is rather extensive. The Programmer's Reference Guide describes all the opcodes starting on Page 256.

Let's get back to programming. The program we about to create will clear the screen, change screen colors, and print a message to the screen.

Begin by cleaning up the work space with F C000 CFFF 00. We'll add a few commands to our list and have a little fun in the process. We will now assume that you know how to get into ASSEMBLY mode. We will not prompt you with the A's (ASSEMBLE), but we will provide the memory addresses for a reference point. You type only the assembly code, not the addresses. To get started in ASSEMBLY mode, you must begin by typing A C000 JSR $E544. This is the first instruction of the program below. If you typed the instruction correctly, you will be prompted with the next memory address (C003). Type the rest of the program as given below. Remember, if you make a mistake, exit ASSEMBLY mode with the return key and retype the line. TYPE the following:

```
C003 LDA #$01
C005 STA $D020
C008 STA $D021
C00B LDX #$00
C00D LDA $C100,X
C010 JSR $FFD2
C013 INX
C014 CPX #$06
C016 BNE $C00D
C018 JMP $C003
C01B BRK
```

You're not done yet. The instruction at $C00D tells us to load the accumulator with the values at memory address $C100, indexed by X (LDA $C100,X). If we are going to pick up some values there, then we had better place them in these memory locations (C100-C105). Following the code, we find that six bytes will be read. The instruction CPX #$06 tells us that.

Let's place the values using the M command at $C100. If you typed M C100, you should have a flashing cursor on the ":". Begin typing after the memory address. Type in the values shown and press return. These value are now stored in memory.

```
.:C100 53 55 50 45 52 21 00 00
```

Before we activate the program, we will advise you that the instruction JMP $C003 will place this program in an endless loop. To break out of the program, press RUNSTOP/RESTORE. Now activate the program by typing G C000. There you have it, a screen full of "SUPER!". Again, this program will not make you a million dollars, but demonstrates a few more programming techniques. Let's get out of the program and analyze the code. Press RUNSTOP/RESTORE to stop. You will be returned to BASIC. Re-activate LOMON, with SYS 32768.

Now let's analyze our program. Through the D command, we can see the SOURCE CODE. We will present a great deal here, so bear with us.

```
C000 20 44 E5 JSR $E544
```

'20' IS THE OPCODE FOR JSR (JUMP TO SUBROUTINE) AND '44 E5' IS STARTING ADDRESS OF THE SUBROUTINE. NOTE THAT THIS ADDRESS IS STORED IN LOW BYTE/HIGH BYTE (REVERSE) ORDER. THIS IS STANDARD PROCEDURE FOR THE PROCESSOR. THIS INSTRUCTION OCCUPIES 3 BYTES OF MEMORY. WE ARE TELLING THE COMPUTER TO GO TO MEMORY LOCATION $E544 AND EXECUTE THE BUILT-IN (ROM) SUBROUTINE LOCATED THERE. IF YOU GET OUT YOUR MEMORY MAP, YOU'LL SEE THAT THIS ROUTINE WILL CLEAR THE SCREEN FOR US. ONCE THIS TASK IS COMPLETED, WE WILL BE

```
                                RETURNED TO OUR PROGRAM, SINCE ROM
                                ROUTINES END WITH AN RTS.

C003 A9 01     LDA #$01         WE WILL NOW LOAD THE ACCUMULATOR
                                (A9) WITH THE IMMEDIATE VALUE $01.
                                NOTICE THE POUND SIGN (#) FOR
                                IMMEDIATE ADDRESSING!

C005 8D 20 D0 STA $D020         '8D' IS THE OPCODE FOR STORE THE
                                ACCUMULATOR. WE WILL STORE THE VALUE
                                FROM THE ACCUMULATOR ($01) INTO
                                MEMORY LOCATION $D020, WHICH IS THE
                                LOCATION FOR BORDER COLOR. AGAIN,
                                NOTICE THAT THE ADDRESS IS STORED IN
                                LOW BYTE/HIGH BYTE ORDER.

C008 8D 21 D0 STA $D021         HERE WE WILL STORE THE VALUE FROM
                                THE ACCUMULATOR ($01) INTO THE
                                BACKGROUND COLOR LOCATION. THE
                                RESULT OF THE LAST THREE
                                INSTRUCTIONS IS TO TURN BORDER AND
                                BACKGROUND TO THE COLOR WHITE. THIS
                                IS THE SAME AS THE BASIC COMMANDS
                                POKE 53281,1:POKE 53280,1.

C00B A2 00     LDX #$00         OUR FIRST EXPERIENCE WITH THE X
                                REGISTER. THE OPCODE FOR LDX
                                IMMEDIATE MODE IS 'A2'. WE WILL
                                INITIALIZE X BY LOADING IT WITH THE
                                VALUE $00. KEEP IN MIND THAT X CAN
                                BE USED AS AN INDEX REGISTER.

C00D BD 00 C1 LDA $C100,X       ANOTHER NEW INSTRUCTION, USING WHAT
                                IS CALLED INDEXED ADDRESSING. THIS
                                INSTRUCTION WILL CAUSE THE COMPUTER
                                TO LOAD A FROM MEMORY LOCATION C100
                                + X. AS WE INCREMENT X WE WILL CAUSE
                                IT TO LOAD FROM SUCCESSIVE MEMORY
                                LOCATIONS.

C010 20 D2 FF JSR $FFD2         PRINT WHAT IS IN THE ACCUMULATOR

C013 E8        INX              INCREMENT X. THE FIRST TIME THROUGH,
                                WE LOADED A FROM LOCATION C100,
                                SINCE X WAS $00. AFTER INX, X WILL
                                CONTAIN AN $01. INX IS SIMILAR TO
                                X=X+1 IN BASIC.

C014 E0 06     CPX #$06         COMPARE X WITH THE IMMEDIATE VALUE
                                #$06. WE WILL PRINT SIX BYTES
                                ALTOGETHER, USING A LOOP SET-UP.
```

```
C016 D0 F5     BNE $C00D      'D0' IS THE OPCODE FOR BRANCH IF NOT
                              EQUAL. WE COMPARED X TO #$06. IF
                              THEY ARE NOT EQUAL, WE NEED TO
                              CONTINUE OUR PRINTING LOOP, SO WE
                              BRANCH BACK UP TO $C00D. IF X DOES
                              EQUAL #$06, IT WON'T BRANCH BUT WILL
                              FALL THROUGH TO THE NEXT INSTRUCTION
                              AT $C018, ENDING THE LOOP. NOTE THAT
                              THE $C00D IS NOT TRANSLATED DIRECTLY
                              INTO HEX CODE, BUT RATHER GIVEN AS A
                              RELATIVE POSITION. THE F5 STANDS FOR
                              A BACKWARDS BRANCH OF 11 BYTES
                              ($0100 - $F5 = $0B = 11 DECIMAL)

C018 4C 03 C0  JMP $C003      '4C' IS THE OPCODE FOR JMP. THIS IS
                              A DIRECT JUMP, LIKE BASIC'S GOTO. AS
                              A RESULT OF THIS INSTRUCTION, THE
                              PROGRAM WILL BE PLACED IN AN ENDLESS
                              LOOP.

C01B 00        BRK            THE PROGRAM WILL NOT REACH THIS
                              INSTRUCTION, BECAUSE OF THE JMP
                              INSTRUCTION BEFORE IT. IT'S GOOD
                              PRACTICE TO INSERT A BRK FOR
                              DEBUGGING PURPOSES.
```

There's a lot to get a hold of here. Go through the explanation until you understand it. Try changing the program to print more characters. If you want to tell it to print more, add them to your message at $C100 and change the CPX to accommodate the additional letters. While we're on the subject, let's look at $C100, with I C100. There's our message. The values next to the message are the ASCII codes for the letters. You will recall that we loaded the accumulator with these values. We printed them through the KERNAL subroutine that prints a character (FFD2).

By the way, the BNE instruction actually tests the Z FLAG. The compare instructions such as CPX look at the difference between the two number to be compared. If there is no difference, the Z (ZERO) flag will be set (1). If there is a difference, Z will be cleared (0) and the BNE will cause a branch.

We saved a great deal of program space and our time by using a loop to do our printing, with the X register as an index. The alternative would be to use a pair of instructions (load the accumulator and jump to the print routine) for EACH byte to print. Loops are one of the elementary techniques used in any type of programming.

You may be asking another question at this time. How did we know which ASCII codes to use and where the screen and border color locations were? These were taken right out of a memory map. Refer to the MEMORY MAP SECTION of this manual.

Through this chapter, we hope we have removed some of the fear associated with machine language programming. This chapter should mark a beginning for those wishing to work with machine language. Don't stop here! Continue your investigation and experimentation. Try altering the examples given by adding features to them. Investigating your MEMORY MAP will also reveal some interesting locations to work with. Once you have exhausted the possibilities presented here, investigate other machine language programs. There is much to be learned through a study of this kind.

HAVE FUN!

One of the more common forms of program protection is the use of an auto-boot. Programs of this type are located in low memory ($0100-0400). The purpose of a 'boot' program is to load and execute the 'main' program. An AUTO-BOOT that is set up properly and loaded with ,8,1, will do this automatically. This makes the job of the 'unprotector' a bit more complicated, but certainly not impossible. We must understand how a program of this type is constructed before we can begin to unprotect it. We will analyze three such programs.

Let's start with a boot program that will reside from $02A7 through $0303.

$02A7 - $0303

When we check a memory map, we find that $02A7-02FF is an unused area of memory. Just past this at $0302-0303 is a vector called the BASIC warm-start vector. Creating a boot that will load into this area of low memory and replace BASIC's warm-start vector with the program's starting address will result in an AUTO-BOOT. We may examine this type of program through LOMON, or we may make our corrections on the disk. The more recent Track and Sector Editors include a disassembly feature that can be very useful in the examination of a program stored in this area.

If you attempt to capture the code after a reset, you will find that the code has been erased. This is performed through the normal initialization process. (Refer to the chapter on INTERRUPTS for an extensive look at the RESET routine.) Usually, an auto-boot program is used to hide the loading of a second boot program. The second boot will load the main program and do the actual error-checking, then JMP to the proper entry point of the main program. Checking through the code of the first boot will probably reveal the starting address of the second boot.

After a program loads, the BASIC operating system in the C-64 will perform the KERNAL CLALL ($FFE7) subroutine. This subroutine will close all open files and perform an indirect jump based on BASIC's warm-start vector located at $0302-$0303. With that in mind, let's begin the construction of our first auto-boot.

Storing a boot program from $02A7 through $0303 makes it possible for us to utilize BASIC's warm-start vector ($0302-$0303) as a pointer for our program's starting address. We will make this clear through the disassembly of our first auto-boot program. We will design our program to begin at $02A7 and end at $0303. This will store our starting address ($02A7) into BASIC's warm-start vector. When the KERNAL CLALL ($FFE7)

subroutine is called, it will end by jumping to our program through this vector.

LOAD 'AUTOBOOT1',8,1 from your program disk. This program will automatically load and run the program called 'ATB1'. The disassembly of 'AUTOBOOT1' is as follows:

```
02A7 JSR $E544     CLEAR THE SCREEN
02AA LDA #$83      RESTORE BASIC'S WARM-START VECTORS
02AC STA $0302     IF WE DO NOT PLACE THE NORMAL VALUES IN
02AF LDA #$A4      THESE VECTORS, OUR PROGRAM MAY GO THROUGH
02B1 STA $0303     AN ENDLESS LOAD
02B4 LDA #$9B      LOAD AND STORE THE COLOR GREY INTO:
02B6 STA $D020     BORDER
02B9 STA $D021     BACKGROUND
02BC LDA #$00      LOAD AND STORE THE COLOR BLACK INTO:
02BE STA $0286     CURRENT CURSOR COLOR
02C1-02C9 NOP      SPACE FOR EXTRA CODE IF NEEDED
02CA LDA #$37      NORMAL VALUE FOR BASIC
02CC STA $01       STORE AT $01
                   IF YOUR PROGRAM REQUIRES THAT YOU FLIP
                   OUT BASIC, YOU WOULD STORE A #$36 IN
                   LOCATION $01.
02CE LDA #$08      FILE NUMBER
02D0 LDX $BA       CURRENT DEVICE NUMBER
02D2 LDY #$01      SECONDARY ADDRESS
02D4 JSR $FFBA     KERNAL SETLFS - SET FILE SPECIFICATIONS
02D7 LDA #$04      LENGTH OF FILE NAME
02D9 LDX #$F0      LOW BYTE OF FILE NAME MEMORY ADDRESS
02DB LDY #$02      HIGH BYTE OF FILE NAME MEMORY ADDRESS
02DD JSR $FFBD     KERNAL SETNAM - SET FILE NAME
02E0 LDA #$00      SELECT LOAD FUNCTION
02E2 JSR $FFD5     KERNAL LOAD - LOAD RAM FROM A DEVICE
02E5 JSR $A68E     WE WILL NOW RESTORE BASIC POINTERS
02E8 JSR $A660     CLOSE ALL FILES AND INITIALIZE BASIC
02EB JMP $A7AE     BASIC'S INTERPRETER LOOP
                   ONCE WE ARE FINISHED LOADING, WE WILL JUMP
                   TO BASIC BECAUSE THE PROGRAM WE WILL BE
                   LOADING WILL BE STORED THERE.
                   IF YOUR PROGRAM IS IN MACHINE CODE, YOU
                   WOULD JUMP TO THE ENTRY POINT HERE.
02EE-02EF NOP      EXTRA SPACE
                   NEXT FOUR BYTES ARE FILE NAME IN HEX
02F0 41            A
02F1 54            T
02F2 42            B
02F3 31            1

02F4-02FF BRK      SPACE FOR LONGER FILENAMES
0300 8B            DEFAULT VALUE - DO NOT CHANGE
0301 E3            DEFAULT VALUE - DO NOT CHANGE
0302 A7            LOW BYTE OF OUR PROGRAM START ADDRESS
0303 02            HIGH BYTE OF OUR PROGRAM START ADDRESS
0304 7C
```

For those just starting out, we feel a bit more explanation is in order.

There are many important concepts to be learned from this boot construction. One important task is to restore BASIC pointers. If your second program is stored in BASIC, the interpreter must be intact. Failure to reset pointers may cause your program to lock up. If your program is in machine language, you won't have to worry about the 'clean-up' process.

The starting address and the program name may be changed through the use of the M command of your ML monitor. To change the name of the program to be loaded, simply store the new programs name at $02F0. Use the M command to examine the area of memory from $02F0 to $02F7 (M 02F0 02F7). Now type in the HEX (ASCII) values for your program name, beginning at $02F0, then press 'RETURN'. The same process is used to store our program's starting address at $02EC and $02ED. Remember, the program's starting address must be stored in the standard low byte/high byte fashion.

It should also be noted that we are using the most common KERNAL calls. You may find programmers using $FFB4 (COMMAND SERIAL TO TALK), $FFB1 (COMMAND THE SERIAL BUS TO LISTEN) and others. The KERNAL calls are still easy to spot, because they begin with $FF--. Keep your memory map handy when you are tracing a program. This auto-boot made it easy for us to see the next file to be loaded, because the LOAD message is printed on the screen. Other programmers won't be so considerate. By inserting the KERNAL routine $FF90 (CONTROL KERNAL MESSAGES), we may hide the load message for the next program to be loaded.

Our second auto-boot example works by changing the KERNAL CLALL VECTOR (whereas the first auto boot used the BASIC warm start vector). The KERNAL CLALL VECTOR is located in memory at $032C-$032D. Just past this vector, at $0334-033B, is an unused area of memory followed by the cassette buffer at $033C-03FB and another unused area at $03FC-03FF. This gives us plenty of room to put our auto boot program ($032C-$03FF).

$032C-$032D KERNAL CLALL VECTOR - CLOSE ALL FILES AND I/O CHANNELS

Through this programs construction, we will change the KERNAL CLALL VECTOR to point to our program's starting address, which is $0334. The normal operation of the CLALL VECTOR is to close all files that have been opened. The CLALL is part of BASIC's normal load routine. As such, it is called automatically at the end of a BASIC load, either direct from the keyboard or from a program. Before we construct the boot, let's trace the KERNAL CLALL routine.

```
FFE7 JMP $F32F
F32F LDA #$00
F331 STA $98       NUMBER OF OPEN FILES=0
F333 LDX #$03
F335 CPX $9A       DEFAULT OUTPUT DEVICE NUMBER
F337 BCS $F33C     SMALLER THAN 3
F339 JSR $EDFE     SEND UNLISTEN COMMAND
F33C CPX $99       DEFAULT INPUT DEVICE NUMBER
F33E BCS $F343     SMALLER THAN 3
F340 JSR $EDEF     SEND UNTALK COMMAND
FE43 STX $9A       RESET OUTPUT TO SCREEN
F345 LDA #$00
F347 STA $99       RESET INPUT TO KEYBOARD
F340 RTS
```

Now let's look at our second auto-boot. Load the program, through LOMON, with L 'AUTOBOOT2',08. The disassembly is as follows:

```
032C 34 ???        USE THE M COMMAND TO STORE OUR AUTO BOOT'S
032D 03 ???        STARTING ADDRESS - DON'T FORGET LOW BYTE
FIRST!
032E - 0333        NO CHANGES IN THIS SECTION OF MEMORY
0334 JSR $FF8A     RESTORE DEFAULT VECTORS
0337 JSR $FFE7     CLOSE ALL FILES
033A LDA #$02      FILE NUMBER
033C LDX $BA       CURRENT DEVICE NUMBER (08)
033E TAY           SECONDARY ADDRESS ($02)
033F JSR $FFBA     SET FILE SPECIFICATIONS
0342 LDA #$04      LENGTH OF FILE NAME-4 BYTES LONG
0344 LDX #$5D      LOW BYTE OF FILE NAME MEMORY ADDRESS
0346 LDY #$03      HIGH BYTE OF FILE NAME MEMORY ADDRESS
0348 JSR $FFBD     SET FILE NAME
034B LDA #$00      SELECT LOAD FUNCTION
034D JSR $FFD5     LOAD RAM FROM A DEVICE
0350 STX $2D
0352 STY $2E       SET START OF BASIC VARIABLES
0354 JSR $A68E     PROGRAM POINTER TO BASIC START
0357 JSR $A660     CLOSE FILES AND INITIALIZE BASIC
035A JMP $A7AE     BASIC'S INTERPRETER LOOP
                   NEXT FOUR BYTES ARE FILE NAME IN HEX
035D 41            A
035E 54            T
035F 42            B
0360 32            2
```

Once you have examined the code, you may see it in action with, LOAD 'AUTOBOOT2',8,1 and press RETURN. The program will load and execute a second program called 'ATB2'. As with AUTOBOOT1, the program we are loading is stored in BASIC. If the program is to execute properly, we must restore BASIC's pointers. BASIC's pointer are reset by the code from $0354 to $035C. We may also use this auto boot to load a machine language program, by replacing the JMP to BASIC's INTERPRETER LOOP with a JMP to your starting address. If you have stored routines beneath the BASIC ROM don't forget to add the code to flip-out BASIC.

The second auto boot is also a simple boot. Keep in mind that a RESET of the computer will erase this code through the initialization process. Notice also that our code extends into the CASSETTE BUFFER ($033C-$03FB). Most disk based programs will not have any use for the cassette buffer.

Both of the auto boot programs we have investigated so far are a source of aggravation to the 'unprotector', but the code is still accessible. As long as we know where they are stored, we may LOAD and examine the code from a machine language monitor. The 'unprotector' may be faced with a job of hunting through memory for the auto boot, but at least the code is accessible. Not so with the next type of auto-boot. The program is 'AUTOBOOT3'. It will load and execute 'ATB3'. LOAD 'AUTOBOOT3',8,1. If you try to load this program through a monitor, you will find that the program takes control of the computer. All efforts to regain control are foiled. A RESET will only erase the code. This is due to the construction of the program and its place in memory. We will suggest ways to gain access to the code, but first let's cover the construction.

This program will fill the STACK with our starting address. What does it all mean? The concepts here are no more difficult to grasp than those presented in the previous two programs, but they do require a little knowledge of the STACK. Stay with it and its operation should become clear to you. Stack operations are explained in the chapter on machine language, but a review may be in order.

The STACK is located in memory from $0100 to $01FF. The STACK is used starting from its highest memory location $01FF to the lowest $0100. The last address placed on the STACK is the first address pulled out. In this auto boot, we are placing $02 throughout STACK memory ($0100-$01FF). The KERNAL LOAD routine is a subroutine. A subroutine ends with a RETURN FROM SUBROUTINE (RTS). As with all subroutines, the address of the JSR $XXXX is pushed on the stack prior to executing the subroutine. After the subroutine has executed, the return address is pulled off the STACK and incremented. This address is placed in the program counter, which contains the address of the next command to be executed. Once the program counter gets its address from memory, it is incremented by one, pointing to the next memory location to execute. In this boot, $0202 will be pulled off the STACK since it is full of $02's. After it is incremented, it will point to $0203. This will be the start of our program code. We could fill the STACK with other memory locations, but be sure that the bytes you use are the same (03 03, 04 04). We cannot be sure which byte will be pulled off the STACK first, so we make all these bytes identical. This way, we may be sure of where our program will start. Remember, the address pulled off of the stack is incremented by one prior to being placed on the program counter.

```
0100 -01FF 02      THIS ENTIRE AREA WILL BE FILLED WITH 02'S.
0200 BRK           UNUSED
0201 BRK           UNUSED
0202 BRK           UNUSED
0203 LDA #$04      LENGTH OF FILE NAME
0205 LDX #$39      LOW BYTE OF FILE NAME MEMORY ADDRESS
0207 LDY #$02      HIGH BYTE OF FILE NAME MEMORY ADDRESS
0209 JSR $FFBD     KERNAL SETNAM - SET THE FILE NAME
020C LDA #$02      FILE 2
020E LDX #$08      DRIVE 8
0210 LDY #$02      SECONDARY ADDRESS
0212 JSR $FFBA     KERNAL SETLFS - SET FILE SPECIFICATIONS
0215 LDA #$00      SELECT LOAD FUNCTION
0217 JSR $FFD5     KERNAL LOAD - LOAD RAM FROM A DEVICE
021A STX $2D
021C STY $2E       SET BEGINNING OF BASIC VARIABLES

                   THE FOLLOWING CODE IS FUN AND BRINGS IN A
                   NEW CONCEPT. NOTICE THE VALUES BEING
                   LOADED. IF WE CONVERT THESE TO DECIMAL, AND
                   LOOK UP THE CHR$ CODES, WE FIND THAT THE
                   WORD RUN AND A CARRIAGE RETURN ARE BEING
                   STUFFED INTO THE KEYBOARD BUFFER. THIS WILL
                   RESULT IN AN AUTO-RUN OF OUR BASIC PROGRAM.

021E LDA #$52      R - DECIMAL 82
0220 STA $0277     THE KEYBOARD BUFFER IS LOCATED IN MEMORY
                   FROM $0277 THROUGH $0280.
0223 LDA #$55      U - DECIMAL 85
0225 STA $0278
0228 LDA #$4E      N - DECIMAL 78
022A STA $0279
022D LDA #$0D      CARRIAGE RETURN - DECIMAL 13
022F STA $027A
0232 LDA #$04      LOAD AND STORE 4 INTO
0234 STA $C6       NUMBER OF CHARACTERS IN KEYBOARD BUFFER
0236 JMP $A474     BASIC'S READY MESSAGE, READ KEYBOARD
                   NEXT FOUR BYTES ARE FILE NAME IN HEX
0239 41            A
023A 54            T
023B 42            B
023C 33            3
```

The main question before us is how to gain access to the code. The easiest way is to purchase a Track and Sector Editor that contains a disassembly feature. You may then examine the code and make the necessary changes on the disk. If you do not have such a program, there is another way.

Once you have determined, through your Track and Sector Editor, that the program resides at $0100, you may change the starting address to another value, say $C100 (see the PPM volume I). You may accomplish this by locating the first block of the file in question and change byte 04 from 01 to C1. Remember, the 3rd and 4th bytes contain the starting address of the program. Once

this address has been changed, we may load it normally and examine the code through LOMON. The code is now located from $C100 - $C23C. You must keep in mind that the code would normally reside at $0100, so you must think of the 'C's' as '0's'. From here you may make any necessary changes. You would now save out the altered program in the standard manner. The last step is to go back in with your Track and Sector Editor and change the starting address back from C1 to 01.

The program called 'AUTOBOOT3C100' on your PROGRAM DISK is a copy of 'AUTOBOOT3', but it resides at $C100. Load the program and compare the code with the original version included here. You will find that the only difference is in where the code resides in memory.

If you wish to use an auto-boot program that resides at $0100 and above, you must construct it in another area of memory and change the load address on the disk. We suggest that you construct it at $C100.

This type of auto-boot program requires that you work with your Track and Sector Editor. The PROGRAM PROTECTION MANUAL VOLUME I contains all the information you'll need to make alterations on the disk, but for your convenience we will review a bit here.

Let's take a look at a typical TRACK 18 SECTOR 01. This is the first block of the DISK DIRECTORY. There is a great deal of information contained in a DIRECTORY listing. It will tell us the names of the files contained on the disk, the file type, the location of the files on the disk, and the number of blocks in each file. This should become clearer through the print-outs included here. Let's take a look:

ASCII MODE

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

0 . . . .  . M O V  E   B A  S I C
1          . . .  . . .  . . .  . . .
2 . . . .  . D I S  K   C H  E C K E
3 R        . . .  . . .  . . .  . . .

4 . . . .  . I D    C H E C  K E R
5          . . .  . . .  . . .  . . .
6 . . . .  . A P P  E N D
7          . . .  . . .  . . .  . . .

8 . . . .  . B L O  C K   A  L   &
9 F R E E    . . .  . . .  . . .  . . .
A . . . .  . D I S  K   A D  D R   C
B H A N G  E . . .  . . .  . . .  . . .

C . . . .  . D I S  K   D R
D          . . .  . . .  . . .  . . .
E . . . .  . B A C  K U P    2 2 8
F          . . .  . . .  . .  % .
```

As you can see, there are eight programs listed on this block of the directory. ASCII mode is very helpful, but it is HEX mode that will reveal the information we will need to locate our files. The next printout will be the HEX listing of 18/01.

HEX MODE

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

0 12048211 004D4F56 45204241 534943A0
1 A0A0A0A0 A0000000 00000000 00000900
2 00008211 01444953 4B204348 45434B45
3 52A0A0A0 A0000000 00000000 00000300

4 00008211 09494420 43484543 4B4552A0
5 A0A0A0A0 A0000000 00000000 00000400
6 00008211 09494420 43484543 4B4552A0
7 A0A0A0A0 A0000000 00000000 00000100

8 00008211 03424C4F 434B2041 4C202620
9 46524545 A0000000 00000000 00000300
A 00008211 11444953 4B204144 44522043
B 48414E47 45000000 00000000 00000400

C 00008213 00444953 4B204452 A0A0A0A0
D A0A0A0A0 A0000000 00000000 00001700
E 00008210 00424143 4B555020 323238A0
F A0A0A0A0 A0000000 00000000 00002500
```

We will analyze the code in the first directory entry. If you back up to ASCII, you will see that this is the file called 'MOVE BASIC'. We chose this file because it is the first entry and contains some additional information. Use the grid for reference points.

GRID	HEX CODE	INFORMATION
0/0	1204	These two bytes contain the HEX values for the link to the next track and sector. The decimal eqivalent is 18/04. The next block of our directory will be at TRACK 18, BLOCK 04.
0/2	82	This byte is where the type of file is given. The 82 tells us that this is an active (not scratched) program file. Check the P.P.M. VOL.I for a description of the other file types.
0/3	1100	These two bytes contain the Track and Sector for the first block of this file. The decimal eqivalent is 17/00. The first block of the file called 'MOVE BASIC' will be located at TRACK 17, BLOCK 00.
0/5	4D4F	='MO' This is the beginning of our program name. The name 'MOVE BASIC' will end at 0/E. Sixteen bytes are reserved for a program name. If the name is shorter than the space reserved, the space will be filled with shifted spaces (A0's in HEX).
1/5	000000	These three bytes are reserved for relative file entries. They would contain pointers used by files of this type.
1/8	00000000	These four bytes are normally 00.
1/C	0000	These two bytes are reserved for the DOS. They will be used during a Save and Replace operation.
1/E	0900	The last two bytes tell us the number of blocks that the program occupies on the disk, in low byte/high byte order. MOVE BASIC occupies 9 blocks.

The other file entries follow the same format. The only difference is in the first two bytes of the entry, which will contain 00/00.

As you can see, the directory can offer a great deal of assistance to those who know how to read it. We will now examine the first block of the program called 'AUTOBOOT2', from the disk that accompanies this manual. When you examine the directory of the P.P.M.VOL.II disk, you find that 'AUTOBOOT2' is a program file (82), bytes three and four tell you its location on the disk, and you learn that it occupies 1 block on

the disk ($01 00). The first block of a file contains some very special information. Let's take a look at that block in HEX MODE.

'AUTOBOOT2' - HEX MODE

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

0 00392C03 340366FE A5F4EDF5 208AFF20
1 E7FFA902 A6BAA820 BAFFA904 A25DA003
2 20BDFFA9 0020D5FF 862D842E 208EA620
3 60A64CAE A7415442 32000000 00000000

4 00000000 00000000 00000000 00000000
5 00000000 00000000 00000000 00000000
6 000082AA 48B2B528 4D54AD32 3536293A
7 4CB24D54 AB323536 AC483A97 34332C4C

8 3A973434 2C483A80 00000054 494E5545
9 22009E0A F401A141 243A8B41 24B22222
A A7353030 00A40AFE 018E0000 00414C59
B 4A454422 00BB08BE 008D3239 30004C59

C C800444E B23800E0 08DC0085 22931111
D 20205748 49434820 54524143 4B20223B
E 545200EA 08DD0053 4BB23230 00FB08DE
F 008B5452 B13137A7 53455B31 38000C09
```

GRID	HEX CODE	INFORMATION
0/0	00	This byte give us the next track in the program link. In this case, the 00 tells us that this is the last block of the file.
0/1	39	This byte would normally contain the sector for the next link in the file. Since this is the last block, this byte tells us and the disk drive where the program ends. Check the grid at 3/9. This is the last good byte of information for this file. The rest of the information on this block is 'garbage'.
0/2	2C	This is the low byte of our memory address. This is a MACHINE LANGUAGE program stored at $032C.
0/3	03	This is the high byte of our memory address. If you would like to change the load address, you would change these two bytes.
0/4 - 3/9		This is the data for the program.

Reading the information contained on the disk is essential to those wishing to study 'AUTOBOOTS'. Beginning here can save you a great deal of time. Your first attempts at altering disk information should be done on a BACKUP disk. Do not make alterations to the original disk if you can avoid it. If you

must, then make a note of the changes you made so that you may return the disk to its original form.

You will find that all of this information will become clearer with experience. Begin by working with the examples we have included on your PROGRAM DISK.

There are three types of interrupts built in to the 6510 processor used on the Commodore 64. An interrupt is a way to force the processor to stop what it is doing and execute another set of instructions. It is different from a RESET in that it is only temporary, that is, the processor can pick up where it left off before it was so rudely interrupted. In this chapter we will explore each type of interrupt and analyze the corresponding KERNAL ROM routine. As we shall see, each ROM routine is controlled by a RAM vector that may be changed by the programmer. The interrupts are:

INTERRUPT TYPE	ROM VECTOR	RAM VECTOR
NMI - NONMASKABLE INTERRUPT	$FFFA-$FFFB	$0318-$0319
IRQ - INTERRUPT REQUEST	$FFFE-$FFFF	$0314-$0315
BRK - BREAK	$FFFE-$FFFF	$0316-$0317

The initial response to any of the three interrupts is similar. The processor will finish the instruction it is currently executing and then take the following steps:

1). The current value of the program counter, which contains the address of the next instruction, is pushed onto the stack. The high byte is put on the stack first, followed by the low byte. (Note: the stack grows backwards in memory from $01FF to $0100).

2). The processor status register, which contains all the flags (carry, etc.) is pushed onto the stack.

3). The processor then consults a specific place in memory for the interrupt routine. For an NMI it looks to $FFFA-$FFFB and for an IRQ or BRK it looks to $FFFE-$FFFF. These locations contain the STARTING ADDRESS of the corresponding interrupt routine vector. The processor then jumps to the starting address given and begins executing the code there.

WHERE it looks initially ($FFFA-B or $FFFE-F) in step 3 above CANNOT BE CHANGED; the processor is designed at the hardware level to do this. Since the KERNAL ROM normally occupies these locations on the C-64, we cannot easily change the CONTENTS of these locations either. So how can we change what happens at interrupt? As we shall see, the interrupt routines pointed to by these ROM vectors all check another location to decide where to proceed. These other locations are in RAM (RAM vectors), and we CAN alter them.

Once an interrupt routine has finished its job (whatever that may be) it should be able to have the processor resume its operations at the point it was interrupted. To do this it must be able to restore the status register and program counter (PC) to their pre-interrupt values. Remember, these values were

pushed onto the stack in steps 1 & 2, so they are still available. The 6510 processor has a special instruction called RTI (RETURN FROM INTERRUPT) which automatically restores these values. Since it restores the program counter, execution continues at the same point it was interrupted. Every interrupt routine should end with an RTI.

Although they share some similarities, the three interrupts have important differences too. The 6502/6510 microprocessor is housed in a plastic case with 40 connecting pins, two of which are dedicated to IRQ and NMI. When a signal is applied to one of these pins, the corresponding interrupt routine will be executed. This is the only way to generate the IRQ and the NMI interrupts.

The main difference between the IRQ (INTERRUPT REQUEST) and the other two interrupts is that we can prevent (mask) the IRQ signal from being recognized by the processor if we wish. We do this by using the machine language instruction SEI (SET INTERRUPT DISABLE). This instruction affects ONLY the IRQ interrupt. No IRQ signal will be noticed until after we do a CLI (CLEAR INTERRUPT DISABLE) or RTI (RETURN FROM INTERRUPT) instruction.

The second type of interrupt is BRK (BREAK). It differs from the others in that BRK is generated through the use of a special machine language instruction (BRK) rather than an electronic signal. When a BRK occurs it causes a special flag (the BRK flag) to be set in the processor status register. Although BRK and IRQ actually jump to the same ROM routine initially, the BRK flag is used to tell them apart.

The third type of interrupt is the NMI (NONMASKABLE INTERRUPT). As the name implies, NMI cannot be disabled (masked) using SEI. It can occur at any time, even while an IRQ or BRK routine is being executed. In fact, even if an NMI and another interrupt occur simultaneously, the NMI is given priority.

I want to emphasize the similarities and differences noted above are a function of the processor itself, rather than the Commodore 64 as a whole. Now let's take a detailed look at each of the KERNAL ROM routines executed by these interrupts to see what the C-64 uses them for. We'll start with the NMI interrupt.

NMI (NONMASKABLE INTERRUPT) ROM $FFFA-B ($FE43)
RAM $0318-9 ($FE47)

Here is the main NMI interrrupt routine for reference in the following discussion:

```
FE43 SEI           SET INTERRUPT DISABLE (NO IRQ INTERRUPTS)
FE44 JMP ($0318)   NMI RAM VECTOR (CONTAINS $FE47 NORMALLY)
FE47 PHA
FE48 TXA           THIS CODE WILL SAVE THE A, X, AND Y
FE49 PHA           REGISTERS
FE4A TYA
FE4B PHA
FE4C LDA #$7F
FE4E STA $DD0D     CIA #2 INTERRUPT CONTROL REGISTER
FE51 LDY $DD0D
FE54 BMI $FE72     BRANCH IF RS-232 ACTIVE
FE56 JSR $FD02     CHECKS FOR CBM80 AT $8000
FE59 BNE $FE5E     IF NOT, CONTINUE
FE5B JMP ($8002)   IF PRESENT, JUMP TO WARM-START ROUTINE
FE5E JSR $F6BC     SET FLAG FOR STOP-KEY
FE61 JSR $FFE1     SCAN STOP KEY
FE64 BNE $FE72     BRANCH IF STOP KEY NOT PRESSED
FE66 JSR $FD15     RUN/STOP-RESTORE PRESSED - SET I/O VECTORS
FE69 JSR $FDA3     INITIALIZE I/O
FE6C JSR $E518     INITIALIZE I/O AND CLEAR SCREEN
FE6F JMP ($A002)   TO BASIC WARM-START
FE72 TYA
FE73 AND $02A1     NMI INTERRUPT CONTROL CIA
FE76 TAX
FE77 AND #$01
FE79 BEQ $FEA3
FE7B LDA $DD00     DATA PORT A-SERIAL BUS, RS-232
FE7E AND #$FB
FE80 ORA $B5
FE82 STA $DD00
FE85 LDA $02A1
FE88 STA $DD0D     CIA #2 INTERRUPT CONTROL REG
FE8B TXA
FE8C AND #$12
FE8E BEQ $FE9D
FE90 AND #$02
FE92 BEQ $FE9A
FE94 JSR $FED6     RS-232 IN
FE97 JMP $FE9D
FE9A JSR $FF07     RS-232 OUT
FE9D JSR $EEBB     RS-232 OUTPUT
FEA0 JMP $FEB6     RESTORE AND EXIT
FEA3 TXA
FEA4 AND #$02
FEA6 BEQ $FEAE
FEA8 JSR $FED6
FEAB JMP $FEB6
FEAE TXA
FEAF AND #$10
FEB1 BEQ $FEB6
```

```
FEB3 JSR $FF07    RS-232 OUT
FEB6 LDA $02A1
FEB9 STA $DD0D
FEBC PLA          THIS CODE RESTORES THE A,X AND Y REGISTERS
FEBD TAY
FEBE PLA
FEBF TAX
FEC0 PLA
FEC1 RTI          RETURN FROM INTERRUPT AND CONTINUE PROGRAM
```

On the Commodore 64, an NMI can be generated by a device on the RS-232 (user) port or by the RESTORE key. In either case, the processor will consult locations $FFFA-$FFFB. These two bytes contain a vector (pointer) to the interrupt routine in ROM. The values found here are $43 FE, respectively, which means the routine is at $FE43 (remember the address bytes are stored in reverse order). The processor will then proceed to $FE43 and begin executing the routine there.

The routine at $FE43 immediately disables the IRQ (with an SEI instruction) so that it won't be interrupted itself. Next, it consults (through JMP ($0318)) another vector located at $0318 & $0319, which is in RAM. The values found there tell it where to proceed next. Normally, this RAM vector points to $FE47 which simply continues the NMI routine. Since this vector is in RAM, however, it can be easily changed to point to our own routine if desired.

In the normal ROM routine at $FE47, it immediately pushes the values of the A, X and Y registers onto the stack because it needs to use them. Next the NMI routine checks a location on CIA #2 to see if the NMI was generated by an RS-232 device, such as a printer or modem. If so, it jumps to the routine to handle RS-232 communications. We're not concerned with RS-232, so we won't discuss it further. In the following discussion we'll assume the NMI was generated by the RESTORE key.

The NMI routine next checks for the presence of CBM80 at $8000. This indicates an autostart program, usually a cartridge. If the CBM80 is present, the values stored at $8002 & $8003 will be used as a 'warm start' vector. Processing will continue at the location indicated at $ 8002 & $8003 (vectors). We may fool the computer into thinking that a cartridge is present by storing a CBM80 at $8000. This allows a programmer to utilize the NMI routine to restart a program in progress. For more detailed information on the CBM80, refer to the original PROGRAM PROTECTION MANUAL.

If there is no CBM80 at $8000, the NMI routine checks the RUN/STOP key. If it is being pressed it is a signal to warm-start BASIC. In this case the routine performs some I/O initialization and clears the screen. Finally, it consults the BASIC warm-start vector at $A002 & $A003 and jumps to the location specified there.

If RUN/STOP is not pressed the routine will continue through some code and evenually restore the A, X and Y registers from the values that were saved on the stack. Finally it executes an RTI which restores the processor status, re-enables IRQ's and continues execution at the point it was interrupted by the NMI.

Before we continue our study of the other interrupts, let's explore the CBM80 set-up in terms of program protection. During our detour, we'll need to explore the RESET and RAM TEST ROUTINES. An understanding of these routines can be invaluable in program protection.

We find a great deal of built-in security for programs that utilize the CBM80. The auto-start feature for cartridges is designed to keep the code from being exposed on RESET. The same protection is provided to any program that uses CBM80 and the RESTORE key to auto-start itself. We have already explored and utilized the techniques used to protect a cartridge in the PROGRAM PROTECTION MANUAL VOLUME I, but auto-start programs that reside in RAM require another look.

If the CBM80 is in RAM, it can be defeated by simply preventing the computer from seeing this part of memory. A cartridge uses the EXROM and GAME lines of the cartridge port to control the memory configuration of the C-64. See the chapter on the 6510 and the PLA for a complete breakdown of how this is done. For our purposes here we only need to look at the function of EXROM.

Normally, the EXROM line stays at a HIGH level (+5 volts). In this state we will have RAM available at $8000-$9FFF (assuming everything else is normal). If EXROM is forced to a LOW level (0 volts) by grounding it, the computer expects to see cartridge ROM there instead of RAM. However, REGARDLESS of whether there is anything plugged into the cartridge port or not, the computer will NOT be able to see the RAM in this area. If EXROM is grounded after loading and running a CBM80-based program, all of a sudden the program can't see its auto-start when we hit RESTORE. If the computer is RESET, we'll see the familiar blue screen and start-up message, except that we'll see 30719 BASIC BYTES FREE instead of the normal 38911. Also, a $55 will have been put at $8000 (RAM). Why all this happens is a matter for further exploration.

We need to examine the RESET routine with concentration on the RAM TEST routine ($FD50). The RESET routine is located at $FCE2 in ROM. The decimal address for this location is 64738. To execute a RESET from BASIC, we enter SYS 64738. Of course we can also force a RESET through our familiar RESET switch.

RESET ROUTINE

```
  FCE2 LDX #$FF
  FCE4 SEI           PREVENT IRQ INTERRUPTS
  FCE5 TXS           SET THE STACK POINTER TO TOP - IMPORTANT!
  FCE6 CLD           CLEAR DECIMAL FLAG TO ENABLE HEX ARITHMETIC
  FCE7 JSR $FD02     CHECKS FOR CBM80 AT $8000
  FCEA BNE $FCEF     SKIP TO $FCEF IF NOT PRESENT
  FCEC JMP ($8000)   JUMP TO CARTRIDGE COLD START
  FCEF STX $D016     SET SCREEN TO 38 COLUMNS
  FCF2 JSR $FDA3     INITIALIZE I/O
  FCF5 JSR $FD50     *RAM TEST - EXPLAINED BELOW*
  FCF8 JSR $FD15     SET HARDWARE & I/O VECTORS (0314-0333)
  FCFB JSR $FF5B     INITIALIZE VIC CHIP (INCL. COLORS)
  FCFE CLI           ALLOW IRQ INTERRUPTS AGAIN
  FCFF JMP ($A000)   JMP TO BASIC COLD-START
```

Let's break down the $FD50 routine (RAM TEST). This is the routine that initializes the work area and places the $55 at $8000 ($A000 normally). The commented code is as follows:

INITIALIZE WORK AREA - RAM TEST

```
  FD50 LDA #$00      THIS SECTION OF CODE WILL CLEAR ZERO
  FD52 TAY           PAGE, PAGE 2 AND PAGE 3 TO ALL $00'S
  FD53 STA $0002,Y
  FD56 STA $0200,Y   NOTE THAT THE STACK AT $0100-$01FF IS
  FD59 STA $0300,Y   NOT RESET (EXCEPT FIRST TWO BYTES)
  FD5C INY
  FD5D BNE $FD53
  FD5F LDX #$3C      INITIALIZE CASSETTE BUFFER POINTER
  FD61 LDY #$03
  FD63 STX $B2
  FD65 STY $B3
  FD67 TAY           THIS SECTION PERFORMS THE RAM TEST
  FD68 LDA #$03
  FD6A STA $C2
  FD6C INC $C2       START TEST AT $0400
  FD6E LDA ($C1),Y   $C1-2 POINTS TO NEXT BYTE TO TEST
  FD70 TAX           PRESERVE VALUE THERE NOW
  FD71 LDA #$55      TEST PATTERN = BINARY 01010101
  FD73 STA ($C1),Y   TRY SAVING TO BYTE BEING TESTED
  FD75 CMP ($C1),Y   COMPARE VALUE THERE NOW WITH TEST PATTERN
  FD77 BNE $FD88     BRANCH IF NOT THE SAME; WE'VE FOUND ROM
  FD79 ROL           DOUBLE-CHECK WITH $AA = BINARY 10101010
  FD7A STA ($C1),Y
  FD7C CMP ($C1),Y
  FD7E BNE $FD88     BRANCH IF ROM FOUND
  FD80 TXA
  FD81 STA ($C1),Y   RESTORE ORIGINAL VALUE TO BYTE
  FD83 INY           NEXT BYTE
  FD84 BNE $FD6E     BRANCH IF NOT DONE WITH PAGE
  FD86 BEQ $FD6C     NEXT PAGE (256-BYTE AREA)
  FD88 TYA           GET LOCATION OF FIRST ROM BYTE...
  FD89 TAX
```

```
FD8A LDY $C2
FD8C CLC
FD8D JSR $FE2D     ... AND SET TOP OF RAM POINTER TO IT
FD90 LDA #$08
FD92 STA $0282     SET PAGE NO. OF BASIC AREA START
FD95 LDA #$04
FD97 STA $0288     SET PAGE NO. OF SCREEN FOR EDITOR
FD9A RTS           ALL DONE
```

The RAM TEST routine starts at $0400 (screen memory) and works its way up until it finds ROM (or no longer finds RAM). It tests a location by storing a test pattern ($55) into it and then trying to load it back out. If the value it gets back matches the test pattern, then it must be in RAM (it double-checks anyway with another pattern, $AA). Since the routine is not supposed to change RAM, it preserves the value that was there originally and replaces it afterward. However, if it stores out the test pattern and can't get it back, then it assumes it's found ROM. Note that it DOESN'T replace the original value in this case. The first test value of $55 is left in memory.

Now we see why we get a $55 at $8000 when EXROM is grounded and the computer is RESET. EXROM prevents the computer from reading the RAM at $8000-$9FFF. When the routine stores out the $55 to location $8000, it does go into RAM, however, wiping out what was there! Since the routine can't read the $55 back because of EXROM, it thinks it's found ROM and doesn't replace the original value. It also records $8000 (32768) as the start of ROM instead of the normal $A000 (40960). Since the BASIC BYTES FREE is calculated by subtracting $0801 (2049) from this value, we get 30719 instead of 38911.

The next interrupt routine is called the IRQ. Sixty times each second an IRQ interrupt signal is given to the microprocessor by the timing hardware. When an IRQ signal is received, the processor will first check the I flag (IRQ disable flag). If the I flag is set (by a SEI instruction), the signal will be ignored. If the I flag is clear, the IRQ will be allowed. Since an IRQ and a BRK are handled initially by the same same routine, the processor must first check its BRK flag to tell which type actually happened. It then selects the proper routine. The IRQ routine performs several housekeeping chores such as scanning the keyboard. If a program is in progress, operation is suspended to allow for the interrupt sequence. This operation takes place so quickly that we do not notice the interruption.

IRQ (INTERRUPT REQUEST) ROM $FFFE-F (FF48)
                         RAM $0314-5 (EA31)

1). As with the NMI interrupt, the current value of the program counter will be placed on the stack in high byte/low byte order.

2). The status register (FLAGS) will be pushed to the stack.

3). The ROM vector at $FFFE-FFFF is consulted for the actual entry point of the IRQ routine. This vector points to a routine located at $FF48 which will then be executed.

```
FF48 PHA           THIS CODE WILL SAVE
FF49 TXA           THE REGISTERS
FF4A PHA
FF4B TYA
FF4C PHA
FF4D TSX
FF4E LDA $0104,X   GET THE BREAK FLAG FROM THE STACK (BIT 4)
FF51 AND #$10      TEST BRK FLAG - CHECK IF INTERRUPT
                   IS FROM A BRK OR AN IRQ
FF53 BEQ $FF58     BRANCH IF IRQ
FF55 JMP ($0316)   BRK ROUTINE VECTOR
FF58 JMP ($0314)   IRQ ROUTINE VECTOR
```

Take particular notice of the last two addresses. An indirect jump based on the contents of $0316-7 will occur if the BRK flag is set. An indirect jump based on the contents of $0314-5 will be executed if the BRK flag is not set. This RAM IRQ vector points to the ROM routine at $EA31. If you wish to add some code to the IRQ routine, you would change the vectors at $0314-$0315 to point to your section of code. At the end of your code, you should jump to the normal ROM routine at $EA31. This will insure that the normal housekeeping chores are done properly. They are as follows:

1). Update system clock and check STOP key. The system clock at $A0-$A2 (BASIC variable TI) is incremented every sixtieths of a second. Next, the STOP key is checked. If the stop key is pressed a flag in zero page is set.

2). Flash the cursor. Every twentieth time the IRQ routine is called, the character at the cursor position is reversed. This causes the cursor to blink 3 times per second.

3). Perform tape I/O. Datasette operation is handled through the IRQ routine. If the datasette is not being controlled by a program, the motor is switched on or off depending on whether a key on the datasette is pressed or not.

4). Read the keyboard. If a key is pressed, the key code is determined and the corresponding ASCII value is placed in the keyboard buffer.

When all these tasks have been completed, we return from the interrupt with RTI. This automatically clears the IRQ disable flag, restores the status register and resumes the interrupted program.

With the above information in mind, let's take a look at the ROM routine at $EA31.

```
INTERRUPT ROUTINE
EA31  JSR $FFEA    INCREMENT TIME CLOCK
EA34  LDA $CC      CURSOR BLINK: $00=OFF, $01=ON
EA36  BNE $EA61    IF NOT BLINKING, THEN CONTINUE
EA38  DEC $CD      DECREMENT CURSOR BLINK TIMER
EA3A  BNE $EA61    IF NOT ZERO, THEN CONTINUE
EA3C  LDA #$14     SET CURSOR BLINK TIMER TO 20 JIFFIES
EA3E  STA $CD
EA40  LDY $D3      GET CURSOR COLUMN
EA42  LSR $CF      IF BLINK SWITCH IS $80 THEN SET CARRY
EA44  LDX $0287    COLOR UNDER THE CURSOR
EA47  LDA ($D1),Y GET CODE OF CHARACTER UNDER CURSOR
EA49  BCS $EA5C    IF THE BLINK SWITCH WAS ON, THEN CONTINUE
EA4B  INC $CF      TURN BLINK SWITCH ON
EA4D  STA $CE      SAVE CHARACTER UNDER CURSOR
EA4F  JSR $EA24    SYNCHRONIZE COLOR POINTER
EA52  LDA ($F3),Y GET COLOR CODE OF CHARACTER
EA54  STA $0287    CURRENT COLOR CODE UNDER THE CURSOR
EA57  LDX $0286    BACKGROUND COLOR UNDER CURSOR
EA5A  LDA $CE      CHARACTER UNDER CURSOR
EA5C  EOR #$80     REVERSE CHARACTER VIDEO
EA5E  JSR $EA1C    SET CHARACTER AND COLOR
EA61  LDA $01
EA63  AND #$10     CHECK FOR THE TAPE DRIVE KEY
EA65  BEQ $EA71    DETERMINE IF PRESSED
EA67  LDY #$00
EA69  STY $C0      CLEAR TAPE INTERLOCK FLAG
EA6B  LDA $01
EA6D  ORA #$20     TAPE DRIVE ON
EA6F  BNE $EA79
EA71  LDA $C0
EA73  BNE $EA7B    TO CHECK KEYBOARD
EA75  LDA $01
EA77  AND #$1F     TAPE DRIVE ON
EA79  STA $01      INPUT-OUTPUT REGISTER
EA7B  JSR $EA87    CHECK KEYBOARD
EA7E  LDA $DC0D    CIA INTERRUPT CONTROL REGISTER
EA81  PLA
EA82  TAY
EA83  PLA          RESTORE REGISTERS
EA84  TAX
EA85  PLA
EA86  RTI          RETURN FROM INTERRUPT
```

We've looked at the code and analyzed the routines, but what does it mean to us? We will explore that question in terms of how you may utilize the interrupt in your own programming and in terms of protecting a program.

The key to the interrupt sequence is that it will pass through a RAM location. This allows the programmer the opportunity to utilize the interrupt for his own purposes.

Let's get to it. Load and execute LOMON so that we may experiment with the IRQ interrupt. Our task will be to add a border color change to the normal interrupt sequence. Remember, the interrupt occurs sixty times each second. We will point the IRQ RAM vector to our routine at $2000.

1). With LOMON activated, type A 1000 SEI followed by RETURN. This disables the IRQ flag, suspending the operation of the normal IRQ interrupt sequence so that it will not interfer with the job we have chosen to perform. All IRQ interrupt sequences should begin this way.

2). The monitor will respond with the next memory location (A1001). Type LDA #$00 and press RETURN. We are now loading the accumulator with the low byte of the location of our interrupt code.

3). Again the monitor responds with the next memory location (A1003). Type STA $0314 and press RETURN. Through this instruction, we are storing the low byte address of our interrupt routine in the low byte of the IRQ RAM VECTOR.

4). We are now at $1006. Type LDA #$20 and press RETURN. This is the high byte of the location of our routine.

5). Type STA $0315 and press RETURN. We will now store the high address byte of our interrupt routine in the high byte of the IRQ RAM VECTOR.

6). Type CLI followed by RETURN. This instruction will enable the IRQ flag so that IRQ interrupts may occur. This is not really necessary since RTI will do this automatically after an IRQ (only). It's purpose is to remind us that must be done.

7). Type RTI followed by RETURN. This will return us from the interrupt sequence back to the program in progress.

Disassemble the code at $1000. Check your code with the disassembly below. Make sure you have programmed the sequence properly.

PROGRAMING THE IRQ RAM VECTOR

```
.,1000 78       SEI          SET THE INTERRUPT - NO INTERRUPTS
                             ALLOWED.

.,1001 A9 00    LDA #$00     LOAD THE LOW BYTE OF THE
                             INTERRUPT SEQUENCE ADDRESS INTO
                             THE ACCUMULATOR - OUR INTERRUPT
                             ROUTINE WILL RESIDE AT $2000

.,1003 8D 14 03 STA $0314    STORE THE LOW BYTE OF OUR PROGRAM
                             ADDRESS INTO THE LOW BYTE OF THE
                             IRQ RAM VECTOR

.,1006 A9 20    LDA #$20     LOAD THE HIGH ADDRESS BYTE OF OUR
                             ROUTINE INTO THE ACCUMULATOR

.,1008 8D 15 03 STA $0315    STORE THE HIGH ADDRESS BYTE OF
                             OUR ROUTINE INTO THE HIGH BYTE OF
                             THE IRQ RAM VECTOR

.,100B 58       CLI          ALLOW IRQ INTERRUPTS TO OCCUR

.,100C 40       RTI          RETURN FROM INTERRUPT - BACK TO
                             THE PROGRAM IN PROGRESS
```

We will now store our interrupt sequence at $2000.

1). Back to assembly mode. Type A 2000 PHA followed by RETURN. We will preserve the registers so that a program in progress may be resumed when we return to normal program execution. This process must be done through the A register. The PHA instruction will push the accumulator onto the stack.

2). Type TXA followed by RETURN. We will now transfer the X register to the accumulator. Remember we can only push values to the stack through the A register. If we wish to preserve X, we must first transfer it to the A register.

3). Type PHA and press RETURN. We are now pushing the transferred X value to the stack.

4). Type TYA followed by RETURN. We will now preserve the Y register, through a transfer to the A register.

5). Type PHA followed by RETURN. We will push the transferred Y value to the stack.

6). Now that we have preserved our registers, we will go about the task of adding our color change. Type LDA $D020 followed by RETURN. We are now loading the border color into the accumulator.

7). Type CLC followed by RETURN. This instruction will clear the carry flag.

8). Type ADC #$01 followed by a return. This adds memory to the accumulator with carry.

9). Type STA $D020 followed by a return. The results will be stored at the border color location.

10). With our color change done we must now retrieve the registers. Type PLA and RETURN - pull the accumulator from the stack.

11). Type TAY and RETURN. We will transfer that value to the Y register.

12). Type PLA and RETURN. Pull the next value off the stack, which was the X register.

13). Type TAX and RETURN. Transfer the value in the accumulator to the X register.

14). Type PLA and RETURN. This is the last value to be pulled from the stack. It is the value for the A register.

15). Type JMP $EA31 and RETURN. This is the ROM routine normally pointed to by the IRQ RAM vector. This allows normal housekeeping to be done.

Disassemble the code at $2000 and see if your disassembly matches the one given below.

ADD A BORDER COLOR CHANGE TO THE NORMAL INTERRUPT SEQUENCE

```
.,2000 48        PHA         PUSH THE A REGISTER ON THE STACK

.,2001 8A        TXA         TRANSFER THE X REGISTER TO THE
                             ACCUMULATOR

.,2002 48        PHA         PUSH IT TO THE STACK

.,2003 98        TYA         TRANSFER THE Y REGISTER TO A

.,2004 48        PHA         PUSH IT ON THE STACK

.,2005 AD 20 DO  LDA $D020   LOAD THE A REGISTER WITH THE BORDER
                             ADDRESS

.,2008 18        CLC         CLEAR THE CARRY FLAG

.,2009 69 01     ADC #$01    ADD WITH CARRY

.,200B 8D 20 DO  STA $D020   STORE THE RESULT AT BORDER COLOR

.,200E 68        PLA         PULL THE A REGISTER OFF THE STACK

.,200F A8        TAY         TRANSFER TO THE Y REGISTER

.,2010 68        PLA         PULL THE NEXT VALUE OFF THE STACK

.,2011 AA        TAX         TRANSFER IT TO THE X REGISTER

.,2012 68        PLA         PULL THE NEXT VALUE OFF THE STACK

.,2013 4C 31 EA  JMP $EA31   JMP TO THE IRQ ROM ROUTINE
```

We are now ready to activate our program with G 1000. If you typed everything in properly, you should now be experiencing an extremely irritating border color change 60 times a second. Everything else should be functioning normally. Let's see. Using the D command, type D 2000 2013. There's our program. Now go up to 2005. Go to the end of the line and change the 20 to a 21. This will stop the flashing. Now move to $200B. Go to the end of the line and again change the 20 to a 21. This should really drive you up a wall. You'll have to use RUN/STOP-RESTORE to stop it. Remember, the NMI (RESTORE key) cannot be masked (disabled) by our SEI, thus we can use it to warm start BASIC and return us to normal.

Not a very practical program, but through its simplicity we are able to gain an understanding of the IRQ function.

As you can see from our example, the IRQ routine is easily accessible to the programmer. While this is a joy for the programmer, it can pose many problems for the 'unprotector'. The programmer can easily store a 'self-destruct' sequence in his program to prevent access to the code through normal means.

In the program above, $2005-$200B contains the code to change the border color. You may also insert code to do whatever you wish during the interrupt cycle.

Our last interrupt is BRK (BREAK).

BRK (BREAK INTERRUPT) ROM $FFFE-F ($FF48)
RAM $0316-7 ($FE66)

Recall that when an IRQ or BRK occurs, the microprocessor will execute the ROM routine at $FF48. Through this routine, it will determine if BIT 4 of the STATUS REGISTER has been set. This is the BRK flag. If it is set, the last interrupt was caused by a BRK and not an IRQ. The following steps will then be taken:

1). The microprocessor will increment the program counter (PC) and store it on the stack (see SPECIAL NOTE below). The status register will be saved on the stack and the BRK FLAG set to indicate a BRK has occurred.

2). The normal IRQ interrupt sequence will be executed to determine if the interrupt was caused by an IRQ or a BRK. This is the ROM routine at $FF48.

3). The processor will execute the routine specified by the BRK vector at $0316-$0317. Under normal circumstances, this vector points it to $FE66, which is within the NMI routine. The net effect is to warm-start BASIC exactly as if RUN/STOP-RESTORE had been used.

The designers of the Commodore 64 chose to route the BRK routine through a vector in RAM, which may be accessed and changed by the user. This can be very useful. Many assemblers and monitors will program the BRK vector to return us to the monitor. This can make the BRK instruction invaluable in debugging machine language programs. We may insert a BRK instruction in our program at some point to verify that execution has reached this point. When the BRK is encountered, we will be returned to the monitor. The contents of all registers will be displayed automatically. We can examine these to determine if the program is executing properly, and then resume execution with a G command (but see SPECIAL NOTE below).

Let's do a little experimenting with the BRK RAM vector. With LOMON loaded and running, look at the code at $0316-$0317, with M 0316. You should see 3F 80 stored in this location. This is the address in LOMON we will jump to when a BRK occurs. Let's change that vector to point to the RESET routine at $FCE2 with :0316 E2 FC. Now put a BRK ($00) instruction at $1000 with :1000 00 or A1000 BRK. Now type G 1000. When the processor executes the BRK, it consults the vector at $0316-$0317. Since it finds the address of the RESET routine in this vector, a software RESET is performed and we see the normal start-up screen.

SPECIAL NOTE: Even though the BRK instruction is only one byte long, THE PC IS INCREMENTED BY TWO before being pushed on the stack in step 1 above. This only happens with BRK and not the other interrupts. Thus when returning from the interrupt via RTI, the processor will not resume execution at the next location directly after the BRK instruction, but rather one byte past that point. Most monitors compensate for this but it can cause maddening problems if you are using BRK and RTI directly in your own routines. TECHNICALLY this is not a bug since it is spelled out in the documentation (see the PROGRAMMER'S REFERENCE GUIDE p.238) but it certainly qualifies as a major quirk of the 6502/6510.

Understanding the three types of interrupts can open new avenues of programming techniques. Begin by expanding the programs illustrated here. The possibilities are unlimited. Give it a try!

COMPILERS

Most home computers sold today come equipped with a version of BASIC (Beginners All-purpose Symbolic Instruction Code). BASIC is a simple, English-like computer language created by Kemeny and Kurtz at Dartmouth College in 1965. They designed it originally to be easy to learn (and teach). It was popular right from the start and today it is the most common computer language in the world.

Its simplicity is probably the main reason for its success, but not the only one. Some credit is also due to the way it is usually designed to work (implemented). A language like BASIC can generally be implemented in one of two main ways. The most common form for BASIC is called an INTERPRETER. The alternative form is called a COMPILER. To understand the differences between the two, we need to take a look at the whole idea of a computer language.

The heart of a computer is the processor, which actually does all the work. The processor has been compared to someone of very limited intelligence who nonetheless has a perfect memory and works VERY fast. When dealing with the processor, you must stick to commands that it can understand and be careful what you tell it to do. The old saying is that it always does what you TELL it, but not necessarily what you WANT!

In the prehistoric days of computing (before 1950), the only way to change the operation of a computer was to hook and unhook wires inside it. By connecting the individual components, called logic gates, in carefully planned ways computer scientists could produce the output they desired.

They soon realized that the binary number system could be used to represent the connections to be made. Binary numbers are made up of only the digits 1 and 0. A 1 represents connection and a 0 means no connection. By converting the planned connections into binary, a set of numbers can be created to tell the computer how to switch its own wires, so to speak.

These numbers represent the first LOW-LEVEL computer language, called MACHINE LANGUAGE. Working with numbers instead of physical wiring simplified the process of programming considerably, but it was still far from convenient. It is safe to say that today no one programs in actual machine language other than a few instructions here and there. Instead we use the result of the next stage of development: ASSEMBLY LANGUAGE.

Assembly language is very similar to machine language. It consists of alphabetic codes called mnemonics which specify the same operations as machine language, but are easier for humans to remember. For example, the 6510 processor in the Commodore 64 has an instruction which in binary is 10101001. The assembler mnemonic for this is simply LDA.

Before a computer can understand a program written in assembly language, it is necessary to translate it. At first this was done by hand, but computer scientists soon found a way to have this tedious process done by the computer itself.

Thus the first ASSEMBLERS were born. An assembler takes a program in assembly language, called the source program, and translates it into a machine language program, called the object program. Interestingly, the first assemblers were written directly in machine code, but from then on the simple assemblers could be used to write better ones!

Hand-in-hand with assemblers go programs called disassemblers. As the name implies, these can take raw machine code and turn it into assembly language (mnemonics). They can't reverse the process completely without human intervention, though, because of the problem of telling data from program instructions (does this 10101001 really stand for LDA or simply hold a data value for use by some other part of the program?).

A single assembly language instruction usually translates into a single machine language instruction. Programs written in either form are usually long and always hard to read. Also, there are some common tasks that show up in almost every program, like arithmetic or printing. In the mid-50's computer scientists began to create the first HIGH-LEVEL languages to solve some of these problems.

A high-level language consists of much more powerful and flexible commands than assembly language. For example, the PRINT command or its equivalent is often the most complicated command in a language. A single PRINT statement can include letters, numbers, variables, TABs and other spacing controls like commas and semicolons. To actually perform the printing as well as preparing the printer or screen to receive the data could easily involve executing thousands of machine language instructions.

Thus a program in BASIC, say, is far shorter and easier to read, write and modify than the equivalent in assembly language. Once again, however, it must be converted to a form that the computer can understand. This is much more complicated and time-consuming than with assembly language. Not only do you have to expand the BASIC statements, you also have to keep track of the program's data. In assembly language we must specify where each piece of information is to be kept by giving the locations directly. In BASIC we use variables and the system has to keep track of them and reserve enough space.

Although high-level languages and ways of implementing them are still evolving, we do have a basic (no pun intended) choice of ways to proceed: a COMPILER or an INTERPRETER. Each has its own advantages and disadvantages, depending upon the environment in which it is to be used. An analogy may help to emphasize this.

Suppose you like Mexican food . Someone recommends a particular cookbook, so you buy it. When you get home you discover that it's in Spanish! Furthermore, it uses a lot of complicated cooking procedures. You know nothing of Spanish, or cooking either for that matter, but you ARE able to follow simple, specific directions, just like a computer.

Fortunately, you also have a Spanish-speaking friend who is an excellent cook. You will represent the processor, the cookbook is a BASIC program, and your friend is going to be a compiler or interpreter program.

You give him a copy of the book and ask his help. What would be the best way to proceed? Probably the most obvious way would be for your friend to translate the entire book. He could convert each recipe into English and explain the procedures used. If there are procedures that are used by more than one recipe, he might add the detailed explanations on to the end and refer you to them at the proper time. This corresponds to COMPILING the cookbook.

Now, this is going to be a lot of work! (hope you're not hungry). Furthermore, the new version will be a lot longer than the original because of the explanations. However, it does seem to be a complete solution. Even though it involves a lot of time and effort initially, this is more than repaid by the ease and speed with which you can now follow the English instructions. Also, you can now give out copies to other friends who only know English (better check the copyright law though!)

Also, if there are certain kinds of errors in the original book your friend can catch them right away. Maybe they have an obviously wrong amount of chili pepper (!!) or used a procedure your friend doesn't understand. You wouldn't catch small errors such as a little too much of one ingredient, though.

Historically, compiling is the approach that was used first in computing, and is still very common on large computers with lots of room to spare. There is another way we can use on our home computer, which has relatively little room available.

To continue the analogy, we aren't going to need all the recipes at the same time. In fact we might never get around to using some of them at all. Why should your friend go through so much work if it's not all necessary?

The alternative is to have your friend hanging on the phone while you are fixing a meal. As you need some information, he can look it up on his end and tell you directly. You can then execute his instructions immediately, so you won't have to remember them at all. You save a lot of paper and work at the start. This corresponds to INTERPRETING the cookbook.

This is going to be slower, though, and you will need to have your friend and the original book available throughout the whole process. Since you are a rank amateur, if a procedure is used in different recipes he will have to explain it each time. If there is an error, you won't find out until you get into the middle of the procedure, thus potentially wasting time and materials.

If you are going to use most of the recipes anyway over a period of time, or if you use a few very often, you'll end up taking up far more of your friend's time this way than the other. If you don't use them extensively, though, the situation is reversed.

Which way is best? By now you should have some idea why this has no simple answer. Each method has advantages and disadvantages relative to the particular situations in which they will be used. We might also be able to mix the two methods to some extent.

For instance, your friend could compile the most commonly prepared recipes and you would only have to call him for the others as needed. Or he could compile each recipe into a sort of shorthand and then interpret this for you as needed. These methods are indeed applicable to computing too.

Now let's look at a programming example to see the difference in the methods and their pros and cons. Examine the following BASIC program.

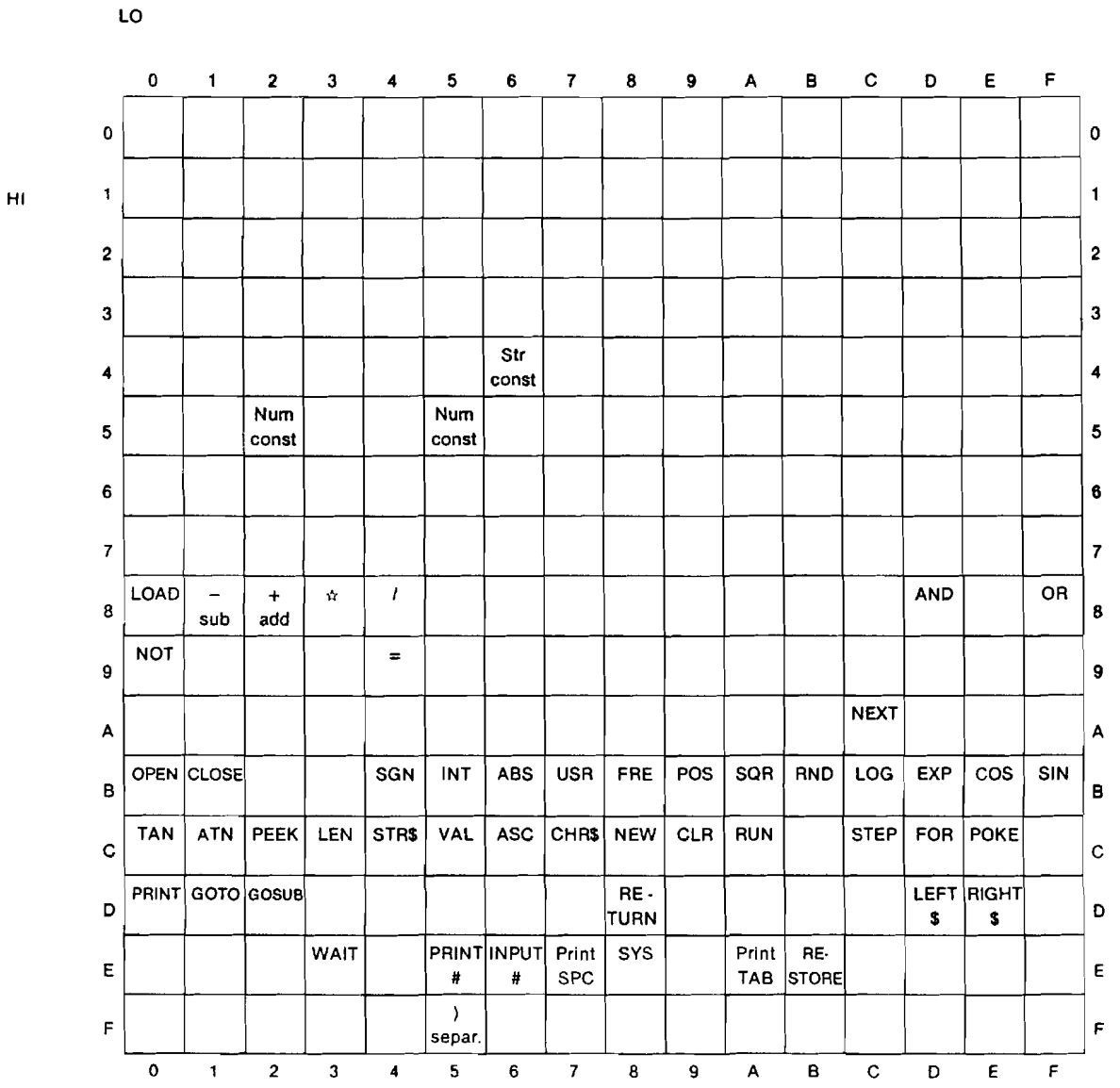
```
10 A=0
20 IF A<10 THEN 40
30 END
40 A=A+1
50 GOTO 20
```

A COMPILER processes the entire program in one chunk before any execution takes place. It checks for syntax errors (typos) as it goes. If there are any such errors, the program will not be executed. The compiler will proceed straight through the program in order by the line numbers, ignoring any GOTOs, GOSUBs, etc. that might be followed once the program is executed. Each statement will be examined exactly once.

In processing our sample program, a compiler would simply examine line 10, then 20, 30, 40 and 50 in that order. Each would be converted to machine language and stored. Since there are no syntax errors, when finished converting it would start the new version executing. The processor would take over from that point.

An INTERPRETER, on the other hand, processes the program one statement at a time. If it finds a syntax error in the statement just processed, it will stop. Otherwise, it will execute the statement. If that involves a branch, it will

HI \ LO	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			=	=			=	+ add	− sub	☆	/		AND	OR	− neg	NOT
1			FOR	NEXT no var	Next var	CLR		POKE		GOTO	GOSUB	ON GOTO	ON GOSUB	RE-TURN	RE-STORE	
2	SGN	INT	ABS	USR	FRE	POS	SQR	RND	LOG	EXP	COS	SIN	TAN	ATN	Peek	LEN
3	STR$	VAL	ASC	CHR$	LEFT $	RIGHT $	MID $	DEF FN	FN		SYS	PRINT FUNC	PRINT ;	PRINT)	PRINT	
4	SPC	TAB		PRINT #		PRINT # func	SET			STOP	READ STR	READ 'Num		WAIT	NEW	END
5			THEN GOTO	INPUT STR		INPUT NUM								LOAD	SAVE	VER-IFY
6	OPEN	CLOSE														
7																
8																
9																
A																
B																
C																
D																
E																
F																


LO
HI
0 1 2 3 4 5 6 7 8 9 A B C D E F
0 LOAD – sub + add ☆ / AND OR
1 NOT =
2 NEXT
3 OPEN CLOSE SGN INT ABS USR FRE POS SQR RND LOG EXP COS SIN
4 TAN ATN PEEK LEN STR$ VAL ASC CHR$ NEW CLR RUN STEP FOR POKE
5 PRINT GOTO GOSUB RE-TURN LEFT$ RIGHT$
6 WAIT PRINT# INPUT# Print SPC SYS Print TAB RE-STORE
7) separ.
Str const
Num const
Num const

follow the branch. The statements skipped over by the branch will not be processed at this time. If the branch takes it back to previously processed statements, they will have to be processed again.

In our example, an interpreter would process line 10 and execute it, then line 20 and execute it. Since line 20 branches to line 40, the interpreter will skip over line 30. Line 40 will be processed and executed, then line 50. Since line 50 goes back to 20, the interpreter will have to reprocess line 20 just as if it had never seen it before.

It will then branch to 40 again and have to reprocess it, etc. Eventually A will equal 10. This time the branch to 40 will not take place, and it will process line 30 next. This is the only time line 30 will be processed. Executing it will then end the program.

Here we see the main difference. The COMPILER will have to process every line, but only once (in an idealized compiler). The INTERPRETER will process only those lines it needs to execute, but it may have to reprocess lines repeatedly.

In comparing the speed of the two methods, it is important to remember that the compiler's use of time is divided into two separate parts: First all the compiling, then the executing. The compiling stage itself generally takes a lot of time but reduces the execution time to a minimum. Also, compiling only needs to be done once. The compiled program, including copies of it, can then be executed any number of times, at high speed.

By contrast, an interpreter goes from processing a statement to executing it, back to processing, etc. It may well finish before the compiler is even done compiling. Once done, however, the compiled program will be able to execute many times faster than the interpreter. If the program is to be executed more than a few times, compiling will save time.

What about memory requirements? Since the interpreter operates one line at a time, it only needs a little extra space. The compiler, however, has to store the entire compiled program, which is invariably much larger than the original. This can be a significant factor for home computers.

As far as syntax errors are concerned, the compiler will find all of them in the compile stage, whereas the interpreter will only find those it actually encounters during execution. This can be very important in a large program. If an error exists, and you test the program with an interpreter, you might not happen to test that particular section. You could put the program into normal use, and then a month or a year later it suddenly fails because that one situation finally comes up. Depending on what the program is used for, this could have serious consequences.

This suggests that a program could be developed using an interpreter for convenience. When it is finished, it could be compiled and used in that form. There may be times when you want the program to stay in BASIC though, for example if you want other people to be able to read it. In this case you can still use the compiler to check the program for syntax errors. Most compilers, however, are not able to handle 100% of the BASIC interpreter commands, so it may signal some good statements as errors.

Is there some happy medium between the two methods? Yes, in fact true compilers as outlined above are rare on home computers. Instead we have a hybrid type called a p-code compiler. Essentially, a p-code compiler will process the entire program like a true compiler would, but instead of producing a machine language object program, it produces what is called a p-code program.

P-code stands for pseudo-code (it's also sometimes called speedcode). It is a special shorthand version of the BASIC program. Since it cannot be executed directly, it must be interpreted by a special program called the runtime package or runtime library (RTL). This interpreter operates much the same as the BASIC interpreter. This may sound more like the worst of both methods, since we have a compile stage and then have to interpret too. However, it is a significant improvement for the following reasons.

First, the p-code object program is somewhat smaller than the BASIC source program. This is because the compiler removes all remarks, line numbers, statement separators (:) and spaces (except those in quotes, of course). In GOTOs and GOSUBs it uses the actual memory location instead of line numbers. Also, it keeps a list of all variables used, so that in place of variable names it can use a variable number. If you count the runtime package, though, a single small program will become larger overall. If you have a set of interrelated programs, however, you need only include the runtime package with the first program (usually a menu) and from then on just load in the p-code programs. This can save both disk space and load time.

Second, and most important, the p-code program can be interpreted much faster than straight BASIC. In BASIC, when a GOTO or GOSUB is performed, the interpreter must search through your program looking for the desired line. On the C64, there are built-in links to the next statement to help speed this up, but in a large program it will still take up considerable time. With p-code, it just jumps directly to the correct location.

The regular BASIC interpreter stores its variables in tables based on the type and the order they were encountered. When it needs to find one, it must search through the proper table trying to match the variable name. The P-code compiler represents its variables by their position in the table, so

again the runtime package can go directly to the correct location. This is especially important in programs with large arrays.

In BASIC programs, numbers are stored exactly as they appear, that is, as ASCII digits. Before one can be used it must be converted to a binary form suitable for calculation. In fact, this must be done each time the number is encountered.If the number is encountered repeatedly,as in a loop, this can take a significant amount of time. In p-code, all this conversion is done only once, at compile time. This speeds up numerical calculations and IF statements as well as loops.

Finally, a single BASIC statement can be split into several pieces when compiled, and the pieces rearranged to improve the speed of execution. Rather than process the statement sequentially, p-code uses a technique called reverse Polish notation (RPN). This is not a joke; it's named for a Polish mathematician whose name is hard to spell. This involves the use of a stack much the same as the stack in the 6510 processor. Without going into detail, this simplifies the logic and reduces the time necessary to match up operations and the data they use. As a result, most statements are stored in reverse, with the data coming first and the p-code command last.

A further advantage of some p-code compilers is that they actually add features to BASIC or allow it to use regular features in new ways. The most popular compilers for the C64 are Petspeed from Oxford Computer Systems and Blitz! from Skyles Electric Works. Each has its own advantages and disadvantages. The new features they offer are useful, but they're not major improvements. You may not want to use them since they will cause syntax errors when testing your BASIC source program.

Also, neither of these compilers can handle 100% of Commodore BASIC. The limitations are different for each one but they are relatively minor. However, you do have to take them into account when you write your BASIC source program. Overall, Blitz! seems to compile faster and have more features and fewer limitations. Abacus Software has just come out with a compiler offering both p-code and true compiler options, but we have not had a chance to evaluate it.

Using a compiler is usually very simple. You just run it and it asks for the filename of your program. It then analyzes your program in several passes, while building the p-code version. It will create some temporary files to help it keep track of things. Finally it saves out the p-code version along with the runtime package. The whole process takes anywhere from a few minutes to half an hour, depending on the compiler and the length of your source code.

When pirates look at the compiled code, they see a large, complex program. They may try to trace the program but since it goes about its job in a roundabout way, this is going to be a long process. Chances are they will give up in frustration long before they get to the protection part of your code.

Let's look at an example of a compiled program. The following table shows a comparison of a BASIC program and its p-code equivalent. This particular program was compiled by Blitz!. Petspeed produces superficially similar code that differs substantially in the details.

BASIC PROGRAM	BLITZ! P-CODE (HEX)
(no equivalent)	15
10 OPEN 15,8,15,"I"	E9 49 BF B8 BF 60 04
20 OPEN 2,8,2,"#"	E9 23 B2 B8 B2 60 04
30 PRINT#15, "U1:2 0 35 01"	BF 42 E7 0D 55 31 3A 20 32 20 30 20 33 35 20 30 31 43
40 GET#15, A$	BF 48 80 46
50 IF A$="2" THEN 70	80 E9 32 02 52 1F D5
60 SYS 64738	A8 90 7C E2 00 00 18 3A
70 CLOSE 2	B2 61
80 CLOSE 15	BF 61
90 LOAD "MAIN",8,1	B1 B8 EC 4D 41 49 4E 5D 03
100 REM THIS IS A SAMPLE REMARK	(no equivalent)
(no equivalent)	4F

Both the BASIC and Blitz! versions are included on the program disk if you want to examine them further. I've included the Petspeed version as well. The BASIC version is called 'UNCOMPILED' and the other two are 'COMPILED.B' (Blitz!) and 'COMPILED.P' (Petspeed).

When you examine the Blitz! version, note that the runtime package occupies the first 6K of the BASIC area and the p-code version comes at the end. Actually, the p-code is preceded by six two-byte pointers at $1F93-$1F9E. These are similar to the regular BASIC pointers at $2D-$38. They indicate the locations of the variable tables, DATA statement table, etc. Of particular interest is the pointer at $1F9D-E, which points to the start of the p-code program. In this case it is at $1FA1. The Petspeed version is organized in a similar way except the

pointers come at the beginning of the runtime package and the package itself is 8k long. The Petspeed p-code starts at $281B.

The example I've used might be part of a protection scheme. As you can see, it checks for an error at track 35, sector 1 (any one will do). If it doesn't find it, it resets the computer. Otherwise it loads the main program file. This may not be a realistic scheme but it will serve as an example.

At the end of this chapter I have included a table giving a partial list of the Blitz! p-codes and their BASIC equivalents. A Petspeed table is also included. (WARNING: These were derived by experimenting and may not be totally accurate. This applies to the following discussion too.) Let's use the Blitz! table to examine the above p-code program.

First of all, note again that there are no line numbers at the beginning of p-code lines. The first line of the p-code contains a $15 byte, which does not correspond to anything in the BASIC program. This is actually a CLR command which is inserted automatically at the beginning of the program.

Line 10 (p-code) starts right off with the codes for 'I'. This is the string at the end of the OPEN statement (remember most statements are stored in reverse). In this case we have what is called a literal string or string constant, that is, a quoted string as opposed to a string variable. If less than 8 characters long, string constants are preceded by a extra byte, which is $E8 plus the length of the string ($E8+1 = $E9). No quotes are used. Next comes the 'I' itself in ASCII ($49).

Numbers less than 16 are stored as a $B0 plus the hex equivalent. Thus $BF stands for 15 ($0F), $B8 for 8 and then another $BF. Again, these codes are in reverse of their normal order. The $60 code stands for OPEN. It is followed by an $04 which indicates that all possible parameters of the OPEN statement are specified (file no., device no., secondary address and string). With fewer parameters, this second byte would be lower.

The next line is similar ($23 is ASCII for '#'). The following line starts with $BF 42, which stands for output to file 15 ($BF). The next TWO bytes stand for the length of the string to be printed. In this case, the string is longer than 8 characters so it uses a different format than our previous examples. The $E7 indicates that the next byte is the actual string length ($0D = 13). Following this is the string itself. Finally, $43 stands for PRINT# for strings.

The next line has $BF 48 for input from file 15. The $80 stands for the variable A$. Non-array variables are represented by an $80 plus the variable number. The number is based on the order they were first encountered in the program, starting at number zero. The $46 represents GET# for strings.

Line 50 checks for the error. First we have $80 for A$, then $E9 32 for the string '2'. The $02 byte stands for =. The $52 is for THEN when it is followed by a line number (as opposed to being followed by a statement such as PRINT). The next two bytes, $1FD5, are actually the address to go to (the location in memory of line 70). Note they are reversed from their normal lo-byte/hi-byte (backwards) order!

The following line contains a $3A for SYS preceded by a binary form of 64738 ($FCE2). Line 70 has a $B2 (=2) followed by $61 for CLOSE. Line 80 is similar.

Line 90 is similar to lines 10 and 20. First we have $B1 for secondary address 1, then $B8 for device 8. Next is the length of the filename ($E8+4 = $EC) followed by the four-byte filename ($4D 41 49 4E = ASCII for 'MAIN'). At the end is $5D, which stands for LOAD, and $03 which indicates a LOAD with three parameters specified.

Notice that there is no p-code corresponding to line 100, since the compiler removes REM statements completely. Finally, in the p-code program there is a $4F. This stands for END, which is added on automatically.

Whew!. This may seem fairly simple once it is explained, but figuring it out from scratch is another thing. And this was just a short program! It doesn't have any of the more obscure situations, such as FOR/NEXT.

It also doesn't have any LET statements, which should be mentioned. LET is not represented by a separate code; instead the number of the variable to receive the value is added to $C0. For example, LET A=1 could appear as $B1 C0. The $B1 is for the number 1. If A is the first variable in the program it would be number 0, hence $C0.

In summary, let me say that compiling remains a very viable program protection option for BASIC programs. It is unmatched in convenience and offers a speed advantage too. It is not totally unreadable, as we have seen, but requires a lot more work than machine language to understand. Of course, this is because of a total lack of documentation. The tables of p-code equivalents I've given required several days of solid work to produce, as incomplete and possibly inaccurate as they are. Applying them to a compiled program is not simple, either.

It is certainly possible in theory to create a 'decompiler' program which would be able to read p-code and reproduce the original BASIC source program (aside from variable and FN names). There have been rumors about them for years, but to date none have materialized. Here's your chance to make a mark. If you come up with a more complete p-code table, or even a decompiler, send it in. If it meets our standards we'll pass it along to our readers through the newsletter or future volumes of the Program Protection Manual. Good luck!

The Commodore 64 contains a MOS 6510 microprocessor. It is a slightly revised version of the 6502 processor, which is used in Commodore's VIC-20 computer and 1540/41 disk drives, as well as the Apple and Atari computers.

All microprocessors are similar in that they understand only a limited set of commands. These commands are organized into groups of related commands which are similar in function but differ in where the actual data comes from or goes to. Each group of related commands is called an INSTRUCTION, and the location of the data is determined by what is called the addressing mode.

For example, take the LDA instruction. This is the most common instruction used on the 6510. LDA actually is a mnemonic (memory aid) which stands for LoaD the Accumulator, also called the A register. The accumulator is like a variable in BASIC in that it can hold a value, but in this case the value is limited to one byte, which means a range of 0 to 255 ($00 to $FF).

Now, the accumulator can load a value from a variety of sources. Therefore, the LDA instruction has several different forms. Each form is denoted by a different one-byte operation code, or OPCODE for short. The particular opcode that is used tells the processor WHAT to do. It also determines the addressing mode, which tells the processor HOW to get its data. Each form usually requires one or two additional bytes of information which actually specify WHERE the data is contained (or contain the data itself). This other info is generally called the OPERAND, and it is usually an address (location in memory)

Let's look at some examples. One form of LDA has the opcode $A9 (the '$' indicates a number specified in hexadecimal or hex). This form of LDA specifies what is called immediate addressing, meaning that the next byte actually contains the value to be loaded into A, say a $05. Here's how this would look using a monitor such as HIMON:

HEX CODE		MNEMONIC		ENGLISH
OPCODE	OPERAND	OPCODE	OPERAND	EXPLANATION
A9	05	LDA	#$05	Load A with value $05

The '#' in the operand's mnemonic is a sign to the assembler to use immediate addressing, so that it can substitute the correct opcode form for LDA. Thus the microprocessor knows the addressing mode by the actual opcode, whereas assemblers (and us humans) find it easier to use a single mnemonic (LDA) for the instruction and indicate the address mode separately (#).

Actually, the operand is NOT usually the actual value to be loaded, but rather it is the ADDRESS or location of the value. Let's look at some other forms of LDA:

HEX CODE OPCODE OPERAND	MNEMONIC OPCODE OPERAND	EXPLANATION LoaD A from :
AD 34 12	LDA $1234	Location $1234
BD 00 20	LDA $2000,X	Location $2000+X
A5 05	LDA $05	Location $05
B5 09	LDA $09,X	Location $09+X

Note the different hex opcodes for each form. The first form uses what is called absolute addressing. The data to be loaded into the accumulator is the one-byte value found at location $1234. In the hex form, the address is always stored with the low-order (least significant, $34) byte FIRST and the high-order (most significant, $12) byte second.

The second form is called absolute, indexed by X. The X register is similar to the accumulator, and in this case is used to hold a one-byte INDEX, or offset. The actual address of the data is calculated by adding the current value of X (say $07) to the operand ($2000, often called the base address). Thus the accumulator would be loaded from location $2007.

The other forms are merely zero-page forms of the first two. This means the high-order byte of the address is assumed to be $00, so that only a single operand byte is needed, in order to specify the low-order byte. Thus the address of the data in the third form would be $0005, and in the fourth it would be $0009 plus the contents of X. The reason there are separate zero-page forms for LDA is to save memory space and execution time.

There are several other forms of LDA, but these should illustrate the point that a single INSTRUCTION can be represented by many different OPCODES. The difference is in the addressing mode: how the operand bytes will be interpreted (value or location). If you consult a standard reference book on the 6510 processor, you will see a total of 56 different instructions listed (this applies to the 6502 as well). With the different addressing modes, there are a total of 152 different 'official' (documented) opcodes.

Now, each combination of instruction and address mode is represented by a separate one-byte opcode. Since a byte can have 256 different values, this leaves 256 - 152 = 104 'unused' opcodes. Actually, most or all of these opcodes are useable, so we prefer to call them 'undocumented' opcodes. Other sources may call them 'unimplemented', 'undefined' or 'illegal' opcodes.

Whatever you call them, these undocumented opcodes can be very useful in program protection. No normal monitor or disassembler will recognize them (they usually appear as ???). Since there is very little information available about them, a pirate will have a tough time following your code. You can use them to shorten a section of code, since they often combine several regular instructions into a single one. You can also lengthen a piece of code by 'burying' a couple normal opcodes in a stretch of undocumented ones.

Let's take a look at some of these new opcodes. Table OP-1 shows the regular 6510 opcode set. Each opcode (byte) is represented by two hex digits, called nybbles. The row headings represent the high-order (most significant or left) nybble and the columns represent the low-order (least significant or right) nybble. The opcode $A9, for instance, is found where the tenth row down ($A = 10) and ninth column across meet. Here you see LDA #.

Notice all the blank spaces in the table. These correspond to undocumented opcodes. All opcodes ending in 3, 7, B or F are open, as are most ending in 2, 4, A or C. This certainly leaves a lot of possibilities! Some of these opcodes are listed in table OP-2. The capitalized mnemonics listed in the left margin are mostly from published material by Joel C. Shepherd. Following the mnemonic is a brief description of its function, and below that is a list of the actual hex opcodes and their corresponding addressing modes.

Take ANDX for example. This stands for store A AND X registers. First, the values in the A and X registers are AND'ed together (a standard logic operation). Neither is changed; the result is placed in the memory location specified by the operand. Let's use the absolute addressing form, opcode $8F, for our example. This could be done by regular 'documented' opcodes but it would take more memory space to store and more processor time to execute:

Documented opcodes		Explanation	Undocumented opcodes	
8E 00 20	STX $2000	Store X value	8F 00 20	STAX $2000
2D 00 20	AND $2000	A = A AND value		
8D 00 20	STA $2000	Store A in $2000		

The difference is even greater if you want the regular code to duplicate the undocumented code exactly. The regular code changes the accumulator, which the undocumented code doesn't.

Here's another way to use them. The two pieces of code below perform the same function (if you ignore things like the X register). In this case, all they do is change the screen and border colors to black.

```
Documented opcodes                          Undocumented opcodes
AD 00 08  LDA $0800    Get a 0              AF 00 08  LDAX $0800
8D 20 D0  STA $D020    Border               8F 20 D0  STAX $D020
8D 21 D0  STA $D021    Screen               8F 21 D0  STAX $D021
00        BRK          Go to monitor        00        BRK
```

The easiest way to try this out is to type in the regular code from a monitor. Then go back and substitute the undocumented opcode bytes in place of the others (they are the only things different between the two routines). Execute the code by using a 'G' command to the beginning of it. It works!

If a pirate tries to disassemble the undocumented codes using a monitor, here is what he/she will see:

```
AF          ???
00          BRK
08          PHP
8F          ???
20 D0 8F    JSR $8FD0
21 D0       AND ($D0,X)
00          BRK
```

Quite a difference!

Upon seeing this most pirates would assume they made a mistake in tracing your program flow, and go back over the code up to that point. Even if they knew about the undocumented codes, many will not want to spend the time and effort to decode a sizeable piece of it. Your main weapons against pirates here are to confuse them and make them work for their ill-gotten gains.

Note that some of the undocumented opcodes are not covered in the second table. These you may want to investigate further yourself. There are a couple of approaches that may be helpful.

One way is to simply try executing the unknown operation. Start by storing the opcode in memory followed by one or two bytes (or possibly none) for the operand. After this you should place a documented instruction to return control to you after execution (like BRK). To test, load the registers and the location you think will be used as the operand with some sample values. If you are using a monitor like HIMON, it is easy to load values into the registers. Use the 'R' command to display the registers, then type your values over the displayed ones and hit RETURN. Finally, execute the instruction. By examining the registers and memory location afterward, you may be able to work out what happened.

TABLE OP-1A DOCUMENTED

LO

HI	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	HI
0	BRK	ORA (Z, X)				ORA Z	ASL Z		PHP	ORA #	ASL A			ORA M	ASL M		0
1	BPL	ORA (Z), Y			BIT Z	ORA Z, X	ASL Z, X		CLC	ORA M, Y				ORA M, X	ASL M, X		1
2	JSR	AND (Z, X)				AND Z	ROL Z		PLP	AND #	ROL A		BIT M	AND M	ROL M		2
3	BMI	AND (Z), Y				AND Z, X	ROL Z, X		SEC	AND M, Y				AND M, X	ROL M, X		3
4	RTI	EOR (Z, X)				EOR Z	LSR Z		PHA	EOR #	LSR A		JMP M	EOR M	LSR M		4
5	BVC	EOR (Z), Y				EOR Z, X	LSR Z, X		CLI	EOR M, Y				EOR M, X	LSR M, X		5
6	RTS	ADC (Z, X)				ADC Z	ROR Z		PLA	ADC #	ROR A		JMP (M)	ADC M	ROR M		6
7	BVS	ADC (Z), Y				ADC Z, X	ROR Z, X		SEI	ADC M, Y				ADC M, X	ROR M, X		7
8		STA (Z, X)			STY Z	STA Z	STX Z		DEY		TXA		STY M	STA M	STX M		8
9	BCC	STA (Z), Y			STY Z, X	STA Z, X	STX Z, Y		TYA	STA M, Y	TXS			STA M, X			9
A	LDY #	LDA (Z, X)	LDX #		LDY Z	LDA Z	LDX Z		TAY	LDA #	TAX		LDY M	LDA M	LDX M		A
B	BCS	LDA (Z), Y			LDY Z, X	LDA Z, X	LDX Z, Y		CLV	LDA M, Y	TSX		LDY M, X	LDA M, X	LDX M, Y		B
C	CPY #	CMP (Z, X)			CPY Z	CMP Z	DEC Z		INY	CMP #	DEX		CPY M	CMP M	DEC M		C
D	BNE	CMP (Z), Y				CMP Z, X	DEC Z, X		CLD	CMP M, Y				CMP M, X	DEC M, X		D
E	CPX #	SBC (Z, X)			CPX Z	SBC Z	INC Z		INX	SBC #	NOP		CPX M	SBC M	INC M		E
F	BEQ	SBC (Z), Y				SBC Z, X	INC Z, X		SED	SBC M, Y				SBC M, X	INC M, X		F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

TABLE OP-1B · DOCUMENTED AND UNDOCUMENTED

HI \ LO	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	BRK	ORA (Z, X)				ORA Z	ASL Z	SLOR Z	PHP	ORA #	ASL A			ORA M	ASL M	SLOR M
1	BPL	ORA (Z), Y			BIT Z	ORA Z, X	ASL Z, X	SLOR Z, X	CLC	ORA M, Y				ORA M, X	ASL M, X	SLOR M, X
2	JSR	AND (Z, X)				AND Z	ROL Z	RLAN M	PLP	AND #	ROL A		BIT M	AND M	ROL M	RLAN M
3	BMI	AND (Z), Y				AND Z, X	ROL Z, X	RLAN Z, X	SEC	AND M, Y				AND M, X	ROL M, X	RLAN M, X
4	RTI	EOR (Z, X)				EOR Z	LSR Z	SREO Z	PHA	EOR #	LSR A		JMP M	EOR M	LSR M	SREO M
5	BVC	EOR (Z), Y				EOR Z, X	LSR Z, X	SREO Z, X	CLI	EOR M, Y				EOR M, X	LSR M, X	SREO M,X
6	RTS	ADC (Z, X)				ADC Z	ROR Z	RRAD Z	PLA	ADC #	ROR A		JMP (M)	ADC M	ROR M	RRAD M
7	BVS	ADC (Z), Y				ADC Z, X	ROR Z, X	RRAD Z, X	SEI	ADC M, Y				ADC M, X	ROR M, X	RRAD M, X
8		STA (Z, X)			STY Z	STA Z	STX Z	ANDX Z	DEY		TXA	ANAX #	STY M	STA M	STX M	ANDX M
9	BCC	STA (Z), Y			STY Z, X	STA Z, X	STX Z, Y	ANDX Z, Y	TYA	STA M, Y	TXS			STA M, X	TSTA M	TSTX M
A	LDY #	LDA (Z, X)	LDX #		LDY Z	LDA Z	LDX Z	LDAX Z	TAY	LDA #	TAX	LDAX #	LDY M	LDA M	LDX M	LDAX M
B	BCS	LDA (Z), Y			LDY Z, X	LDA Z, X	LDX Z, Y	LDAX Z, Y	CLV	LDA M, Y	TSX	ANSP M, Y	LDY M, X	LDA M, X	LDX M, Y	LDAX M, Y
C	CPY #	CMP (Z, X)			CPY Z	CMP Z	DEC Z	DCMP Z	INY	CMP #	DEX	SUBX #	CPY M	CMP M	DEC M	DCMP M
D	BNE	CMP (Z), Y				CMP Z, X	DEC Z, X	DCMP Z, X	CLD	CMP M, Y				CMP M, X	DEC M, X	DCMP M, X
E	CPX #	SBC (Z, X)			CPX Z	SBC Z	INC Z	ISBC M	INX	SBC #	NOP		CPX M	SBC M	INC M	ISBC M
F	BEQ	SBC (Z), Y				SBC Z, X	INC Z, X	ISBC M, X	SED	SBC M, Y				SBC M, X	INC M, X	ISBC M, X

As an example, let's test opcode $87. First, load in one of the monitors on the program disk. Using the ':' memory modify command, enter the test bytes at location $1000. Use the Go command 'G' to start execution.

```
:1000 87 FB 00 00

G 1000
```

Suppose $87 requires only one byte for its operand (data) address. The next byte after the $87 is $FB, so this is the address it will use ($FB is an unused location in zero-page). Since the following byte is $00, it will then perform a BRK, which returns us to the monitor.

Now suppose, on the other hand, $87 requires two operand bytes. In this case the bytes it will use are $FB 00, forming the address $00FB (remember the bytes are in reverse order). This actually specifies the same zero-page location as before. The next byte after $FB 00 is another $00, so the next instruction executed would be BRK again.

So either way, we'll be sure to get back to the monitor. How do we tell which of the two possibilities above actually happpened? When you re-enter the monitor, it will display the contents of all the registers, including the program counter (PC). The PC tells you the address of the last instruction executed (which $00 byte caused the BRK). From this we can usually determine how many bytes the instruction required.

In the example above, we get a PC value of $1002. Thus it was the $00 at $1002 that caused the BRK. This means that $87 requires only one operand byte (the $FB). If you check table OP-2, you'll see that $87 is ANDX with zero-page addressing, which simply means it uses only one operand byte. This confirms our experimental result.

In short, you can find the total number of bytes an instruction used (including the opcode itself) by simply subtracting the location of the instruction ($1000) from the PC value after the instruction executes ($1002). Some monitors, such as HESMON, return a PC value of $1003 in the test above. This simply means you have to subtract 1 from your answer. There's no problem as long as you know which way your monitor works.

You'll find that some of these codes will lock-up your computer. This is not dangerous, just frustrating. Unlike all those science fiction movies you've seen, you can't damage your computer by typing in the wrong command (the disk drive is another matter - see the editorial on bad blocks). This lock-up can happen if the opcode you're testing performs a branch or jump.

On the 6510 processor, a branch instruction can go up to 127 bytes forward or 128 bytes backwards only. As long as you surround your instruction with this many $00 (BRK) bytes, a branch is sure to hit one and thus return you to the monitor. A jump (JMP) or jump subroutine (JSR), however, can end up anywhere in memory. To prevent lock-up, fill as much of memory as possible with $00, and watch out what you use for your test operand. Still, you may have to use your reset switch to recover in some cases.

As you might guess, the trial and error method can be very time-consuming. Another approach is to examine table OP-1 for patterns among the regular instructions and then apply the patterns to the undocumented codes.

For instance, the regular opcode $05 is ORA with zero-page address (single operand byte), and regular opcode $06 is ASL with zero-page. From this you might guess that the undocumented code $07 also uses zero-page addressing. Likewise, since $15 and $16 are the X-indexed versions of $05 and $06, respectively, you might conclude that $17 could be an X-indexed version of $07. A glance at table OP-2 confirms these guesses.

You might also notice that the regular opcode $0E is the absolute (two-byte operand) form of $06. Since $0E = $06 + $08, you might guess that the undocumented opcode $0F is the absolute form of $07 ($0F = $07 + $08). Also, $1E is the X-indexed form of $0E, so $1F could be the X-indexed form of $0F. These observations also turn out to be true.

In fact, the undocumented opcode $07 is actually a combination of the regular opcodes $05 and $06. We called it SLOR because it first performs an ASL and then an ORA. Most of the codes in columns 7 and F work this way, subject to the following general rules:

1). If possible, the two functions are executed simultaneously. Otherwise, the one with the higher opcode is executed first. This explains why $07 (SLOR) first performs a $06 (ASL) and then an $05 (ORA).

2). If two values are to be stored into the same location (not normally possible) the values will be AND'ed together first. For example, this accounts for the operation of ANDX, which first AND's the A and X registers together and then stores the result in memory.

3). If the two functions it performs use different indexes, i.e. one is indexed by X and the other by Y, then the new opcode will use Y-indexing. For example, opcode $B5 is LDA zero-page with X-indexing and $B6 is LDX zero-page with Y-indexing. Undocumented opcode $B7 loads both A and X from zero-page, with Y-indexing.

The patterns do not always hold true (see code $9F for example), but they can give you some idea of what to look for when experimenting. Of course, you will have to use the trial and error method outlined above to confirm your guesses.

A third approach is to study the internal design of the microprocessor to predict how it will handle the undocumented opcodes. It must use a fairly simple system to decide which functions to perform, based on looking at the bits of the opcode. This is called instruction decoding and is what you are trying to estimate from analyzing the patterns in the table.

Unfortunately, the required information is very hard to obtain. It may well be proprietary (trade secret) and thus not available to the public. It would, however, settle the question once and for all - unless they redesign the processor!

This is not as unlikely as it sounds, and brings up some of the disadvantage of these opcodes. Since there is so little information available, you should experiment with an opcode to confirm its function, even for those we've included in our tables. To enter undocumented opcodes into a program you will have to look up the opcode and store the hex value into memory with a monitor one instruction at a time. This can be very tedious. An alternative would be to modify a standard assembler or monitor to handle them, perhaps one written in BASIC.

Finally, and most important, 'Commodore Semiconductor Group cannot assume liability for the use of undefined opcodes'. This means that if the manufacturer redesigns the processor (to correct bugs or reduce power use or size) there is no guarantee that the unofficial opcodes will still function the same.

In fact, the 6502 processor upon which the 6510 is based has had several revisions. The unofficial codes in this chapter will work on some 6502's and not others, especially ones from different manufacturers. We Commodore 64 owners have been lucky so far; there are no reported differences between the 6510 processors in use. This is probably due to the fact that they are manufactured by MOS Technologies, which is owned by Commodore.

Undocumented opcodes have not been used in many commercial programs so far, although there have been some. In conclusion, they offer excellent, if largely untapped, program protection potential.

Table OP-2: UNDOCUMENTED OPCODES

ANDX - Takes the accumulator and X-register, AND's them together and stores the result in memory. The accumulator and X-register are not changed.

MODES: 87 Zero page
97 Zero page, indexed by the Y-register
8F Absolute

ANXM - Takes the accumulator, X-register and operand byte, AND's them together and stores the result in the accumulator. The X-register is not changed.

MODES: 8B Immediate

AXSP - Takes the contents of the memory location, indexed by the Y-register, AND's it with the stack pointer, and stores the result in the stack pointer, accumulator and X-register. The memory location is not changed.

MODES: BB Absolute

DCMP - Decrements the memory location, subtracts the new contents of the memory location from the accumulator and puts the result in the accumulator.

MODES: C7 Zero page
D7 Zero page, indexed by the X-register
CF Absolute
DF Absolute, indexed by the X-register

ISBC - Increments the memory location, subtracts the new contents of the memory location and the carry flag from the accumulator, and places the result in the accumulator and carry flag.

MODES: E7 Zero page
F7 Zero page, indexed by the X-register
EF Absolute
FF Absolute, indexed by the X-register

LDAX - Loads the accumulator and X-register from the memory location. The memory location is not changed.

MODES: A7 Zero page
B7 Zero page, indexed by the Y-register
AF Absolute
BF Absolute, indexed by the Y-register
AB Immediate

RLAN - Rotates the bits of the memory location left, AND's the new contents of the memory location with the accumulator and places the result in the accumulator.

MODES: 27 Zero page
37 Zero page, indexed by the X-register
2F Absolute
3F Absolute, indexed by the X-register

RRAD - Rotates the bits of the memory location right, adds the new contents of the memory location and carry flag to the accumulator, and places the result in the accumulator and carry flag.

MODES: 67 Zero page
77 Zero page, indexed by the X-register
6F Absolute
7F Absolute, indexed by the X-register

SLOR - Shifts the memory location left, OR's the new contents of the memory location with the accumulator, and places the result in the accumulator.

MODES: 07 Zero page
17 Zero page, indexed by the X-register
0F Absolute
1F Absolute, indexed by the X-register

SREO - Shifts the memory location right, EXCLUSIVE-OR's the new contents of the memory location with the accumulator, and places the result in the accumulator.

MODES: 47 Zero page
57 Zero page, indexed by the X-register
4F Absolute
5F Absolute, indexed by the X-register

SUBX - Subtracts the value given from the X-register and places the result back in the X-register.

MODES: CB Immediate

TSTA - AND's the accumulator with the value $04 and places the result in the memory location. The accumulator is not changed.

MODES: 9F Absolute

TSTX - AND's the X-register with the value $04 and places the result in the memory location. The X-register is not changed.

MODES: 9E Absolute

Note: Addressing mode definitions

Absolute - A two-byte ADDRESS given in standard lo-byte, hi-byte (reverse) order. Address $1234 would be specified as $34 12.

Zero page - A one-byte ADDRESS, with the hi-byte assumed to be $00. Address $0012 would be specified as $12.

Immediate - A one-byte VALUE specified directly; not a memory location.

ENCRYPTION is a hot topic today, not only in program protection but also in a lot of other applications. Everyone from the federal government on down through businesses (including bookie joints!) to game programmers seems to be interested in new ways to scramble information so it can't be read - until the right time. As much time as is spent on creating new encryption methods, certainly a far greater amount is spent trying to break these 'codes'.

What is encryption? In answering that question most people would use the word 'code' somewhere along the line. The concept of encryption does include the idea of codes but can go far beyond them into some very advanced mathematics. Fortunately for all of us, these methods are beyond the scope of this manual, as well as most program protection schemes!

Strictly speaking, a code is a direct substitution of one unit of information for another according to a set rule. These units of information can be letters, words, digits, whole numbers, even sounds or lights. Often, one type of information is substituted for another. The rule used for encoding can be very simple or very complex. For every encoding rule there is also a corresponding decoding rule which reverses the original substitution.

Encrypting includes straight encoding but is a more general process. Encryption includes other methods too, including some whereby the information is expanded as it is encrypted, either by a mathematical method or simply by embedding it within a larger body of information. Of course, all encryption methods have corresponding decryption methods. Although this chapter will involve techniques that are actually encoding methods, I'll use the general term encryption.

The history of encryption is a long story whose beginning is lost. Archaeologists have traced it back at least as far as ancient Babylonia. Clay tokens were used by merchants to label sealed jars of merchandise for shipment. Robbers would not be able to tell which jars contained valuables, yet the receivers could inventory them without opening perishable goods.

Eventually the tokens' meanings became well known, and they began to be used in everyday communication. Rather than make the tokens themselves, people used them to make impressions in clay tablets. This was the beginning of written language. Not only is this the earliest known example of encryption, but also the earliest case of breaking such a scheme! Imagine that, piracy goes all the way back to 3500 B.C.!

The Egyptians, Greeks and Romans all had military encryption schemes. In building their pyramids, the pharoahs took elaborate precautions to disguise the entrances and passageways, yet they left clues so that the gods could enter. This gives a whole new meaning to the word 'encrypt'. The Pythagorean Society of ancient Greece used a complex code involving musical notes and mathematics. Leonardo Da Vinci wrote all of his notes backwards, so that they had to be read with a mirror. Other examples from history are common.

The United States has used encryption in war and peacetime as well. Let's not forget Paul Revere, who arranged for a comrade to signal 'One if by land, two if by sea'. In World War II, Navaho Indians were used as encrypters. Their language is different from any other, and no dictionary had ever been compiled. A message in their language could only be understood by another Navaho.

Today we have the National Security Agency, which uses the world's most powerful computers to intercept and decrypt telephone, telex and other communications. They have even been involved in commercial encryption. The National Bureau of Standards and IBM recently collaborated in the creation of a standard encryption method called DES (Data Encryption Standard). Originally it called for a 64-bit 'key' (mathematical basis) chosen by the user. The NSA persuaded them to reduce it to 60 bits, presumably so they could break it more easily. Some people believe that the NSA was involved in the design of the scheme right from the start, working through IBM, and that it has a so-called 'trapdoor' built into it so the NSA can break at will.

Perhaps the most secure encryption scheme ever invented is the one called RSA, after the initials of its inventors. Without going into detail, it involves taking huge numbers (200 digits) to different powers based on the message to be sent. Breaking it would be equivalent to solving one of the oldest problems in mathematics, that of determining all the factors of a number (other numbers that can be divided evenly into the original number) in a reasonable amount of time. It is estimated that with a 200-digit key, it would take the largest computers (like the NSA's) billions of years to crack this code by trying all the possibilities. The NSA tried to have publication of the method halted for national security reasons, but they were not successful.

Unfortunately, this method requires a lot of computer time to encrypt or decrypt a message, so it is not practical for our purposes. It does illustrate the point that the sky's the limit on how hard a method can be to crack. Realistically, we do not need anything this complex to protect programs. Most of the methods used today are quite simple. They rank about as hard as that Dick Tracy Secret Code you used as a child, and yet they manage to confound most people.

A really good method would require a considerable amount of design time on the part of the programmer and also a lot of time to decrypt as the program is being loaded. Probably the real reason we have not seen many high-caliber encryption schemes is that such a scheme is only as secure as the machine it is loaded into. That is, lifting a program from memory once it is loaded and decrypted is generally a lot simpler than cracking the encryption scheme from scratch. The main value of encryption is to prevent direct modification of the program on disk and to add a certain amount of overall difficulty to breaking it.

With that in mind, let's look at a very common encryption routine. This one uses the exclusive-OR (EOR) instruction of the 6510 processor. Before we examine the scheme itself, we need to review the characteristics of the EOR function. While we're at it we'll look at two other logic instructions, namely AND and ORA. These three functions are found on almost all processors. In fact, these functions were discovered by mathematicians long before computers were invented.

The following table summarizes EOR, AND and OR:

TABLE EN-1

EOR	0	1
0	0	1
1	1	0

AND	0	1
0	0	0
1	0	1

ORA	0	1
0	0	1
1	1	1

In each case, the function takes two binary digits (bits) as input data and gives another bit as the result. The row and column headings are used to select the inputs, and the result is found in the square where the row and column meet. For instance, performing EOR with a 1 as one input and a 0 as the other input yields the result 1.

Notice that it does not matter which order the inputs are in. Doing a 0 EOR 1 gives the same result as 1 EOR 0, namely 1. This is true for all three functions. The result of 1 AND 0 equals 0 AND 1 (both equal 0) and 1 ORA 0 equals 0 ORA 1 (both equal 1). Check for yourself that these are the results predicted by the table.

Actually, the EOR, AND and ORA instructions on the 6510 each take two whole BYTES as input. They perform their function on pairs of corresponding BITS from each input BYTE. The answer is a single byte made up of the individual result bits. Each pair of bits is done independently, so these bit-pair results are all we need to determine the byte result in any situation.

You may be wondering how the results in table EN-1 were derived, or hoping there is some easier way to remember them besides memorization. Rest assured, they represent very simple

ideas. As their names imply, AND and ORA are related to the English words 'and' and 'or'. EOR is a relative of ORA, and the idea it represents is sometimes expressed in English using 'or' also. There are a number of ways to remember them; I'll give you a couple.

One way is to give a general rule that tells, based on the inputs, when the function yields a result of 1. For AND, the result is 1 only when BOTH inputs are 1; otherwise it is 0. You can see this from the table. The ORA function gives a 1 result when ANY of the inputs is 1; it gives a 0 only when both are 0. AND and ORA are called dual functions since if you replace all the 1's in the AND table with 0's and replace all the 0's with 1's (including the row and column headers) you will get the table for ORA.

As for EOR, the rule is that it gives a 1 when EXACTLY ONE of the inputs is a 1; that is, when EITHER input is a 1 but NOT BOTH. It's called 'exclusive-or' because its rule for producing a 1 'excludes' the case where both inputs are 1. It is closely related to ORA, as you can see. They differ only in the case of two 1 inputs. Regular ORA is sometimes called 'inclusive-or' (IOR) since its rule for producing 1 'includes' the case where both inputs are 1.

Another way to remember them is to ask yourself a particular question. If the answer is YES, then the function yields a 1. If the answer is NO then the function gives a 0. For AND, ask yourself 'Are BOTH inputs 1? '. For ORA, ask 'Is ANY input a 1? '. For EOR, perhaps surprisingly, you can ask yourself 'Are the inputs DIFFERENT ?'.

One thing you should not rely on is the use of these words in English, since they aren't used consistently. Much if not most of the time when we use 'or' we really mean 'exclusive-or'. For instance, if you tell your kids 'Clean up your room OR I'll punish you' you mean either one or the other will happen, but surely not both!

When you make a statement where you mean that either or both things could be true, you are using the regular ORA, as in 'My disk drive is out of alignment OR this disk is screwed up'. Maybe both! We can make this distinction clearer by using 'either/or' for EOR and 'and/or' for ORA. Thus we could say 'EITHER you clean your room OR I'll punish you' and 'My disk drive is out of alignment AND/OR this disk is screwed up'. There are also some occasions where we use 'and' where we mean 'or', but they are less common.

To reinforce your understanding of these functions, let's look at a few examples involving whole bytes. Below I have given the result when the same two sample bytes are AND'ed, ORA'ed and EOR'ed. The two bytes were chosen so that every possible combination of bits is illustrated (twice in fact). To distinguish the original bytes let's call the top one the INPUT

and the bottom one the VALUE (but remember that you'll get the same result regardless of which one is which). For reference the hex equivalent for each byte is given and the corresponding 6510 code is shown.

```
     BINARY      HEX          BINARY      HEX          BINARY      HEX
     0011 1010   $3A          0011 1010   $3A          0011 1010   $3A
EOR  0101 1100   $5C     AND  0101 1100   $5C     ORA  0101 1100   $5C
     0110 0110   $66          0001 1000   $18          0111 1110   $7E

   A9 3A  LDA #$3A          A9 3A  LDA #$3A          A9 3A  LDA #$3A
   49 5C  EOR #$5C          29 5C  AND #$5C          09 5C  ORA #$5C
```

Note again that each pair of bits (top and bottom) is operated on independently to give the result bit. Be sure to verify each result using table EN-1 to help complete your understanding.

There is an interesting phenomenon which we can illustrate with our examples. If we take the result of the EOR function and EOR it again with the value byte, we will get the input byte back!

```
     BINARY      HEX                          BINARY      HEX
     0011 1010   $3A   Input                  0110 0110   $66   Result
EOR  0101 1100   $5C   Value          EOR     0101 1100   $5C   Value
     0110 0110   $66   Result                 0011 1010   $3A   Input

     A9 3A LDA #$3A   Input                   A9 66 LDA #$66   Result
     49 5C EOR #$5C   Value                   49 5C EOR #$5C   Value
```

We call EOR a reversible function because of this. Actually, we can EOR the result with either original byte, value or input. No information is lost in the EOR process; if we know the result and one of the bytes we can always recover the other one. This is a handy feature to exploit in an encryption scheme since we can use the same routine to both encrypt and decrypt! We'll see an example of this soon.

If we try this with AND or ORA, we won't be able to reverse them.

```
     BINARY      HEX                          BINARY      HEX
     0011 1010   $3A   Input                  0001 1000   $18   Result
AND  0101 1100   $5C   Value          AND     0101 1100   $5C   Value
     0001 1000   $18   Result                 0001 1000   $18   Same

     49 3A LDA #$3A   Input                   49 18 LDA #$18   Result
     29 5C AND #$5C   Value                   29 5C AND #$5C   Value
```

```
      BINARY     HEX
      0011 1010  $3A   Input
ORA   0101 1100  $5C   Value
      0111 1110  $7E   Result

      49 3A LDA #$3A   Input
      09 5C ORA #$5C   Value
```

```
      BINARY     HEX
      0111 1110  $7E   Result
ORA   0101 1100  $5C   Value
      0111 1110  $7E   Same

      49 7E LDA #$7E   Result
      09 5C ORA #$5C   Value
```

Note that, in fact, in all cases the result stayed the same. The original bytes have contributed all they can and can't alter the result any more. So we can't reverse these functions by repeating them with one of the original bytes. Can we reverse them some other way? Unfortunately not, since information has actually been lost in this process. Looking back at table EN-1 we can see why this is true.

Suppose we know that the original value bit of an AND operation was a 0 (top row of the AND table). If the result bit was a 0, we can't tell if the input bit was a 0 or a 1, since both would have given a 0 (it's the only possible result). If we know the original value was a 1 (second row of the table), then we CAN tell what the input was: If the result is 0, the input was 0; if the result is 1, the input was 1. This suggests that we always make sure to AND with a value of 1 so as to be able to get back the input. Unfortunately, AND'ing with 1 doesn't change the input at all!

The exact same thing happens with ORA, except for reversing the roles of 0 and 1 (since they're dual functions as defined above). The only way to be sure you can recover an input is to have originally ORA'ed it with 0, but this does not change the input. Not much of an encryption scheme!

With the preliminaries out of the way, let's look at the role of EOR in encryption. One particular value is worth mentioning for use with EOR. If you look back to table EN-1, you will see that if the value bit is 1, the result bit will be the opposite of the input bit. For whole bytes, this means that if we use a value of $FF (binary 1111 1111) the result byte will equal the input byte with all the bits 'flipped', that is, all 0's will be replaced with 1's and vice versa. This is a fairly common value to see in program protection methods.

We'll use this value in our first encryption routine. The following routine takes one page (256-byte chunk) of memory at the beginning of the BASIC area, EOR's it with the value $FF, and puts it back in the same place, byte by byte. It's called 'ENCRYPT BASIC' on the program disk.

```
1000 A0 00      LDY #$00        Start with offset of zero
1002 B9 01 08   LDA $0801,Y     Load A from loc. $0801+offset
1005 49 FF      EOR #$FF        Exclusive-or A with value $FF
1007 99 01 08   STA $0801,Y     Put encrypted byte back
100A C8         INY             Increase offset
100B D0 F5      BNE $1002       Branch if more to do
100D 60         RTS             Finished; return to BASIC
```

A few words of explanation are in order. The routine uses indexed addressing to load and save the accumulator. This form of addressing uses the Y register as an offset from the address specified in the LDA instruction. Thus the actual address to be loaded from or saved to is the sum of the address given ($0801) and the value of the index (Y). As we increase the index (INY) we step through memory one byte at a time. After Y reaches $FF (255), it will reset to zero when incremented again. As long as Y hasn't been reset to zero the branch instruction (BNE) will continue the processing. Eventually Y will be reset to zero; this time the branch is not taken and we fall through to the RTS, which returns us back to BASIC.

This takes care of how the looping is set up. The actual work of encrypting is done by the EOR instruction at location 1005. This instruction EOR's the contents of the ACCUMULATOR (A) directly with the value specified ($FF). This is called immediate addressing and is specified by a '#' before the value. The result is placed back in the accumulator.

To try this routine out, first load it from the program disk with LOAD 'ENCRYPT BASIC',8,1 (don't do this from a monitor). Next you need to load a BASIC program to try it out on. There's one on the disk for just this purpose: LOAD 'BASIC SAMPLE',8. List it to see what it looks like normally. To encrypt it, execute the routine with a SYS 4096 (= $1000) from BASIC.

Now try to list it again. Your nice BASIC program is a mess! All is not lost, though; because we used EOR the program is easily recovered by running the same routine over it again. Try it! Do another SYS 4096 and list again. Voila!

You can go back and forth from encrypted to decrypted form as many times as you wish. You can even save the encrypted version with a regular BASIC save. However, loading it back in from BASIC presents a bit of a problem. BASIC attempts to re-link the statement line pointers when it finishes the load (see PPM Vol. 1 for a discussion of BASIC line pointers). In almost all cases this will mess up the encrypted program.

The solution is to load the program back in from machine language. You can kill two birds with one stone by putting the loading and decryption routines into an autoboot program. You will also have to set the BASIC variable pointers (loc. $2D-$34) to enable the program to run. This is done by BASIC automatically at the end of a load.

Now let's look at a more general encryption routine. This one is called 'ENCRYPT ANY' on the program disk. Here is what it looks like:

```
C000  A6 FD     LDX $FD         No. of pages to do
C002  A0 00     LDY #$00        Start with offset of zero
C004  B1 FB     LDA ($FB),Y     Load A indirect, indexed
C006  45 FE     EOR $FE         EOR A with contents of loc. $FE
C008  91 FB     STA ($FB),Y     Replace encrypted byte
C00A  C8        INY             Increase offset
C00B  D0 F7     BNE $C004       Repeat if not done with page
C00D  E6 FC     INC $FC         Set pointer to next page
C00F  CA        DEX             Decrease # pages left to do
C010  D0 F2     BNE $C004       Repeat if not done
C012  00        BRK             Jump back to monitor
```

Memory locations used:

00FB-FC	Two-byte pointer to start of code being processed
00FD	Number of pages (256-byte chunks) to process
00FE	Location of constant to be EOR'ed with code.

Now for the gory details. Locations $FB-FC hold a two-byte POINTER to the beginning of the code to be processed. The code itself doesn't start at $FB; it can be almost anywhere. This pointer is in standard lo-byte, hi-byte order; that is, if you wanted it to encrypt code starting at $1234 you would put a value of $34 into location $FD and $12 into $FE.

Location $FD tells the routine how many 256-byte pages to process, starting at the address given by $FB-FC. The minimum value you should use is 1; if you put a 0 here it will try to do all of memory!

Location $FE holds a constant which will be EOR'ed with each byte of code to produce the encrypted version. You can use any value you want here. Remember that a value of $FF (binary 1111 1111) will flip all the bits in the result byte to the opposite of the original, as in our first routine. Note that a value of $00 will not cause any change at all! Values in-between will flip only those bits in the result byte that have a 1 in the corresponding position in your value.

When you execute the routine, it first loads X with the number of pages to do and then sets Y to zero. The Y register is used as an offset as in our first example. This time, the address to load A from or store it to is formed by first getting the starting address from $FB-$FC, then adding the contents of Y to it. This is called indirect indexed addressing. The indirect part is indicated by putting $FB in parentheses. You can read these parentheses as 'the contents of the two bytes starting at'. The indexing by Y is indicated by the ',Y'.

Having gotten the byte of code to be encrypted into the accumulator, it is then EOR'ed with the value you stored at $FE. Note that it is not using the VALUE $FE; it is getting its

value from LOCATION $FE. The encrypted byte is then stored back into its original spot. Then Y is incremented. If Y has not been reset to zero, we loop back to $C004 to continue the page.

When Y does reach zero, we have finished this page, so we increment the hi-byte of the address pointer ($FC) to point to the next page. Next we decrement X to reduce the number of pages left to do. If X is not zero, we have more to do so we again loop back to $1004. When X reaches zero we are done, so we exit back to the monitor with a BRK instruction.

To use the routine, first load in a monitor that doesn't reside at $C000 (LOMON from the program disk will do). Next, load the routine from the program disk with L 'ENCRYPT ANY', 08. Then load in your program to be encrypted. Make sure it doesn't use the same area of memory as the monitor or encryption routine. If necessary, you can transfer the encryption routine anywhere you want in memory without having to change it. The only possible conflict would be if your program occupies locations $FB to $FE. This is not very likely, but the routine is easily changed to get around it if necessary.

Before executing the routine, you must put your values into the memory locations at $FB to $FE. The memory command M 00FB can be used for this. It will display the current contents of these locations. You can then type your values over the current ones and hit RETURN. The monitor will store the values.

Let's try it out. We need a piece of code to use for an example. Hmm... why not let it encrypt a copy of itself? Load in your monitor and the routine from the disk. Transfer a copy of the routine down to, say, $6000 with the transfer command: T C000 C012 6000. Now set the start pointer ($FB-$FC) to $6000 (remember to reverse the order of the bytes), set the number of pages ($FD) to $01, and put a value in the constant location ($FE).

After all this, you can execute the routine with a G 1000 command from the monitor. In a flash the code is encrypted and you're back in the monitor. Now try to disassemble the encrypted version at $6000. Depending on the constant you used, you'll generally find absolute garbage. Occasionally you'll get a few bytes here and there that look like a valid instruction. This can even increase the protection value of encryption since it may mislead a pirate looking at your code.

Now you want to get your code back. Before re-executing the routine you'll have to go back and put the starting address at $FB-FC again. This is because the routine alters the pointer as it executes (actually, it only alters the high byte at $FC). Do a G C000 and look at the code again. It should be completely restored.

If you want to look at this from the pirate's point of view, get set up with the monitor and encryption routine in memory. Then load the program called 'ENCRYPTED' from the program disk. It's less than one block long and starts at $6000. It was encrypted with this routine using a constant of my own choosing. Your task is to decrypt it. I'll even give you a hint: I didn't use $00 or $FF. Happy hunting!

This little exercise will demonstrate that even knowing where the code is and having the routine to decrypt it, you still need to know the 'magic' value. This suggests having the value loaded in separately from the decryption routine, perhaps based on some other protection scheme. By combining schemes in this manner, you multiply the difficulty involved in breaking the program.

We've really only scratched the surface of encryption in this chapter. We haven't even exhausted the possibilities of EOR. For example, some protection schemes EOR each byte with the preceding one, rather than the same constant each time. This sets up a 'chain' that has to be followed to properly decrypt the code.

Another idea is to use the ADC and SBC instructions of the 6510 (Add with Carry and Subtract with Carry). You can use these alone or in combination with EOR. For instance, to encrypt a byte, ADC a value and then EOR with another value. To decrypt, first EOR and then SBC using the same two values respectively.

You might also find a way to use the shift and rotate instructions (ASL, LSR, ROL and ROR), increment and decrement instructions (INC and DEC) or even the BASIC multi-byte math routines (+, -, *, / etc.).

Basically, anything you can do to a byte or group of bytes can be an encryption scheme if it can be 'undone' reliably. This chapter should serve as a springboard to give you a push in the right direction. The possibilities are limited only by your own creativity.

Many times during the course of this manual we refer to modifying the KERNAL ROM or some other ROM chip. You might just wonder how this is to be accomplished. A ROM chip is a permanent memory chip. The instructions contained on the chip are a permanent part of the physical construction of the chip and may not be altered. How then do we modify the KERNAL ROM?? With an EPROM programmer, that's how.

Any program that may be on a ROM can be placed on an EPROM. An EPROM (Eraseable, Programmable, Read Only Memory) may be programmed, erased and reprogrammed thousands of times. When the EPROM is programmed it will retain the information even when power is turned off, just like a ROM. Whereas RAM (Random Access Memory) will lose its memory shortly after power is turned off. EPROM's may be considered as a way to permanently store a program or other data, just as a disk may be considered as permanent memory. An EPROM is similar to a disk in that information may be stored on the EPROM and later erased if necessary, then more information may be stored back on the EPROM.

EPROMs are not truly permanent memory. They may be damaged by physical abuse, they are subject to electrical shocks, static electricity and other forms of failure. In short, an EPROM is similar to a disk, with the proper care they will last for years. If an EPROM is abused it will not last very long.

Why then would we want to put our data on EPROMs?? EPROMs, like ROMs, may be installed into the computer, the disk drive or the cartridge port. We may modify the information contained on the EPROM to perform custom operations and then re-install the EPROM into the computer. We can have our own custom KERNAL, BASIC, DOS or cartridge routine. When information is stored on an EPROM we don't have to wait for it to load in from disk.

Earlier we discussed the RESET, INTERRUPT and BREAK functions of the C-64. If, for instance, we wanted to modify the RESET routine of the computer it would only be necessary to burn a new EPROM and install it in your computer. If we wish to make a custom DOS for the disk drive all that is required is to burn an EPROM and install it in the drive. Pretty simple, isn't it??

Let's follow the procedure to make a new EPROM for the disk drive. We will modify the DOS ROM that is located from $E000 to $FFFF. This is the chip located at the right rear of the disk drive and numbered 901229. In every drive that we have examined this chip is socketed and may be easily removed. This way we may easily replace the DOS ROM with our custom EPROM. Wait a minute, even if we can just replace the DOS ROM with an EPROM how do we program the EPROM with our special routine??? With an EPROM programmer of course!! Just any old EPROM programmer

should work, but if you are going to buy one we have a recommendation: the PROMENADE by JASON RANHEIM (available from CSM). This is the most versatile, cost effective and durable EPROM programmer we can find. It retails for around $100.00. It can program more types of EPROMs than EPROM programmers costing over $2000.00. The PROMENADE is packaged in an durable aluminium housing and is really made to last. All of the internal circuitry is protected from overloads and improperly inserted EPROMs. So if you should happen to make a mistake in the type of EPROM or how you install the EPROM you can not damage the EPROM programmer (although it is possible to blow a chip through a mistake).

Let's say that you have an EPROM programmer how do you decide which type of EPROM to use? (The PROMENADE will program over 25 different types of EPROMs). Well, the EPROM that replaces the ROM in the drive is a MCM 68764. The MCM 68764 is a pin for pin replacement for the ROMs in the drive and the ROMs in the computer. You just unplug the ROM and insert your custom EPROM, pretty easy huh? The only problem with the MCM 68764 EPROM is the price: they retail at $40.00 each (ouch). It can get very expensive burning new EPROMs for the drive and the computer at $40.00 each. But wait, there is a lower cost solution. This consists of a 2764 EPROM and an adapter (both available from CSM). The ROM in the disk drive has 24 pins (as does the MCM 68764). The 2764 EPROM has 28 pins. The adapter allows you to use the 2764 EPROM in the disk drive. Why bother with an adapter and a 28 pin EPROM?? The price is why. The 2764 and the adapter may be purchased for less that $20.00 for both items. That's less than 1/2 the retail price of the MCM 68764. Now the price to modify your drive and your computer is down within the reach of the average person.

The 2764 EPROM has 8K of memory and directly plugs into the cartridge boards for the C-64. Some 2764's have reached the surplus market and sometimes can be found at HAMfests for as little as $3.00 each for perfectly good used EPROMs (a real bargain). For our money we feel that the 2764 EPROM is the best buy dollar for dollar. If you only need to put a 2K program on an EPROM you may use the 2764. You can program only the amount of memory that you need with the PROMENADE. Then, at a later date, you can program the rest of the 8K chip. Just as you can save to a disk until it is full, you may also save to an EPROM until it is full. After you have programmed an EPROM you can erase it and reprogram it, just like you can erase disks. In order to erase an EPROM it is necessary to use ultraviolet (UV) light. All you have to do is expose the EPROM to UV light for 10-15 minutes and it is fully erased. There are many commercial EPROM erasers on the market today, the one that we prefer is call DATARASE by WALLING CO, priced under $40.00 (available from CSM). EPROMs may be erased by other sources of UV light (such as the sun), but we don't recommend it.

O.K. back to programming EPROMs. The PROMENADE plugs into the modem port of the C-64 (don't use any EPROM programmer on the SX-64 due to a power supply problem of the SX). The software for the PROMENADE is included and may be loaded from the disk and RUN. Type: LOAD"PROMOS*",8: then (RETURN), now type RUN. The screen will display a copyright message and return to the 'READY' prompt. You are now ready to modify the DOS ROM, so lets go.

1). Carefully remove the $E000 to $FFFF ROM from the disk drive.

2). Type: Z (RETURN) - this will zero the socket on the PROMENADE.

3). Insert the DOS ROM into the PROMENADE socket and close the lever on the socket. Be sure to insert the ROM as shown on the PROMENADE.

4). Read the data from the ROM with the following commands: the (L) = the English pound key:
(L)8192,16383,0,48 (RETURN)

5). The data from the DOS ROM has now been stored in the computer from memory location 8192 decimal ($2000) to 16383 decimal ($3FFF). The '0' means to start reading from the very first byte of the EPROM (byte 0) and the '48' determines which type of EPROM (24 pin, 28 pin, 2K, 4K, 8K etc). The PROMENADE manual fully describes the commands to be used for various EPROMs.

6). If you wish to modify the DOS ROM or to make a disk copy of the ROM now is the time to do this. If you only want to burn an exact replacement of the ROM go to step 7 now. Load a ML monitor and save the DOS ROM memory out to disk. You should use a ML monitor that resides at $C000 (49152) so that there is no chance of over writing the DOS ROM. Remember that the DOS will reside in the computer from $2000 to $3FFF. If you wish to modify the DOS, it may be accomplished very easily from the ML monitor and the modified version may then be saved to disk. A little later on we will give you some tips on where and how to modify the DOS. If you have entered a ML monitor it may be necessary to power down and reload the PROMOS software from disk after you have saved out the modified version of the ROM. Then reload the modified version of your DOS ROM. NOTE: this depends upon the type of ML monitor that you are using, some monitors may exit properly into PROMOS, others do not.

7A). If you are going to use the 2764 EPROM to replace the ROM all you have to do is insert the 2764 into the PROMENADE and use the following commands to burn the EPROM. The (PI) = the pi symbol (shifted up arrow)
(PI)8192,16383,0,5,7 (RETURN)

Remove the EPROM from the programmer when it has finished programming (1-2 minutes) and insert it into an adapter.

7B). If you are going to use the MCM 68764 EPROM use the following commands.
(PI)8192,16383,0,48,15 (RETURN)

8). Install the EPROM into the disk drive and you are done. Programming an EPROM takes less that 10 minutes once you have become proficient with the EPROM programmer.

MODIFICATION OF THE DOS

HARDWARE MODIFICATION OF THE OPERATING SYSTEM FOR MORE THAN THIRTY-FIVE TRACKS OR FOR EXTRA SECTORS.

Caution: Some drives may not be physically able to go to track 40. The read/write head may become stuck at track 38 on some drives. This is not serious, just go into the drive and free the head with your hand if it gets stuck.

This method will require the 'burning' of replacement EPROMS for the disk drive. You will be able to add extra tracks or vary the number of sectors on a track when you use this technique. One problem will be evident when you change the number of tracks on the disk: You will only be able to list the directories of a disk with the same number of tracks as the drive is set up for. For instance: if you have a forty track drive you can only list the directories of the forty track disks. This is due to the way the 1541 wants to find the BAM, NAME, and ID of the disk. As you add extra tracks you must also add room for increased area in the BAM (each track requires 4 bytes for the BAM). The BAM will be expanded into the area normally used by the NAME and ID of the disk. The NAME and ID must be located immediately following the BAM of the last track. If you add five tracks to the disk you must increase the BAM by 20 bytes. The NAME and ID will also be moved 20 bytes on the disk (the drive will automatically locate the NAME and ID immediately after the BAM).

The items to change for the extra tracks will be the location of the end of the BAM (four bytes for each track) and the comparisons for the maximum number of tracks. The other area of memory to change, whenever you modify the DOS, is the ROM TEST ($EAE4-$EAE9). This is where DOS checks the ROM to insure that there has not been any malfunctions of the operating system by performing a checksum on the DOS ROMs. We will totally bypass the ROM test, this allows you make any modification to the DOS that you wish. These are the following locations to change for a forty track drive.

$D08C	CHANGE TO	$A4	(LOCATION OF BAM+4 bytes/track)
$EAE4	CHANGE TO	$EA	(ELIMINATE ROM TEST)
$EAE5	CHANGE TO	$EA	(ELIMINATE ROM TEST)
$EAE8	CHANGE TO	$EA	(ELIMINATE ROM TEST)
$EAE9	CHANGE TO	$EA	(ELIMINATE ROM TEST)
$EEEE	CHANGE TO	$A4	(LOCATION OF BAM+4 bytes/track)
$FD90	CHANGE TO	$29	(MAX. # OF TRACKS +1)
$FE88	CHANGE TO	$A4	(LOCATION OF BAM+4 bytes/track)
$FED7	CHANGE TO	$29	(MAX. # OF TRACKS +1)

If you wish to modify the number of sectors per track, change the code from $FED1-$FED4. Each byte represents the number of sectors on the different tracks ($FED1 is for TR 31-35;$FED2 is for TR 25-30; ETC.). There are certain limits as to the number of sectors on a disk. Don't try to add too many, it won't work. As a matter of fact, later in this manual, we will cover just how many sectors you can put on a track and why. Bytes $FED8-$FEDA indicate where the sector change will occur (tracks 31, 25 & 18).

How to make the modifications to the disk drive: First locate and remove the ROM chips from the disk drive. The $C000-$DFFF chip is marked 325302, the $E000-$FFFF chip is marked 901229. Both chips are located at the rear of the circuit board. The $C000-$DFFF chip is not socketed on some disk drives. If your chip is not socketed, don't try to remove it from the board (unless you are proficient at this type of work), just modify the $E000-$FFFF chip. If you just modify the $E000 chip the drive will not format 40 track disks nor will the BAM operate properly (unless you modify both chips). But, you will at least be able to read and write 40 tracks if you only modify the $E000-$FFFF ROM chip.

Insert the $C000-$DFFF ROM into your PROMENADE. Down load the chip memory into the computer. Use a ML monitor to modify the code as specified above. Follow the instruction outlined above and in the PROMENADE manual. Then remove the original chip from the PROMENADE and insert an erased EPROM into the PROMENADE and burn the EPROM. Insert the EPROM into the proper socket of the disk drive and use the same procedure on the other chip.

The total time required to complete the process is less than 1 hour and can provide some very interesting results. Don't be afraid to experiment with the computer or disk drive. Try some modifications on your equipment and see what fun you can have.

Later on in this manual we offer some more advanced insights on EPROM programming. The information contained in those chapters is not essential to programming EPROMs. If fact the information provided is for the experienced programmer only!! If you only wish to program a few of the 'normal' EPROMs, it does not get any more difficult than presented here.

Bear with us in this chapter. This information can get very confusing. It is important to have at least a basic understanding of how the microprocessor and the associated memory circuits work. We are not going to make an electronics engineer out of the reader. We are only going give you some ideas on how the computer works. We will use some big fancy words in this chapter. Don't try to remember all of them. We will explain the important terms and concepts.

The C-64 uses the 6510 as its microprocessor chip. The 6510 is closely related to the more common 6502 microprocessor. The 6510's internal architecture is identical to the MOS Technology version of the 6502. This is to provide software compatibility. Both the 6502 and the 6510 use the same instruction set. The most important difference between the 6510 and the 6502 is the eight bit Bi-directional I/O port feature of the 6510. Actually, only six bits (I/O lines) are available in the version used in the C-64. The other two bits of the I/O port have been reserved for the Non Maskable Interrupt and the Ready lines.

Ok, let's try to understand what all those big fancy words mean. First of all let's define just what a microprocessor is. A microprocessor is the central processing unit of the computer. The microprocessor will perform a large number of functions, including:

1). Getting instructions and data from memory.

2). Decoding instructions.

3). Performing arithmetic and logic operations specified by the instructions.

4). Providing timing and control signals for all the components of the computer.

5). Transferring data to and from Input & Output (I/O) devices (printers , disk drive, monitor, keyboard, etc.).

6). Responding to signals from the I/O devices (interrupts, etc).

The 6510 processor will perform these functions based upon what the program instructions tell the processor to do. This is what makes the computer so flexible. When we want to change the function of the computer, all we have to do is give the computer new instructions (such as a program). The 6510 will perform any combinations of functions based solely upon the instructions that it receives.

The 6510 can address up to 64K (65536 bytes) of memory. This is the maximum number of addresses (memory locations) that the 6510 can look at any one time. A memory location is an area in the computer that can contain (store) a value. The value contained in each memory location must be between 0 and 255 ($00 to $FF). These memory locations are where the computer program will be stored. The memory can either be in RAM (Random Access Memory) or in ROM (Read Only Memory). It does not matter to the computer if the program resides permanently on a computer chip (ROM) or will be erased when the power is lost (RAM).

The C-64 contains a full 64K of RAM and it contains another 1K of 4-bit Color RAM. It also contains a full 20K of ROM. Wait a minute, 64K + 1K + 20K = 85K. The 6510 can only address a maximum of 64K!! How is it possible that we have 85K of memory in the C-64?? Well, the 6510 microprocessor can only see 64k of memory at any one time. The rest of the memory is hidden from the microprocessor by a device known as a Program Logic Array (PLA). The PLA will switch various sections of memory in or out depending upon the specific requirements of the computer.

The function of the microprocessor and the PLA is a very important relationship that should be understood. Quite simply the PLA will turn one section of memory on and another off, based upon the requirements of the program. If the programmer wants the microprocessor to see RAM at the memory location normally occupied by BASIC ROM, all the programmer has to do is change the value stored in memory location $0001. Memory location $0001 is the eight (six) bit I/O register. Based upon the bit pattern in location $0001 the PLA will automatically reconfigure memory by enabling and/or disabling the proper memory chip(s). Memory may also be reconfigured through the PLA by installing a cartridge in the computer. Just as we can change the memory configuration by changing location $0001, we can accomplish the same task by grounding the EXROM and/or GAME lines of the cartridge port.

We have now established that the C-64 does contain 85K of memory and that the 6510 processor can access only 64K of it at a time. By simply having the PLA enable or disable the various chips, the computer can see different memory at the same location. The PLA may be controlled either by software (a computer program) or by hardware (the cartridge board).

Now that we have a basic understanding of how the PLA and the microprocessor function, let's see what happens on power-up.

When we first turn our computer on, the 6510 microprocessor will set its LORAM, HIRAM and the CHAREN lines high. For our purposes we will consider a line to be high when it is 5 volts (appx.) and a line is said to be low when it is at 0 volts. The computer will interpret a line that is high as a binary 1 and interpret a line that is low as a binary 0. The program

counter, the stack pointer and the flags of the status register will all contain indeterminate (random) values. At this point the microprocessor is lost. Since none of the registers retain any information after power is turned off, all values will have to be reset before any meaningful data may be processed. How then, does the computer reset these values? With a RESET, of course! The C-64 contains a special circuit that forces the computer to RESET upon power-up. Every time the power is turned on, the microprocessor will be RESET. This is exactly the same RESET condition as when you press your reset button (providing you have installed one).

When the computer is RESET the microprocessor will set the LORAM, HIRAM and CHAREN lines high (5v) and the microprocessor will do an indirect jump to the address contained at memory locations $FFFC and $FFFD. These two locations are referred to as VECTORS. A VECTOR does not contain the actual routine that the computer will execute upon RESET. What this VECTOR does contain is the LOCATION where the RESET routine may be found. In other words, the locations $FFFC and $FFFD will tell the computer where to find the actual RESET routine. ON the C-64 the memory location $FCE2 (64738 decimal) is the actual entry point into its RESET routine. We have covered the RESET routine that the C-64 uses elsewhere in this manual, so we will not cover it in depth here. It is important to note that the software RESET (SYS 64738 or JMP $FCE2) is different from the hardware RESET (actually grounding the RESET line of the microprocessor). The hardware RESET will perform all of the functions of the software RESET but first it will set the LORAM, the HIRAM and the CHAREN lines high. Any references made in this chapter to a RESET refers to a hardware RESET unless otherwise noted.

As you can see, the computer will 'look' to the memory location $FFFC and $FFFD for its RESET VECTOR. If we were to 'burn' ourselves a new KERNAL ROM (EPROM) and substitute it for the original ROM, we would be able to force the computer to perform a different RESET routine. It must be noted that if we wish to use an alternate RESET routine it will be important to perform a few specific functions early on in our routine.

1). SEI - Set the IRQ interrupt disable.

2). The stack pointer should be set to a specific value (usually $FF).

3). The Decimal flag should be cleared (CLD).

4). The Carry flag should be either set or cleared (as required) prior to performing any arithmetic functions.

5). Clear the interrupt prior to leaving the RESET routine.

These are functions that should (must) be performed in any RESET routine.

We have established that the microprocessor must be RESET upon power-up. Now let's find out why the microprocessor looks to ROM for its RESET routine rather that looking to RAM.

Earlier we mentioned that the 6510 contains an 8-bit Input/Output (I/O) port. In the version of the 6510 microprocessor that is contained in the C-64 only 6 lines are used for I/O, the other two lines are used for other purposes (NMI and READY lines). Each of these six lines correspond to bits of memory location $0001. If we consult figure xx1 we can see six pins of the microprocessor that are labeled P0, P1, P2, P3, P4, & P5. These pins correspond to the bits 0 thru 5 contained in memory location $0001, respectively (i.e. P0 will reflect bit 0 etc.).

Further explanation of just what an I/O port is required. An I/O port is a special area of the microprocessor that is reserved for communications with other devices. It consists of a few selected lines (pins) of the microprocessor that may be controlled by another device (when the lines act as inputs) OR the lines may be used to control other devices (when the lines act as outputs). When any I/O line is set to input the microprocessor will automatically sense any change in the values (voltage level) placed on this line by other devices. A change in the voltage level applied to the line will cause the corresponding bit in memory location $0001 to change. If the voltage applied to one of these pins is 5 volts, the microprocessor will interpret this as a one and change the appropriate bit in location $0001 to a value of one. Correspondingly, when an I/O line is set to output the microprocessor will 'look' to memory location $0001 and set the appropriate output line high (5v) or low (0v) based upon the appropriate bit in $0001.

We have established that location $0001 is the actual I/O port. We now need to know how to switch these lines from Input to Output. This is controlled by memory location $0000. If a bit of memory location $0000 is set to a 1, the corresponding bit of location $0001 is set to an output. Likewise, if a bit of memory location $0000 is set to a 0, the corresponding bit of location $0001 will be set to input. For example, if we set bit 0 of location $0000 to a 1, bit 0 of location $0001 will be an output. Each line of the I/O port may be set independently of the other lines. Memory locations $0000 and $0001 are contained 'on board' the microprocessor. When we load or store a value in memory locations $0000 & $0001 this is actually being done inside the microprocessor. All other memory locations are outside of the microprocessor in the 'normal' RAM or ROM or I/O devices.

As we said before, only six lines of the I/O port are used by the microprocessor as actual I/O lines. Three of the six I/O lines are used only for the cassette recorder and will not be discussed any further (P3, P4 & P5 are for the cassette). Lines

P0, P1 & P2 are used by the microprocessor to control access to the various areas of memory via the PLA. Refer to figure 6510 & PLA-1 for the following discussion. On power up or on RESET, the microprocessor will automatically set lines P0, P1, & P2 to outputs and force them high (5v). This should be considered as the 'normal' state of the microprocessor. During a RESET, the RESET routine will verify that the proper lines are set to outputs and that they contain the proper values by storing the appropriate values at locations $0000 & $0001. If we examine memory after a RESET, we will see that location $0001 contains a value of $37. The low nibble (7) indicates that bits 0, 1 & 2 are indeed set high. We will also see that memory location $0000 contains $2F. The low nibble (F) indicates that lines P0, P1, P2, & P3 are set to outputs.

We have already mentioned that the PLA controls the area of memory that the microprocessor will see. Let's take a more specific look at what controls the PLA. Earlier we mentioned that upon power up or RESET the LORAM, the HIRAM and the CHAREN lines of the microprocessor will be set high (5v). These lines correspond to three bits of memory location $0001 (bits 00, 01 & 02 respectively).

Line P0 is the LORAM line. Normally this line is high (5v). The LORAM line controls the BASIC interpreter memory only ($A000-$BFFF). When the LORAM line is high (5v) the PLA will cause the microprocessor to see BASIC ROM at this location ($A000-$BFFF). When the LORAM line is low (0v) the PLA will cause the microprocessor to see RAM at this location ($A000-$BFFF). We can easily control the LORAM line by changing memory location $0001 from a $37 to a $36 (setting bit 0 to a 0).

Line P1 is the HIRAM line. Normally this line is high (5v). The HIRAM line controls the KERNAL memory ($E000-$FFFF) and BASIC memory ($A000-$BFFF). When the HIRAM line is high (5v), the PLA will cause the microprocessor to see KERNAL ROM at this location ($E000-$FFFF) and allows LORAM to control BASIC ROM. When the HIRAM line is low (0v) the PLA will cause the microprocessor to see RAM at BOTH KERNAL AND BASIC locations ($A000-$BFFF AND $E000-$FFFF). We can easily control the HIRAM line by changing memory location $0001 from a $37 to a $35 (setting bit 1 to 0). Be sure to set the interrupt (SEI) prior to setting the HIRAM line low and then clear the interrupt (CLI) upon resetting the HIRAM high.

It is important to note that if we wish to turn off the KERNAL ROM it will also cause the BASIC ROM to be turned off. If the computer is configured, using HIRAM, to see RAM at $E000-$FFFF it will also see RAM at $A000-$BFFF. Remember that BASIC may be turned off by itself (with LORAM), but if we turn off the KERNAL ROM (with HIRAM) we will also turn off the BASIC ROM!!!

If line P0 (LORAM) and P1 (HIRAM) are both set low (0v) a very interesting thing will occur. The computer will now be

configured to see all 64K of RAM. The KERNAL ROM, the BASIC ROM and the I/O devices at $D000-$DFFF will all be switched out. The microprocessor will now see only the 64K of RAM. This configuration will allow the microprocessor to use all 64k of RAM. We can easily control both the HIRAM and the LORAM lines by storing a value of $34 at memory location $0001. Keep in mind that the user will have to switch the I/O devices at $D000-$DFFF back in for any I/O operations (communications with the screen, disk drive, keyboard, etc.).

Line P2 is the CHAREN line. Normally this line is high (5v). The CHAREN line controls the I/O devices at $D000-$DFFF. When the CHAREN line is high the microprocessor will see I/O devices (VIC chip, SID chip and the CIAs) at this area of memory. When the CHAREN line is low (0v) the microprocessor will see the 4k character generator ROM at this location. The computer will not be able to access any I/O devices if the CHAREN line is low. The only time that the CHAREN line would normally be held low is if the user wished to download the character ROM to RAM. To set the CHAREN line low all one has to do is to change memory location $0001 from $37 to $33 (setting bit 2 to 0).

We have now covered all the software selections of the PLA. Remember that the PLA will select a specific area of memory based upon the requirements of the microprocessor. All one has to do is to set byte $0001 to a specified value and the PLA will do the rest. Before we proceed into the hardware control of the PLA, let's take a look at the normal sequence of operation of the microprocessor.

When the microprocessor is in operation the program counter will keep track of the memory location that is currently being accessed. The following is a brief description of the sequence of events that occurs when the microprocessor gets a byte of data from memory.

1). The microprocessor will send out the address of the current byte of memory that is requested. The microprocessor will also specify if the byte is going to be read or to be written. In this example we will assume a read of data (LDA - LoaD the Accumulator).

2). The PLA will decode the address specified by the program counter and select (enable) the chip required by the microprocessor.

3). The selected memory chip will decode the address specified by the microprocessor and select the appropriate memory location from within the chip.

4). The selected chip then makes the data available and the microprocessor loads the data into the appropriate register.

All of this sounds pretty time consuming, doesn't it?? Actually the time required to fetch a byte of data from memory takes less than 1 millionth (1/1,000,000) of a second in the C-64. The instruction JMP $4000 (4C 00 40) takes only 3 millionths of a second to execute. 1 millionth is used to fetch and decode the instruction (4C), 1 millionth to fetch the low byte (00), 1 millionth to fetch the high byte (40) and place these values on the program counter. Thereby effecting the JMP instruction in only three clock cycles.

HARDWARE CONTROL OF THE PLA

The memory that the microprocessor sees may also be controlled by hardware. Hardware control requires an actual connection from the pins on the cartridge port to ground. Two of the lines connected from the PLA to the cartridge port will control memory configuration. The PLA will monitor the voltage level of these two lines. These two lines are called the EXROM line and the GAME line. These two lines are normally high (5v). When either (or both) of these lines are grounded that PLA will reconfigure the memory that the microprocessor sees.

Grounding only the EXROM line will cause the PLA to reconfigure memory so that the microprocessor will look to the cartridge port to find the memory from $8000-$9FFF. All of the other memory locations will remain intact. BASIC ROM, KERNAL ROM and the I/O devices will remain in effect. Under normal circumstances the EXROM line would be grounded only if a cartridge had been installed. If we were to ground the EXROM line without a cartridge installed the microprocessor would not find any memory at these locations ($8000-$9FFF). The PLA does not care if any memory exists at the memory locations that the microprocessor is looking at. If we ground the EXROM line without plugging in a cartridge, the PLA will prevent the microprocessor from seeing any memory other than what is at the cartridge port (nothing in this example). The microprocessor will only find random garbage in this area. This is a way for the PLA to prevent the microprocessor from seeing the RAM normally at $8000-$9FFF. REMEMBER THAT WHEN THE EXROM LINE IS GROUNDED THE PLA WILL CAUSE THE MICROPROCESSOR TO SEE ONLY THAT MEMORY THAT IS PLUGGED INTO THE CARTRIDGE PORT. THIS WILL OCCUR WHETHER THERE IS A CARTRIDGE PLUGGED IN OR NOT!

Grounding only the GAME line will cause the PLA to reconfigure memory so that the computer will be able to use cartridges designed for the 'ULTIMAX' system. The KERNAL ROM and the BASIC ROM will be switched out and the microprocessor will look to the cartridge port for memory in the $8000-$9FFF and the $E000-$FFFF range. This configuration of memory will cause the microprocessor not to see ANY memory in the following areas of memory; $1000-$7FFF and $A000-$CFFF ('images' may appear in these open areas). Memory locations $0000-$0FFF will appear as the normal RAM and $D000-$DFFF will appear as the normal I/O devices. The microprocessor will look for memory locations

$8000-$9FFF and $E000-$FFFF on the cartridge port. Again, this memory configuration is only for those cartridges that emulate the 'ULTIMAX' system.

Grounding BOTH the EXROM and the GAME lines at the same time will cause the PLA to reconfigure memory so that the microprocessor will look to the cartridge port for memory at locations $8000-$BFFF. This configuration will allow the use of 16K of continuous cartridge memory. 8K will reside in the normal area of cartridge memory ($8000-$9FFF). The other 8K will reside in the area of memory that is normally reserved by BASIC ($A000-$BFFF). This memory configuration will also allow for the programmer to switch between the RAM and ROM located at memory locations $8000-$9FFF. By controlling the LORAM line the programmer may select RAM or cartridge ROM. When the LORAM line is high the PLA will cause the microprocessor to see ROM at location $8000-$9FFF. When the LORAM line is low the PLA will cause the microprocessor to see RAM at locations $8000-$9FFF and the microprocessor will still see the cartridge ROM located at $A000-$BFFF.

We have now covered the major functions of the PLA and microprocessor combination used in the C-64 as they relate to memory management. The PLA also has a few other important functions. When the microprocessor writes to an area of memory that contains both RAM and ROM (BASIC ROM $A000-$BFFF, for example) the PLA will allow the microprocessor to write to the underlying RAM. The PLA will decode the microprocessor's instructions when it is reading and writing. The PLA will then 'decide' what memory that the microprocessor should have access to (RAM or ROM). If the microprocessor is going to write (STA) a value in memory, the PLA will select the appropriate memory (ROM can not be written to). If the microprocessor will be reading (LDA) a value from memory, the PLA will select the proper area of memory based upon the LORAM, HIRAM, EXROM and GAME lines. The one deviation from the preceding example is where the microprocessor writes to the memory at $D000-$DFFF. This memory normally contains the I/O devices, rather than RAM or ROM. Because of this, the PLA will allow the microprocessor to both read and write to these addresses. These address do not normally refer to actual RAM/ROM memory locations used by the 6510. They primarily contain the onboard registers of the I/O devices and the color RAM used by the VIC chip. The VIC (video) chip, the SID (sound) chip, the CIA's (communication) chips and the color RAM are located in this area of memory.

The VIC chip can also access (look at) memory. The VIC chip can only address 16K of memory at any one time. The VIC chip also causes the PLA to select what area of memory is available to the VIC chip. For instance, when the VIC chip wants to access the CHARACTER ROM, the PLA will select this chip rather than the I/O devices normally located from $D000-$DFFF. For our purposes we have covered all that is need to be said about the 6510 microprocessor and the PLA.

If you have a hard time digesting all the information presented to you in this chapter, DON'T WORRY ABOUT IT!!! A tremendous amount of information has been presented here. Let's just review a few of the more important concepts:

1). The 6510 microprocessor is RESET upon power up.

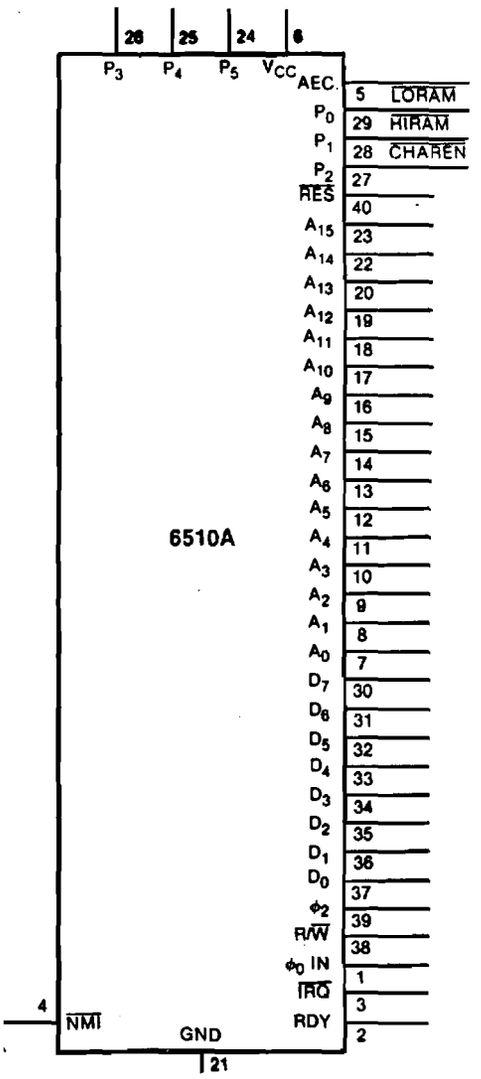
2). Whenever the microprocessor is RESET the LORAM, the HIRAM and the CHAREN lines will be set high.

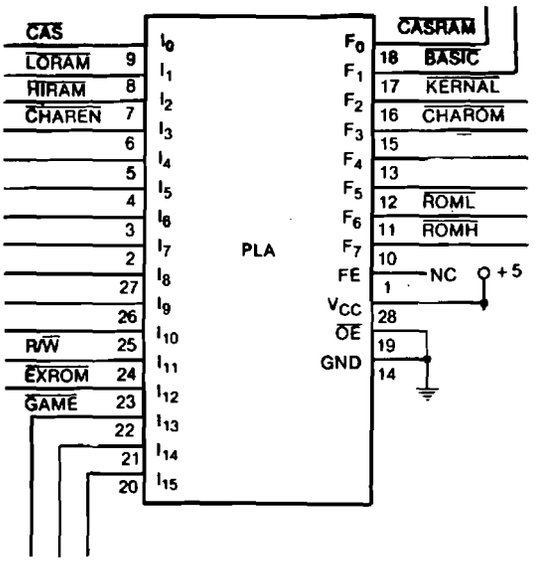
3). The PLA will control the microprocessor's access to various areas of memory.

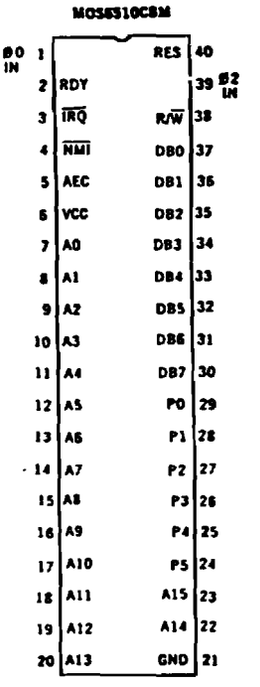
4). The PLA may be controlled by both hardware and software methods.

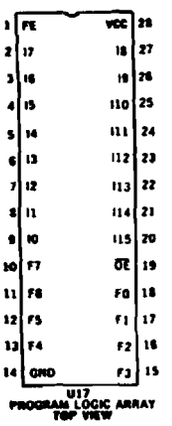
5). By grounding the EXROM line we can prevent the microprocessor from seeing RAM at locations $8000-$9FFF (very important).

6). A software RESET (SYS 64738 or JMP $FCE2) is different than a hardware RESET.


26
25
24
6
P3
P4
P5
VCC
AEC
5 LORAM
P0 29 HIRAM
P1 28 CHAREN
P2 27
RES 40
A15 23
A14 22
A13 20
A12 19
A11 18
A10 17
A9 16
A8 15
A7 14
A6 13
A5 12
A4 11
A3 10
A2 9
A1 8
A0 7
D7 30
D6 31
D5 32
D4 33
D3 34
D2 35
D1 36
D0 37
Φ2 39
R/W 38
Φ0 IN 1
IRQ 3
4 NMI
GND
RDY 2
6510A
21


CAS
LORAM 9
HIRAM 8
CHAREN 7
6
5
4
3
2
27
26
R/W 25
EXROM 24
GAME 23
22
21
20
I0
I1
I2
I3
I4
I5
I6
I7
I8
I9
I10
I11
I12
I13
I14
I15
PLA
F0 CASRAM
F1 18 BASIC
F2 17 KERNAL
F3 16 CHAROM
F4 15
F5 13
F6 12 ROML
F7 11 ROMH
FE 10
VCC 1 NC +5
OE 28
GND 19
14


MOS6510CBM
Φ0 IN 1
2 RDY
3 IRQ
4 NMI
5 AEC
6 VCC
7 A0
8 A1
9 A2
10 A3
11 A4
12 A5
13 A6
14 A7
15 A8
16 A9
17 A10
18 A11
19 A12
20 A13
RES 40
39 Φ2 IN
R/W 38
DB0 37
DB1 36
DB2 35
DB3 34
DB4 33
DB5 32
DB6 31
DB7 30
P0 29
P1 28
P2 27
P3 26
P4 25
P5 24
A15 23
A14 22
GND 21


1 FE
2 17
3 16
4 15
5 14
6 13
7 12
8 11
9 10
10 F7
11 F6
12 F5
13 F4
14 GND
VCC 28
18 27
19 26
110 25
111 24
112 23
113 22
114 21
115 20
OE 19
F0 18
F1 17
F2 16
F3 15
U17
PROGRAM LOGIC ARRAY
TOP VIEW

GCR RECORDING

When we read the ads for the latest copy programs we see phrases like copies half-tracks, sync to particular reference sectors, reproduce density/frequency alterations, non-standard sectors, singular sync, extra sectors, non-standard sync, gap bytes, bit or nybble copier, and so on. The days of the 'bad blocks' are quickly coming to an end. The stakes have been raised. In order to understand and deal with these newer protection schemes, you will have to understand the way the 1541 disk drive works on a more fundamental level. In this chapter the so-called GCR (Group Cyclic Recording or Group Code Recording) method of encoding data on the disk will be explained. An understanding of this little known subject is essential if you want to be able to use some of the tools provided with this book. With the tool-kit provided and an understanding of this chapter you will be able to look directly at the header information and even at the so called 'gap bytes' of any sector on the disk. You will be able to 'see' why you are getting an error 29, 23 etc when you try to read a particular block of information from the disk. Furthermore you will be able to see if any funny business has taken place. You can compare your copy disk to the original on a very fundamental level.

Before getting into a discussion of the GCR code we will first explain the purpose behind converting ASCII codes into this GCR code. The basic reason has to do with the way in which data, in the form of binary digits or 'bits', are actually written onto the disk.

If you were able to 'watch' the read/write head of your disk drive as it created a bit-pattern on the disk, you would find that, contrary to popular belief, the on/off or 1/0 pattern does not correspond to a magnetized/non-magnetized pattern of little magnetic domains on the disk. In other words 1/0 does not correspond with magnetized/non magnetized. Instead, a binary 1 corresponds to a change in magnetic state while a binary zero corresponds to no change. This concept is not easy to appreciate and yet, behind it is the reason for the GCR code. If we add one more concept namely that of the 'clock cycle', all this should become clear!

The disk drive uses an internal timer when data is being read from or written to the disk. The timer will keep track of how long a bit of data should be. The timer can operate at four different speeds (clock rates). The clock rate that the disk drive uses is dependent upon the track that is being read. Tracks 1-17 will use one clock rate, 18-24 will use another, 25-30 another and 31-35 still another clock rate. If the disk drive uses a higher clock rate, more bits per second will be written to the disk. Conversely, if the disk drive uses a lower clock rate, less bits per second will be written to the disk. The term clock rate may also be used synonymously with the term density.

The 1541 disk drive has an internal crystal-controlled clock which operates at the rate of 16 MHz (sixteen million cycles per second). If the disk drive 'clocked out' bits at that rate we could theoretically increase the number of bytes a disk could hold by a factor of four. Unfortunately, the data bits would be so close together on the disk at this clock rate, that a change in the magnetic state could not be accurately detected by the 1541. The internal clock frequency is divided down to provide a more reasonable timing rate (for the sake of simplicity, just remember that the clock rate of 16 Mhz is divided down). The actual timing relationship is fairly complicated and will not be discussed here in detail. The clock rate is first divided by a value of 13, 14, 15 or 16, depending upon which track the drive is trying to access. Tracks 1-17 will use a divisor of 13, tracks 18-24 will use 14, tracks 25-30 will use 15 and tracks 31-35 will use 16. The divisor that is used here determines the denstiy that the track will be written at. Each clock rate will again be divided, this time by a divisor or 4. We have now obtained the actual bit density that will be used for the track.

Let's try just one example of bit density. The internal clock runs at 16 Mhz. On the outer tracks the divisor used to determine density is 13. When 16 Mhz is divided by 13 you get 1.2307 Mhz. Dividing 1.2307 Mhz (1,230,700) by 4 equals 307,692 bits per second. The bit density of 307,692 bits/sec on tracks 1-17 is the highest used on the 1541. The number of bits/sec is higher on the outer tracks because the disk moves faster as we get farther from the center. Since the disk is moving faster, we can write bits at a faster rate without them getting too close together. On inner tracks like 31-35 the clock rate is only 250,000 bits/sec. This insures that the distance between successive changes in the magnetic state on the disk is large enough to prevent 'blurring'. The concept of 'density' is very important to understand. Some of the latest protection schemes play games with the density. This is discussed at greater length elsewhere in the book.

The Read/Write (R/W) head of the disk drive plays an important role in data storage. When the drive is writing data to the disk, the data will be converted from data bits into electrical impulses. The R/W head will convert these electrical impulses into magnetic areas on the surface of the disk. These areas on the disk will retain the magnetic information stored there. The R/W head is similar to an electro-magnetic (remember your high school science classes?). When the disk drive is reading information from the disk drive, the R/W head will convert the magnetic areas on the disk back into electrical impulses. These impulses will be interpreted by the logic board of the disk drive and converted back to data bits.

We can now get a more visual image of just how information is placed on the disk. Let's say that we wish to store the binary pattern of 11010011 onto the disk. During the first clock cycle the write circuitry will reverse the flow of the electrical current to the R/W head. The change in current will produce a reversal in the magnetic field produced by the R/W head. This reversal of the magnetic field corresponds to the first binary 1 in our number. Now, during the second clock cycle, the electrical current to the read/write head will be reversed again. This change in current would induce another reversal in magnetic field on the disk. The change in magnetic field on the disk corresponds to the second binary 1 (remember, a CHANGE in magnetic zone corresponds to a 1). During the third clock cycle the read/write head current and head magnetization would remain unchanged. Since the magnetic state did not change, this will be interpreted as a binary 0 when read. Then in the fourth clock cycle the currrent/magnetization will change again to represent another 1 bit. This process will continue until all the data has been written. Remember that a 1 bit written to the disk will produce a CHANGE in the magnetic zone on the disk, and a 0 bit will NOT produce a change.

During the read mode of the disk drive the same basic procedure applies. The R/W head will convert changes of magnetic zone on the disk into electrical impulses. These electrical impulses will then be converted into binary digits (bits of data) by the logic board. When the disk drive is reading data, the clock will be used to time the gaps between changes in magnetic zones on the disk, using the appropriate clock rate. If a change in the magnetic zone occurs during the time specified by the clock, a binary 1 will be registered. If time runs out before a change in the magnetic zone is detected, a binary 0 will result.

Now we can get to the reason for the existance of the GCR code. Remember, a binary 0 corresponds to no signal coming from the read/write head during a given clock cycle. Suppose that a disk is being read on a drive that runs slightly slower (or faster) than the one on which it was created. If the information being read happened to contain many binary zeros in a row e.g. 10000001, the information could easily be misinterpreted. With the drive speed a little slow, it might appear to the read/write head that no change in the magnetic zones had taken place in 7 clock cycles, although in the example above there are only 6 zeros in succession. At a slightly fast drive speed the change could come early, and thus the read/write head will only see five zeros non-change intervals. Thus it is very important that we never have very many zeros in succession. Unfortunately much data saved to the disk drive does have many zeros in succession. Enter the GCR coding system.

By now you should be somewhat familiar with the ASCII code. When you use your machine language monitor to examine the memory of the computer you see pairs of hexadecimal digits. Each hex digit is called a nybble. You will also notice, when you look at the computer's memory, that there are often places

in which there are many zeros in succession. A hex zero nybble ($0) translates into %0000 in binary. We have discussed the reasons why several zeros in succession can make it difficult or impossible to interpret the data correctly when read from the disk. What happens in the GCR is similar to a direct substitution code like ASCII, except that we increase the length of the 'message'. A hex byte (8 bits) is broken down into its individual nybbles (4 bits each). Each nybble (4 bits) is directly replaced by a 5-BIT GCR code. To convert from hex to GCR (or vice versa) all one has to do is look in the GCR table below and substitute one value for the other. Of course the GCR code will be 20% larger than its hex counterpart, since 4 bits are replaced by 5. Following is a table of GCR codes:

Table GCR-1: GCR Code

HEX	Binary	GCR
0	0000	01010
1	0001	01011
2	0010	10010
3	0011	10011
4	0100	01110
5	0101	01111
6	0110	10110
7	0111	10111
8	1000	01001
9	1001	11001
A	1010	11010
B	1011	11011
C	1100	01101
D	1101	11101
E	1110	11110
F	1111	10101

An examination of the GCR codes reveals that no combination of two GCR nybbles will ever yield more than two 0 bits in succession. This insures accuracy in interpreting the bit pattern even when drive speeds are not perfectly matched. Second, no two successive GCR nybbles will ever generate a pattern containing more than eight 1 bits in a row. This is also important since the DOS assigns a special significance to a pattern of 10 or more 1's in a row. This is the so-called 'sync mark' which will be discussed in more detail later.

Since hex digits are converted to GCR when data is written to the disk, this makes it difficult to interpret 'raw' data from the disk. The tool-kit provided with this book has a program (GCR READ) which allows you to find a sync mark and then read 1000 GCR bytes into the buffer in the disk drive. This data can then be transferred to the computer's memory, where you can examine it with your machine language monitor. (Note: If you have a copy of DI-SECTOR's DRVMON, you can 'look' directly into the disk drive's memory and avoid the time spent transferring

data to and from the 1541. You can also disassemble and modify routines in the disk drive.) If you use the interpret mode of the monitor you will find that the ASCII representation of the GCR data looks like garbage. In order to make sense out of this information you will have to convert it from GCR to binary form. Following is an example using one of the headers from track 35. However, in order to make sense out of this information we will digress slightly into a review of the structure of the 'header' of a data block.

A sector on a disk actually contains quite a bit more information than is revealed by a track and sector editor. First comes a sync mark. Remember, a sync mark consists of 10 or more (usually 40) 1 bits in succession. The sync mark is a unique 'finger print' which allows the drive to find a reference point. After the sync mark comes the header identifier byte. This is the hex value $08 and tells the drive that the information following is header information as opposed to data. Now comes the header checksum byte. This byte is obtained by exclusive-oring (EOR) the track number, the sector number and the two ID bytes (ID1 and ID2). The DOS always calculates the checksum expected and compares it to the header checksum found on the disk. If a difference is found an error 27 is signalled. After the header block checksum on the disk comes the sector number, the track number, and then the two ID bytes in reverse order (ID2, ID1). Note that these four bytes are shown in the wrong order in the diagram in the back of the 1541 USER'S GUIDE. Following the ID bytes come two $0F bytes (not shown in the USER'S GUIDE) and then the header gap, which consists of eight $55 bytes. The header gap bytes simply provide padding which allows the drive time to recover before encountering the data block.

Immediately after these header gap bytes comes another sync mark and then the data block identifier byte, which is a hex $07. This identifier ($07) is necessary so the DOS can tell the difference between the header block and the data block. Following the $07 are the 256 bytes of data you normally see with a track and sector editor. Finally there is a data block checksum byte whose value lets DOS know if some of the data has been corrupted. Again this checksum is obtained from EOR'ing the 256 data bytes together. If the checksum found after the data block does not match the calculated value, an error 23 is generated. (Note: When an error 23 is signalled, the data has already been read into disk drive memory and can at least be partially recovered. This is not true of a header checksum error, error 27. However, using the tools provided with this book you could read the raw data after the error 27 and recover it!). After the data checksum comes two $00 bytes for padding and then an intersector (tail) gap of variable length before the next sector's sync mark.

You may be wondering why the information in table GCR-2 does not seem to conform to this pattern. Where are the $08 and $07?? Why do there seem to be 10 header gap bytes? The answer is that we are looking at a stream of 10-bit GCR bytes which are being displayed in 8-bit segments. Kind of confusing? Remember, the raw data that is read back in from the disk is in GCR coding. GCR coding implies ten bits per byte. When we view these data bits from the ML monitor we only see 8 bits at a time, so the extra bits from one GCR byte get tacked on to the beginning of the next byte.

We will now decode the header information in line 5000 in table GCR-2. We must first convert the 8-bit hex digits into binary to get the actual bit stream. Then we will regroup the bits into groups of 5 and match these 5-bit GCR nybbles with the GCR code in table GCR-1

Starting with EA 52 67 A5 36 53 77 5E 95 55 etc. we get:

```
  E   A     5   2     6   7     A   5     3   6
1110 1010 0101 0010 0110 0111 1010 0101 0011 0110
          ------------

  5   3     7   7     5   E     9   5     5   5
0101 0011 0111 0111 0101 1110 1001 0101 0101 0101
```

We must now regroup these 4-bit nybbles into 5-bit GCR nybbles. There is, however, one problem we must resolve before we can do this. The routine which reads the GCR code from the disk will also try to read the sync marks from the disk. The hardware of the 1541 disk drive will not allow us to directly read sync marks from the disk. When a sync mark is encountered on the disk the bits read will be unreliable until the sync mark has passed. The byte read from the disk will appear to be somewhat random. For this reason we should consider 'throwing out' the byte that represents the sync mark.

But how do we know which byte represents the sync mark? The problem is easily solved, since we know that a header identifier byte follows the sync. We must find an $08, which translates into 10 bit code (using the GCR values for 0 and 8) as: 01010 01001 since 0=01010 and 8=01001. A careful examination of the binary pattern above will find that this pattern exists starting with the 9th bit. This is marked with underlining. In essence if we just 'throw out' the byte that represents the sync mark we may directly interpret the GCR code. In our example the GCR code for the header begins at the 9th bit. Starting here we must now translate each group of 5 GCR bits back into a 4-bit hex digit. If we group the above pattern of binary digits in groups of five, starting with the ninth bit, we get:

```
01010 01001 10011 11010 01010 01101 10010 10011

   0   8       3   A       0   C       2   3

01110 11101 01111 01001 01010 10101 01010 10101
   4   D       5   8       0   F       0   F
```

Looking these values up in the GCR table reveals the true identity of this block. We get:

0 8 3 A 0 C 2 3 4 D 5 8 0 F 0 F

Let's interpret these eight values: 1) $08 is the header block ID; 2) $3A is the header block checksum; 3) $0C is the sector number, in this case sector 12 since $0C = 12 decimal; 4) $23 is the track number, in this case 35 since $23 = 35 decimal; 5) $4D is ID2, which is an ASCII 'M'; 6) $58 is ID1, which is 'X'; and 7-8) Finally we have our two 0F 'padding bytes'. We have decoded the GCR header!!

You will notice that in the original GCR data in figure GCR-2 there are ten 55 bytes following the 95 byte. These are not all to be interpreted as 'header gap' bytes. If you look closely, you will see that we have 'used up' two of the 55's which followed the 95 in order to get our two 0F bytes. This leaves exactly 8 55's left to account for. These are the gap bytes we're expecting.

Note: <u>DOS does not translate either sync bytes or gap bytes when they are written to the disk.</u> Thus, sync bytes are actually long strings of binary 1's and gap bytes remain $55 (binary %01010101). During the read mode sync bytes are not shown directly, although we will see a few remnants where the sync bytes should be.

Let's proceed to line 5010 in the example. In the middle of the line we see an FF then a 55 D4 A5 29 etc. The FF corresponds to our sync mark for the data block. It is in this area that we should be able to find the data block ID ($07). Let's see if the decoding process makes sense.

Write out the 8-bit binary for these hex codes.

```
  F   F     5   5     D   4     A   5     2   9
1111 1111 0101 0101 1101 0100 1010 0101 0010 1001
```

We are looking for the ten bit pattern for 07 which is 01010 10111. This is found immediately after the 8 binary 1's (which are remnants of the sync mark). Remembering to disregard the remnants of the sync mark, we should now group the data into groups of 5 bits each, beginning with 01010 10111. We get:

```
01010 10111 01010 01010 01010 01010 01010 01010
   0   7       0   0       0   0       0   0
```

Look up the 5-bit GCR nybbles in table GCR-1. We find that the first two nybbles are 07 as expected. This is followed by 00 00 00. These 00 bytes are the first few data bytes. Evidently this data block has been filled with all 00's. You can use the routine you have been provided with to read a sector containing data. It would be a good exercise to then decode at least the header and part of the data block for practice with GCR.

Now look at line 5158 in table GCR-2. We see 4A 55 55 55 55 55 55 FF. These 55's are the so-called intersector gap bytes (tail gap). They give DOS some breathing room between sectors. The number of these gap bytes varies from track to track and in fact is not a constant for a given track. Drive speed can affect how many intersector gap bytes are written during formatting. The DOS actually calculates how many bytes can be written to a given track. (This value varies with drive speed.) It then determines how many byte are required on that track (based on the number of sectors) and figures how many bytes are left over to use in the tail gaps. The tail gap bytes provide an interesting and subtle place to hide a value for protection purposes. Supposedly the new bit copiers can duplicate these bytes, though.

After the 55's we see an FF. This is the remnant of the sync mark which denotes the beginning of the next header block. See if you can decode the header beginning on line 5140. You can compare your translation to the one below:

Raw hex bytes: 52 67 B5 76 53 77 5E 95 55

```
  5   2     6   7     B   5     7   6     5   3
0101 0010 0110 0111 1011 0101 0111 0110 0101 0011

  7   7     5   E     9   5     5   5
0111 0111 0101 1110 1001 0101 0101 0101
```

Now group these 4-bit hex nybbles into 5-bit GCR nybbles.

```
01010 01001 10011 11011 01010 11101 10010 10011
   0   8       3   B       0   D       2   3

01110 11101 01111 01001 01010 10101 ...
   4   D       5   8       0   F
```

Now translate via the GCR-HEX table. You will find that you get:

08 3B 0D 23 4D 58 0F...

This makes complete sense. We have our header ID (08) followed by a new checksum (3B). We then find the sector and track numbers (0D and 23) which translate into decimal as sector 13 and track 35. This makes sense since our last block was track 35 sector 12. We then find ID2 and ID1 to be identical to that found in the last block (4D and 58). Finally we have our padding bytes (0F's).

The value of the programs provided can now be appreciated. You are no longer limited to looking at the data block with a track and sector editor. You can now look at 1000 bytes of raw data. Nothing can be hidden from your view. You see the gap bytes. You can compare your copy disk against the original right down to the last bit!! Any difference may be used to distinguish your copy from the original by the protection scheme.

If you wish to investigate a track at the GCR level you should run the program 'GCR READ' from the program disk. You will be prompted for a track number. The routine will sync-up (meaning line-up) on the first sync mark it comes to. It will then read the next 1024 bytes into the disk drive's memory starting at $0400 and ending at $07FF. This memory can be transferred to the computer's memory and then dumped to the printer in hex/ASCII by your ML monitor. You will end up with a listing similar to that shown in table GCR-2. You can then translate the headers, looking for any funny business. Since the sector you sync-up on is random, you will have to repeat this process discarding the duplicates until all sectors have been read in and printed. You will then be able to go to work determining what exactly is going on in the area of copy protection.

In the older protection schemes you will find only errors in the header:

Error 20 indicates that the header cannot be found, i.e., the $08 identifier is missing.

Error 21 indicates that no sync mark was found within a certain time (20 ms).

Error 22 indicates data block not present (i.e. no $07 was found).

Error 23 indicates a checksum error in the data block. (Note: If you need to put an error 23 on a copy disk to make it work, you may need to use the same checksum as that on the original. It is possible for the protection scheme to verify not just an error 23, but an error 23 with a particular checksum. Also, don't forget that the scheme may also use the data in the block.

Error 27 indicates an error in the header checksum. Again, a particular checksum value may be required and the other header information is potentially usable.

Error 29 indicates a disk ID mismatch. This means that the two IDs found in the particular sector do not match the ID's that exist at track 18 sector 0.

Artificially inducing errors 20, 21, 22, 23, 27 and 29 and then checking for them is the basis of what we have been calling the 'old style' of protection. Two problems exist with these types of protection methods:

1). Reading most of these errors causes the 1541's stepper motor to beat up against its end stop. As we all know, the beating of the drive takes during the 'bump' is largely responsible for the many drives which are going out of alignment. This method of protection is unacceptable for this reason.

2). This method of protection no longer really provides any real protection. There are many copy utilities, both commercial and public domain, which can be used to put these errors onto a copy disk. There are even programs like Omni Clone which automatically reproduce these errors as they copy. The proliferation of these copy programs makes 'bad blocks' an ineffective method of protection.

The latest protection methods are far more sophisticated and often deal with extra sectors, displaced sectors, extra gap bytes, displaced track numbers, changes in density and other ways of creating disks which contain a unique pattern that can be tested for but not easily reproduced. Let's look at a few specifics on these methods

DISPLACED SECTORS: Also known as non-standard sectors. You may discover, as you investigate the header information of a particular track, that the sectors are not in proper order or that some sectors have been duplicated. In our example above we found that the first sector was sector 12 of track 35. The sector following was sector 13 of track 35. On a normally formatted track, the sectors are in order from 0 to the maximum for that track, so any deviation from this is abnormal. Displaced sectors may be used as a method of copy protection by having the program check for the displaced sectors.

EXTRA SECTORS: Tracks 18-24 normally contain 19 sectors numbered from 0 to 18. If you find a sector numbered 19 on tracks in this range, you have discovered that the protection scheme involves extra sectors. If the number of gap bytes are reduced it is possible to put an extra sector on these tracks only.

DISPLACED TRACK NUMBERS: Sophisticated protection schemes may involve formatting a track with sectors whose headers contain an incorrect track number. Electronic Arts has done this on track 35. If you run your GCR reading program on track 35 of an unbroken EA disk you will find that the track numbers are 34!

GAP BYTES: On a normally-formatted track there are eight (8) gap bytes ($55) separating the header from the data block. It is possible using a modified drive or special software to change this number or to use a character other than $55. This condition is not easily created and forms the basis for newer forms of protection.

The above methods just scratch the surface of the newer protection schemes and will be dealt with at greater length elsewhere in the book. There is, however, another method of protection which will be discussed at length in this chapter. This method is perhaps the most difficult to reproduce with a copy program. It involves altered bit densities.

On the earlier pages of this chapter we introduced the concept of 'bit density' or the number of bits/second which are clocked out on a particular track. It was explained there that the number of bits/sec is greatest on the outer tracks (1-17) where the surface of the disk is moving at the greatest speed. Points farther from the center move on circles of greater circumference. Thus points farther from the center move a greater distance in the same time. (By the way, one revolution takes 1/5 of a second = 200 ms.)

Bits are clocked out at four different speeds depending on which track is being written to. Below is a table of clock rates:

Tracks	Clock Rate	Divisor	Bits
1-17	307,692 b/s	13	11
18-24	285,714 b/s	14	10
25-30	266,667 b/s	15	01
31-40	250,000 b/s	16	00

The various rates are achieved by dividing the 16 MHz clock rate by 13, 14, 15 and 16 respectively. Then the value obtained is divided by 4. This, in turn, is the actual clock rate used. The number of bits per inch varies from track to track with the largest number on the inner tracks since their circumferences are very small. The reason the clock rate must be slowed from its 16 MHz frequency is so that the bits don't get too close together. A blurring effect takes place if the magnetic domains are too close to each other.

Density may be changed simply by changing setting (1) or clearing (0) two bits at memory location $1C00 in the disk drive. Bits 5 and 6 of memory location $1C00 (which is part of the 6522 Disk Controller VIA) control the density with which bits are read and written. The four possible states of these two bits (00,01,10,11) correspond to the four densities as shown above. You will find as you read further into this book that $1C00 is an extremely important location in the disk drive. The bit pattern at that location not only controls the density but also controls the turning-on of the motor and the cycling of the read head in half-track increments. Methods of moving the head in half-track increments are discussed elsewhere.

Under normal circumstances the DOS will select the proper values for bits 5 and 6 from a look-up table to ensure that data is written at the proper density. The bit pattern for tracks 1-17 is 11, tracks 18-24 is 10, tracks 19-30 is 01 and tracks 31-35 use 00. What if we decide to take control of $1C00 and write data to track 1 using a different density? Since this is normally written at the highest density we would be writing less bits per second than normal. The distance between 'bits' would be greater. If a copy program is unaware that the track is written at a nonstandard (wrong) density, it will not be reading the track at the right density. i.e. the clock will still be clocking bits at 307,692 bits/second. What will happen? It should be clear from our earlier discussion on how information is recorded on the disk that reading at the wrong clock rate will simply makes the information appear to be garbage. There is no way the copy program can 'know' the clock rate that was used to create the information on the track. It will read it at the wrong density and write a lot of garbage on the copy disk. The copy disk will not pass the copy protection scheme under these circumstances.

The problem can be compounded greatly by using all four densities on one track!! Now the copy program must figure out WHERE the density changes occur in order to duplicate the track. If the information on the track follows the normal format, i.e., a sync mark followed by an $08, followed by a header etc., the copy program could conceivably figure out where the density changes take place by trial and error, knowing what it should find. If, however, the information on the track is simply a pattern of bytes known only to the creators of the protection scheme, it is going to be very difficult (impossible?) for a copy program to figure out exactly where the density changes occur. It seems that a foolproof protection scheme can be created which finds a single sync mark and then reads a fixed number of bytes. It then changes the density and reads another fixed number of bytes. It could keep doing this, changing the density many times. The protection scheme knows where the density changes took place and so it can read back a predictable pattern of bytes. There is virtually no way a copy program can duplicate this disk since there is very little chance of finding where the density has been changed by trying to read the disk.

You have tools which allow you to examine a track on the bit level. They are not powerful enough to account for density changes. Here is an important point: Once the protection schemes become sophisticated enough, copy programs will no longer work. The only way left to get a copy of the program will be to 'break' the protection scheme, i.e., alter the program so that it runs without examining the disk for special 'finger prints'. If the protection check is made only once at the start, as is common now, it is also possible to lift a working copy of the program from the memory of the computer after it has passed its protection test. Other chapters in the book deal with these topics.

A FEW SELECTED DISK DRIVE MEMORY LOCATIONS

DEC	HEX	FUNCTION
000	$00	COMMAND CODE FOR BUFFER 0
001	$01	COMMAND CODE FOR BUFFER 1
002	$02	COMMAND CODE FOR BUFFER 2
003	$03	COMMAND CODE FOR BUFFER 3
004	$04	COMMAND CODE FOR BUFFER 4
006	$06-$07	TRACK AND SECTOR FOR BUFFER 0
008	$08-$09	TRACK AND SECTOR FOR BUFFER 1
010	$0A-$0B	TRACK AND SECTOR FOR BUFFER 2
012	$0C-$0D	TRACK AND SECTOR FOR BUFFER 3
014	$0E-$0F	TRACK AND SECTOR FOR BUFFER 4
018	$12-$13	ID FOR DRIVE 0
020	$14-$15	ID FOR DRIVE 1 UNUSED
022	$16-$17	ID FROM HEADER
024	$18-$19	TRACK AND SECTOR
026	$1A	CHECKSUM
048	$30-$31	BUFFER POINTER FOR DISK CONTROLLER
057	$39	CONSTANT 8, MARK FOR BEGINNING OF DATA BLOCK HEADER
058	$3A	PARITY FOR DATA BUFFER
061	$3D	DRIVE NUMBER FOR DISK CONTROLLER
063	$3F	BUFFER NUMBER FOR DISK CONTROLLER
067	$43	NUMBER OF SECTORS PER TRACK FOR FORMATTING
071	$47	CONSTANT 7, MARK FOR BEGINNING OF DATA BLOCK HEADER
074	$4A	STEP COUNTER FOR HEAD TRANSPORT
081	$51	ACTUAL TRACK NUMBER FOR FORMATTING

TABLE GCR-2

```
5000 EA 52 67 A5 36 53 77 5E  ■Rg↓6Sw^
5008 55 55 55 55 55 55 55 55  .UUUUUUU
5010 55 55 55 FF 55 D4 A5 29  UUUU¢↓)
5018 4A 52 94 A5 29 4A 52 94  JR.↓)JR.
5020 A5 29 4A 52 94 A5 29 4A  ↓)JR.↓)J
5028 52 94 A5 29 4A 52 94 A5  R.↓)JR.↓
5030 29 4A 52 94 A5 29 4A 52  )JR.↓)JR
5038 94 A5 29 4A 52 94 A5 29  .↓)JR.↓)
5040 4A 52 94 A5 29 4A 52 94  JR.↓)JR.
5048 A5 29 4A 52 94 A5 29 4A  ↓)JR.↓)J
5050 52 94 A5 29 4A 52 94 A5  R.↓)JR.↓
5058 29 4A 52 94 A5 29 4A 52  )JR.↓)JR
5060 94 A5 29 4A 52 94 A5 29  .↓)JR.↓)
5068 4A 52 94 A5 29 4A 52 94  JR.↓)JR.
5070 A5 29 4A 52 94 A5 29 4A  ↓)JR.↓)J
5078 52 94 A5 29 4A 52 94 A5  R.↓)JR.↓
5080 29 4A 52 94 A5 29 4A 52  )JR.↓)JR
5088 94 A5 29 4A 52 94 A5 29  .↓)JR.↓)
5090 4A 52 94 A5 29 4A 52 94  JR.↓)JR.
5098 A5 29 4A 52 94 A5 29 4A  ↓)JR.↓)J
50A0 52 94 A5 29 4A 52 94 A5  R.↓)JR.↓
50A8 29 4A 52 94 A5 29 4A 52  )JR.↓)JR
50B0 94 A5 29 4A 52 94 A5 29  .↓)JR.↓)
50B8 4A 52 94 A5 29 4A 52 94  JR.↓)JR.
50C0 A5 29 4A 52 94 A5 29 4A  ↓)JR.↓)J
50C8 52 94 A5 29 4A 52 94 A5  R.↓)JR.↓
50D0 29 4A 52 94 A5 29 4A 52  )JR.↓)JR
50D8 94 A5 29 4A 52 94 A5 29  .↓)JR.↓)
50E0 4A 52 94 A5 29 4A 52 94  JR.↓)JR.
50E8 A5 29 4A 52 94 A5 29 4A  ↓)JR.↓)J
50F0 52 94 A5 29 4A 52 94 A5  R.↓)JR.↓
50F8 29 4A 52 94 A5 29 4A 52  )JR.↓)JR
5100 94 A5 29 4A 52 94 A5 29  .↓)JR.↓)
5108 4A 52 94 A5 29 4A 52 94  JR.↓)JR.
5110 A5 29 4A 52 94 A5 29 4A  ↓)JR.↓)J
5118 52 94 A5 29 4A 52 94 A5  R.↓)JR.↓
5120 29 4A 52 94 A5 29 4A 52  )JR.↓)JR
5128 94 A5 29 4A 52 94 A5 29  .↓)JR.↓)
5130 4A 52 94 A5 29 4A 52 94  JR.↓)JR.
5138 A5 29 4A 52 94 A5 29 4A  ↓)JR.↓)J
5140 52 94 A5 29 4A 52 94 A5  R.↓)JR.↓
5148 29 4A 52 94 A5 29 4A 52  )JR.↓)JR
5150 94 A5 29 4A 52 94 A5 29  .↓)JR.↓)
5158 4A 55 55 55 55 55 55 FF  JUUUUUU
5160 52 67 B5 76 53 77 5E 95  Rg╒vSw^.
5168 55 55 55 55 55 55 55 55  UUUUUUUU
5170 55 55 FF 55 D4 A5 29 4A  UUU¢↓)J
5178 52 94 A5 29 4A 52 94 A5  R.↓)JR.↓
5180 29 4A 52 94 A5 29 4A 52  )JR.↓)JR
5188 94 A5 29 4A 52 94 A5 29  .↓)JR.↓)
5190 4A 52 94 A5 29 4A 52 94  JR.↓)JR.
5198 A5 29 4A 52 94 A5 29 4A  ↓)JR.↓)J
51A0 52 94 A5 29 4A 52 94 A5  R.↓)JR.↓
51A8 29 4A 52 94 A5 29 4A 52  )JR.↓)JR
51B0 94 A5 29 4A 52 94 A5 29  .↓)JR.↓)
51B8 4A 52 94 A5 29 4A 52 94  JR.↓)JR.
```

Sync and header
gap bytes
gap bytes, data sync and data block
data block

end of data block
tail gap bytes and header sync
header block
gap bytes
gap bytes, data sync and data block
data block

In the last chapter the GCR coding was explained. You have now used the program 'GCR READ' in conjunction with your monitor to read 'raw' information from a disk. In this chapter we will explain how the program works and how you can modify it to further explore the workings of the 1541 disk drive. One important item should be mentioned here: in the next chapter you will be introduced to a more powerful and easier to use tool. This tool is called DRVMON64 by STARPOINT SOFTWARE (the DISECTOR people). If you have this program you will find the next chapter more enlightening.

As you can see from the program listing below, 'GCR READ' prompts you for the track number (1-40). It is not a good idea to try to go past track 40 or even 38 since part of the reading head may ride up on a screw head inside the disk drive and jam the reading head. If you suspect that this has happened, you will have to take the plastic cover off the drive, physically move the head backwards a bit and initialize the drive (or try a direct BUMP as explained below - ouch!). Next you are asked how many blocks of GCR data you wish to transfer (1 thru 4 blocks) to the computer. The machine code which is loaded in the disk drive always reads four blocks of GCR code into the disk drive RAM (from $0400 to $07FF). But since it takes a fair amount of time to transfer the data to the C-64 so you can view it, you may want to respond with a '1' here to avoid a long wait.

Now the program goes to work. First we open the command channel and initialize the drive (line 50). We then read in the machine language from DATA statements and write the DOS routine into disk drive memory starting at $0300. (The proper syntax of the M-W (MEMORY-WRITE) command is as follows: PRINT#15, 'M-W' CHR$(L) CHR$(H) CHR$(N) CHR$(V) - here L and H are the LO and HI bytes respectively of the address in the disk drive memory we wish to write to. N is the number of bytes, in our case 1. V is the decimal value we wish to write to that address.) That is the purpose of lines 100-120.

Now we get to the interesting part. Before explaining what is going on at lines 160-180 we will have to explain some of the details of the operation of the 1541 disk drive.

A memory map of the disk drive would show that zero-page addresses are very important to its operation, just as they are in the C-64. Address $0000 to $0005 are used to queue up (line up) jobs in the disk drive. The interrupt cycle in the disk drive (or rather the disk controller) scans these memory locations. If it finds that one of them contains a special value (discussed below) things begin to happen.

Location $0000 uses buffer #0 located at $0300 for data storage and uses locations $0006 and $0007 for its track and sector numbers respectively. Similarly $0001 uses buffer #1 at $0400 for data and uses $0008 and $0009 for its track and sector. Likewise for $0002 - $0004. For our present experiments we will use only buffer #0. Thus $0000 will contain our 'job code' and $0006 and $0007 will contain our track and sector references. Below is a table of the valid job codes:

JOB CODE	DESCRIPTION
$80	READ
$90	WRITE
$A0	VERIFY
$B0	SEEK
$C0	BUMP
$D0	JUMP
$E0	EXECUTE

When the disk controller finds one of these codes in $0000 (or $0001 - $0004) it will look to $0006 and $0007 for track and sector references (if needed) and then carry out the job request. In our program you can see that we are putting the track input at line 10 into location $0006. Then at line 180 we put a $E0 (224) into address $0000. The $E0 is a very powerful instruction which causes a 'job queue execute' of the ML program stored in buffer #0 (at $0300). What happens is this: The disk is brought up to speed, then the head is moved to the track specified. The drive takes care of all the complications entailed in moving the head to that position, e.g. accelerating and decelerating the head, calculating the number of half steps etc. Once the read/write head has reached the specified track, the ML code in buffer #0, location $0300 is executed (a disassembly of that code follows the BASIC program listing). The code we have placed there causes the drive to: 1) find a sync mark and 2) read 1K (4 256-byte blocks) of GCR data into the disk drive RAM starting at $0400. The routine will end through a RTS instruction. Thus, when the routine in buffer #0 has finished, the disk drive memory contains 1K of GCR code. Lines 250-310 transfer 1,2,3 or 4 blocks of that code via the M-R instruction to address $5000 in the C-64 memory so that we can examine it with a machine language monitor.

One other bit of useful information is transferred. Lines 200-220 transfer a copy of address $0000 from the disk drive and display it on the screen calling it 'JOB STATUS' (error code). The job queue execute command will leave a value at address $0000 when it is finished, telling us if any errors have occurred. These values are catalogued below:

ERROR CODE	MEANING
$01	O.K. (no error)
$02	Header block not found (20)
$03	No sync character found (21)
$04	Data block not present (22)
$05	Checksum error in data block (23)
$07	Write verify error (25)
$08	Write protect on (26)
$09	Checksum error in hdr (27)
$0B	Disk ID mismatch (29)

Note that if you add 18 to the decimal value of the error code you will get the regular disk error number. The method employed by our DOS routine will normally leave a $04 as the job status. This is acceptable due to the way that internal DOS of 1541 handles error reporting. If you try to read information from a track past track 35, you may find a $03 as the job status. This means that no sync mark was detected. This is because those tracks are not normally formatted and do not normally have sync marks. If you get a $04 or $00 as the job status when reading past track 35, you have discovered that there is at least a sync mark on the track in question and maybe data. An examination of the contents of memory from $5000 to $5400 in the computer will reveal the nature of the information contained on the track in question. Since 'normal' disks contain no information beyond track 35, you may find that some of the newer protection schemes are putting data on track 36 or 37. Some of the newer copy programs (DiskMaker etc.) will copy the data contained on tracks beyond 35.

GCR READ is a versatile program in that you can easily modify it to do other things by changing the 224 in line 180 to a different job code. Using 176 ($B0), for example, will do a sector seek (fill in the sector number at the end of line 170)

If you get a memory map of the 1541 (Try 'The Anatomy of the 1541' from ABACUS or 'Inside Commodore DOS' from Datamost) you can learn to write your own DOS routines. You can test them by putting your code into the DATA statements from 320-460 and then changing the ending value of the loop in line 90 from 71 to the appropriate value.

Warning: When you are using job queue commands, you are controlling the disk drive at a very fundamental level. At this level the drive obeys your commands without doing much (if any) error checking. You can easily destroy a disk, even one with a write protect tab in place!! When you are testing your routines, always use an old disk or one for which you have a copy.

The 1541 is a complicated and powerful piece of hardware. We hope that this brief look at some of its workings has been enlightening. The newest generation of copy protection schemes are all going to be based in complex uses of the power of the 1541. The action will be taking place inside the disk drive where special routines in disk drive RAM are being executed to make the drive move in half-tracks, or beyond track 35, or to extra sectors, or at the wrong density..... This is where program protection is headed.

```
10 INPUT "[CH]INPUT TRACK NO.";T
20 IF T<OORT>40THEN10
30 INPUT "NUMBER BLOCKS TO TRANSFER.";B
40 IF B<1ORB>4THEN30
50 OPEN15,8,15,"I"
60 :
70 REM READ ML CODE AND WRITE TO $0300 IN 1541
80 :
90 FOR I=0 TO 71
100 READ V
110 PRINT#15,"M-W"CHR$(I)CHR$(3)CHR$(1)CHR$(V)
120 NEXT I
130 :
140 REM JOB QUE EXECUTE
150 :
160 PRINT#15,"M-W"CHR$(6)CHR$(0)CHR$(1)CHR$(T)
170 PRINT#15,"M-W"CHR$(7)CHR$(0)CHR$(1)CHR$(0)
180 PRINT#15,"M-W"CHR$(0)CHR$(0)CHR$(1)CHR$(224)
190 FOR I=1TO2000:NEXT:REM WAIT
200 PRINT#15,"M-R"CHR$(0)CHR$(0)
210 GET#15,A$:A=ASC(A$+CHR$(0))
220 PRINT"[CD][CD]JOB STATUS ($00):";A
230 PRINT "[CD][CD]PLEASE WAIT-TRANSFERRING DATA FROM 1541 TO $5000"
240 REM DUMP CONTENTS OF $0400 TO $07FF TO $5000
250 FOR I=4*256TO(4+B)*256-1
260 H=INT(I/256):REM HIGH ORDER BYTE
270 L=I-H*256   :REM LOW  ORDER BYTE
280 PRINT#15,"M-R"CHR$(L)CHR$(H)
290 GET#15,A$:A=ASC(A$+CHR$(0))
300 POKE(19456+I),A:REM POKE $5000 IN C-64
310 NEXT I
315 REM SYS 49152
320 DATA 169,0,170,169,0
330 DATA 133,48,169,4,133
340 DATA 49,169,3,72,169
350 DATA 20,72,76,86,245
360 DATA 80,254,184,173,1
370 DATA 28,145,48,200,208
380 DATA 245,230,49,232,224
390 DATA 4,208,238,160,186
400 DATA 80,254,184,173,1
410 DATA 28,153,0,1,200
420 DATA 208,244,32,224,248
430 DATA 165,56,197,71,240
440 DATA 5,169,4,76,105
450 DATA 249,32,233,245,76
460 DATA 254,244
```

CONVERSION TABLE FOR GCR

DECIMAL	HEX	BINARY	GCR	HEX	DECIMAL
00	00	0000	01010	00	00
01	01	0001	01011	01	01
02	02	0010	10010	02	02
03	03	0011	10011	03	03
04	04	0100	01110	04	04
05	05	0101	01111	05	05
06	06	0110	10110	06	06
07	07	0111	10111	07	07
08	08	1000	01001	08	08
09	09	1001	11001	09	09
10	0A	1010	11010	0A	10
11	0B	1011	11011	0B	11
12	0C	1100	01101	0C	12
13	0D	1101	11101	0D	13
14	0E	1110	11110	0E	14
15	0F	1111	10101	0F	15

WORKING 'INSIDE' THE DISK DRIVE

In order to work inside the disk drive, it will be necessary to understand how to use an 'essential' tool. The tool that we are speaking of is called 'DRVMON64' by STARPOINT SOFTWARE (the DI-SECTOR people). DRVMON64 is probably the most powerful tool available for working inside the disk drive. DRVMON64 is also one of the most overlooked tools around. DRVMON64 is copyrighted by STARPOINT SOFTWARE and contained on the DI-SECTOR disk. One interesting fact about this program is that it is not copy protected.

Two other items that you should have in your 'tool kit' are the books INSIDE COMMODORE DOS by Richard Immers and Gerald Nuefeld and THE ANATOMY OF THE 1541 DISK DRIVE by Abacus. With these two books and DRVMON64 there is virtually nothing that cannot be accomplished inside the 1541. INSIDE COMMODORE DOS is an excellent book, well written, easy to understand and by far the most comprehensive book on the disk drive.

HOW TO USE DRVMON64

DRVMON64 is a ML monitor that will work equally well in the disk drive or the computer. It is possible to transfer data between the disk drive and the computer. It is possible to assemble or disassemble code in the computer or directly in the drive's memory. As a matter of fact, all of the ML monitor functions that are available in the computer are available in the drive with DRVMON64.

If you don't currently have a copy of the DRVMON64 we would strongly suggest that you obtain one (directly from STARPOINT SOFTWARE, STAR ROUTE 10, GAZELLE, CA. 96034). Since it is a copyrighted program we cannot supply you with a copy of it. If you already own DRVMON64 we suggest that you get out your copy of the DI-SECTOR manual and read the DRVMON64 instructions before proceeding any further. If you don't currently own a copy DRVMON64, read this chapter anyway. You will still get a lot out of it. Use the program called 'GCR READ' from the last chapter, making the appropriate changes in the BASIC program.

Load and execute the version of DRVMON64 that resides at 49152 ($C000). This the version of the program that we will use for our discussion. When DRVMON64 is first executed the monitor will be working in the computer's memory. To use the monitor in the disk drive simply type 08 (0 the letter, not 0 the number) then RETURN. You will notice that there is a ']' (bracket sign) next to the cursor. This indicates that you are in the disk drive's memory. Let's try some real easy commands. Type 'M 0000 003F' (RETURN). The code that you see is the actual code from the disk drive's memory starting at $0000 and ending at $003F. You may also use the other commands (A, D, F, G, etc.) when you are in the disk drive. Try the following command 'D FCAA' (RETURN). You will now see a portion of the ROM memory

disassembled. DRVMON64 is fast, it is easy to use and it is probably the best tool for examining the disk drive's memory.

For now we will concentrate on one area of the drive's memory, the job command queue #0 located at $0000 and the track and sector for buffer #0 located at $0006 (track) and $0007 (sector). The disk drive will use these locations to perform many of its tasks. When the disk drive reads a block of data from the disk it will use the job command queue located at $0000 to perform this function. This location is checked by the disk drive through its interrupt routine (IRQ). If a routine is going to use the job command queue it is essential that the Interrupt flag be clear (0). If the proper value (command code) is stored at location $0000 the drive will execute the function that corresponds to the command code. Following is a table that describes the command codes and their meanings. All values are in hex:

80	READ A SECTOR
90	WRITE A SECTOR
A0	VERIFY A SECTOR
B0	SEEK A TRACK (ANY SECTOR)
B8	SEEK A TRACK & SECTOR
C0	BUMP - FIND TRACK 1
D0	JUMP TO ML PROGRAM IN BUFFER
E0	EXECUTE CODE IN BUFFER - FIRST THE DRIVE WILL BRING THE DISK UP TO SPEED SET THE DENSITY TO THE PROPER VALUE AND SEEK THE PROPER TRACK. THEN THE CODE IN THE BUFFER WILL BE EXECUTED.

The track and sector number should be stored at locations $0006 and $0007 respectively BEFORE storing the command code at location $0000. When we use DRVMON64 it is possible to set all the values at the same time and let the program do the rest for us. So much for theory, let's have some fun! Format a disk and leave it in the disk drive. This way we can have something to work with. Do not experiment with a valuable disk. We will use the M command of DRVMON64 to perform the following experiments. For the sake of clarity it may help to remove the cover of the disk drive and the metal shield covering the circuit board. This will enable you to see the actual results (of the R/W head) that we are going to obtain. Type in the following lines directly next to the] (bracket sign).

```
]@I   (RETURN)
```

This will initialize the disk drive. DRVMON64 also contains a built in DOS wedge. Now type in the following line. Don't add any extra spaces or data to this line.

```
]F 0300 03FF 00    (RETURN)
```

This command will fill data buffer 0 (located from $0300 to $03FF) with 00's. This gives us a fresh work space. Next type in the following line.

```
]:0000 80 00 00 00 00 00 01 05   (RETURN)
```

Let's examine the above line. The $80 (at location $0000) specifies read a block. The $01 (at location $0006) specifies the track number. The $05 (at location $0007) specifies the sector number. If everything went as expected the drive will come to life, the disk will start spinning, the head will move and track 1, sector 05 will be read into buffer 0 (located from $0300 to $03FF). Before we examine the code from $0300 to $03FF let's be sure that the drive was able to properly read the data. To do this it will be necessary to examine the job command queue for an error message. After a command is processed by the drive, the DOS will store an error code back in the command code queue. The error codes are interpreted as follows:

01	NO ERROR - JOB COMPLETED
02	HEADER BLOCK NOT FOUND
03	SYNC NOT FOUND
04	DATA BLOCK NOT FOUND
05	DATA BLOCK CHECKSUM ERROR
07	VERIFY ERROR (AFTER WRITE)
08	WRITE PROTECT ERROR DURING WRITE
09	HEADER BLOCK CHECKSUM ERROR
0A	DATA BLOCK TOO LONG
0B	ID MISMATCH ERROR
10	BYTE DECODING ERROR

Type 'M 0000 0007' (RETURN). We will now be able to examine the contents of location $0000 for the error message. If there was no error the value of $01 will be contained at location $0000. If any other value is contained there, repeat the procedure starting with the '@I' command. Use the 'M' command to examine memory from $0300 to $03FF. This is the area of memory that contains the data block that was read.

Now that you can read a normal data block into memory let's go a little farther. Just by substituting a $90 at location $0000 we can write the data at $0300-$03FF out to any sector. Try using the other commands to Verify, Seek or Bump. One word of caution though; if you use the JUMP or EXECUTE commands be sure that there is a valid ML routine at location $0300. If you don't have a valid ML routine located there the disk drive may lock up. A power-off or a RESET will be required to restore your drive to normal.

It is not necessary to use a valid track or sector number at memory locations $0006 & $0007. Try repeating the above procedure with a track number of $24 (decimal 36). It is

possible to move the head to any track on the disk (1 to 40) and read data from the disk. The data that will be read must have been written in the standard 1541 format. Otherwise an error will be returned and the data read will not be valid. It is interesting to note that the head will not 'BUMP' when errors are encountered when using the job command queue to execute direct commands. Try removing your disk and execute the read command. The head will not BUMP when an error is encountered if we are directly executing the command through the job command queue.

Warning: When you are directly using job queue commands, you are controlling the disk drive at a very fundamental level. At this level the drive obeys your commands without doing much (if any) error checking. You can easily wipe out a disk, even if the write protect is covered!! When you are testing your routines always use an old disk or one which you have a copy of.

If you wish to use the EXECUTE command, it will be necessary to write your own ML read and write routines. With the use of these routines it is possible to read and write data to the disk ANYWHERE from track 1/2 to track 40 1/2.

One other very interesting area in the disk drive's memory is the disk controller port B, location $1C00. Memory location $1C00 is where we can control the stepper motor, the drive motor and the density selection. Each bit of the byte at location $1C00 has a special function (location $1C00 is the disk controller I/O port). Following is a table that illustrates the function of each bit

BIT	VALUE	FUNCTION
0	$01	BITS 0 & 1 ARE CYCLED TO STEP
1	$02	(MOVE) THE HEAD IN AND OUT
2	$04	MOTOR ON AND OFF (1 BIT IS ON)
3	$08	DRIVE BUSY LED (LIGHT)
4	$10	WRITE PROTECT DETECT
5	$20	DENSITY SELECT
6	$40	DENSITY SELECT
7	$80	SYNC DETECT

Let's just try another little experiment. We will be changing the values contained at location $1C00. Be sure that the cover is removed from your drive so that you may see the half tracking. Initialize your drive (@I) then type in the following line from DRVMON64. Be sure you are working in the drive's memory (]).

```
]M 1C00 1C07   (RETURN)
```

You will see that the value contained a location $1C00 is a $D2 (binary %1101 0010). Examining the binary value we can see that bit 2 contains a 0. This means that the drive motor is off (remember bits are numbered from 0-7). If we want to turn on

the drive motor all we have to do is set bit 2 to the value of 1. This will result in the binary value of % 1101 0110 (hex $D6). To store the value of $D6 in memory location $1C00 use the following command.

]:1C00 D6   (RETURN)

The drive motor will come on and disk will spin. The disk will continue to spin as long as bit number 2 of $1C00 is set to a value of 1 (HEX $D6 = BINARY % 1101 0110). To turn the motor off move the cursor up to the $D6 and change it to the value of $D2 (then press RETURN). The motor will turn off. In order to move the head in half tracks it will be necessary to cycle the 0 & 1 bits in the following fashion. First initialize the drive (@I). Then use the following command:

]:1C00 D6   (RETURN)

Move the cursor back up to the $D6 and change the value to $D7, then $D4, then $D5, then $D6, then $D7, then $D4, then $D5, then $D6, etc. (be sure to omit the '$' when storing the values to memory). To move the head in half tracks all we are really doing is cycling bits 0 & 1 of location $1C00. Use the following sequence of bit values 00, 01, 10, 11, 00, 01, 10, 11, etc. Start with the appropriate bit pattern (10) and cycle through the others. Change only those bits that directly affect the location of the head (bits 0 & 1). The first $D6 will just turn the drive motor on, the $D7 will move the head 1/2 track in, the $D4 will again move the head 1/2 track in, etc. If you were to analyze the bit pattern used at location $1C00 it will become apparent that all we are doing is cycling bits 0 & 1 to move the head. Conversely, if we wish to move the head out from track 18, we would do the following. Initialize the drive (@I) so that we start with the head at track 18. Type in the following:

]:1C00 D6   (RETURN)

This $D6 will turn on the drive. Then move the cursor back up to the $D6 and change the value to $D5, then $D4, then $D7, then $D6, then $D5, then $D4, then $D7, then $D6, etc. In order to move the head in the opposite direction all we have to do is cycle bits 0 & 1 in the reverse order. In the above example we directly changed (cycled) the bits in location by storing the proper values at location $1C00. You will find that more experienced programmers will cycle the bits by ANDing and ORAing the bits of location $1C00. It is not important how the cycling of bits takes place (directly storing values or by AND & ORA). What is important is that the bit pattern at location $1C00 is cycled in the proper sequence.

Two other bits of location $1C00 are important to us when we are working in the disk drive. These are bits 5 and 6 of location $1C00. Bits 5 and 6 control the density at which data will be read from or written to the disk. The following chart reflects the density selections available on the 1541.

BIT 6	BIT 5	TRACKS
1	1	1 - 17
1	0	18 - 24
0	1	25 - 30
0	0	31 - 35

In order to read data from a track written at the normal density, bits 5 & 6 must be set properly. If data is read from the disk at the wrong density the data will appear to be garbage. If you are going to move the head yourself (by cycling bits 0 & 1) be sure to set the density to the proper value prior to reading or writing from the disk.

The next locations that we will cover in this chapter are location $1C01 (data port A) and location $1C03 (data direction for port A). Location $1C01 is the data port for GCR data transfer to and from (I/O) the disk. Location $1C03 is the data direction port for location $1C01. This location controls whether we will read data from or write data to the disk. Storing a value of $FF at location $1C03 will allow data to be written to port A (location $1C01); this will turn on the write mode of port A. Storing a value of $00 at location $1C03 will allow us to read data from port A (location $1C01); this will turn on the read mode of port A.

Before we may actually write to the disk one more area of the disk drive must be explained. This is the Peripheral Control Register (PCR) located at $1C0C. The PCR is used to change the Read/Write (R/W) Head from the read mode to the write mode. Bit 5 of the PCR is used to control the mode of the R/W head. When bit 5 of location $1C0C is set to a value of 1 the drive will be in the read mode. When bit 5 is cleared (0) the R/W head will be in the Write mode. Always set this location to the desired mode prior to selecting location $1C03. After every normal DOS routine this location is set to read.

The internal DOS of the disk drive will take care of cycling the bits to move the head, set the density during normal operation, set the PCR to the proper mode and set the data direction port. The DOS will also perform the actions faster and more accurately than we can with our crude little experiment here. We hope that you have learned a little about the inner workings of the disk drive. Don't be afraid to experiment; try moving the head to various tracks. If you go too far the drive may have to be initialized. Just don't perform too many 'BUMPS' and you should not experience any problems. Be sure to remove the cover of your drive so that you may see the Read/Write head move to these 'exotic' locations.

Time for review! We have covered a lot of material here. Let's take a short break and recap what you have learned.

1). The command queue is checked by the IRQ routine of the disk drive. When a valid command code is stored in the queue, the appropriate function will be executed. Since the command queue uses the IRQ to process its commands the Interrupt flag of the drive must be clear in order to function. When a programmer directly uses the job command queue the head will not 'bump' if an error is encountered.

2). Memory location $1C00 is primarily used to control the stepper motor, the drives motor and to select the proper density.

3). Location $1C01 (data port A) is used to read data from or write data to the disk.

4). Location $1C03 (data direction port) is used to control the direction of the data port A (located at $1C01). When location $1C03 is set to $FF data port A is in the write mode. When location $1C03 is set to $00 data port A is in the read mode.

5). Location $1C0C (PCR) controls the mode of the R/W head itself. Bit 5 of this location is used to control the selection of read (1) or write (0). Set this location to the proper mode prior to setting the data direction port (location $1C03). Always leave the PCR in the read mode after writing to the disk!

Before we discuss some of the more exotic protection methods available on the 1541 it is essential to understand the normal format that is used on the 1541 disk drive.

The disk drive begins the format routine with the familiar 'BUMP'. The disk drive is forced to step the R/W head outward 46 tracks (towards track 1). Since even the best of disk drives can only go to track 41, this insures that the R/W head will definitely hit the end stop. The R/W head is positioned by BUMPing up against the end stop and then 'bouncing' away from the end stop approximately 0.006 to 0.010 of an inch. This initial BUMP is critical to the overall alignment of the 1541. If the clearance (0.006-0.010 in.) is not correct, the disk will be formatted out of alignment. All the tracks are dependent upon the initial clearance obtained after the BUMP. For instance, if the clearance obtained after the BUMP is 0.001 in. all of the tracks formatted by this drive will be 0.005-0.009 in. out of alignment. Remember that the BUMP initially positions the R/W head at the proper location.

The 1541 uses a track spacing of 48 tracks per inch (48 tpi). This means that each track is approximately 0.020 in. apart center to center. In order for the head to move from track to track it is necessary for the stepper motor to step two times. Each step of the stepper motor is only 0.010 in. (approx.). This means that the stepper motor is capable of moving the head at the rate of 96 tracks per inch (96 tpi or appx. 0.010 in. per step). Half tracking (96 tpi) is a normal and necessary function of the 1541 disk drive. The movement of the stepper motor is controlled by the DOS (Disk Operating System). During the normal format procedure, data is written only on the full tracks (48 tpi).

We have now established that the R/W head of the drive is left approximately 0.010 in. away from the end stop after a BUMP. This roughly corresponds to the center to center distance of a half track. Now that the drive has properly located the R/W head at track 1 the format procedure will begin.

When formatting a track, the drive first puts the R/W head into write mode and sets the controller to the proper density for that track. Next the disk drive will write a track full of sync marks ($FF) to the disk, thereby erasing the track. Then the drive will write 4000 bytes of the value $55 to the disk. Approximately 1/2 the track will contain a giant sync mark ($FF's) and the other half will contain a series of $55's. Now the DOS will go through a complicated and time consuming routine to establish the actual length of the track in number of bytes. Each drive will vary in the rotational speed of the disk. Some drives are faster, others are slower. If a drive spins slower than normal there will be more room for data on a

track (bits will be closer together). If a drive spins faster than normal there will be less room for data on the track (bits will be farther apart). Remember, bits are timed out (clocked out) at a certain number of bits per second. This complicated calculation procedure will compensate somewhat for the speed variations of the disk drive. Once the approximate number of bytes that can be stored on the track is calculated, the length of the gap required between sectors is determined (inter-sector gap or sector tail gap). The length of this gap is varied as necessary from one track to another in order to keep the spacing between the sectors on a track uniform.

The drive will then compute the header images for the particular track that is to be formatted. These images are the GCR coded form of the actual header blocks for the sectors on the track. Each will include the header block identifier ($08), the checksum for the header, the sector number, the track number, the second ID byte, the first ID byte, two $0F bytes. All of the images for the track will be created and stored in RAM ahead of time. The GCR image of a dummy data block is also created and stored in RAM. The data block image includes the data block identifier ($07), 256 bytes of dummy data, the data block checksum and two $00 bytes. Next the track is 'cleared' by writing the value $55 to track 10240 times. This will again erase the track. Following is a representation of how the header block will be written to the disk.

SYNC MARK	HEADER BLOCK ID	SECTOR NUMBER	TRACK NUMBER	ID2	ID1	$0F	$0F	HEADER GAP

*NOT GCR

The header block is immediately followed by the data block. Following is a representation of how the data block will be written to the disk.

SYNC MARK	DATA BLOCK ID	256 DATA BYTES	DATA BLOCK CHKSUM.	$00 $00 BYTES	INTER-SECTOR GAP	NEXT SECTOR

Now the actual formatting of the track will begin. First, five $FF bytes are written to the disk as a sync mark. Sync marks are written directly to the disk as the value $FF (%1111 1111); they are NOT converted into GCR. Next, the GCR image for sector 0's header is written. Then the header gap is written to the disk. This consists of eight $55 bytes. The header gap is NOT converted into GCR, it is written to the disk as the value $55. Next, the data block sync mark is written to the disk. This again consists of five $FF bytes written directly to the disk. Then the data block (including identifier $07) is written to the disk in the GCR coded form. The drive will now write out the proper number of inter-sector gap bytes for that track. The number of gap bytes written was determined by the elaborate calculations before formatting began.

This whole process will be repeated until all of the sectors have been written to the track. Once the last sector has been

written to the disk the drive will kill the write mode and leave the drive in the read mode. Now the drive will attempt to verify the format of the disk. If the format verified OK the drive will step the R/W head to the next track and begin the procedure again. Another track will be formatted until all 35 tracks are done.

After the disk has been formatted the BAM and other information on track 18, sector 0 will be written to the disk. The disk is now completely formatted and ready to accept data and/or programs. Most microcomputers will use the same types of disks (5-1/4 in. diameter). What makes a disk unique is the format or the way that data will be stored on the disk.

Whenever the 1541 disk drive is in the write mode (i.e. writing data to the disk) a 'GUARD BAND' will also be written to the disk. This guard band will erase an area on the disk to either side of the track. The purpose of a guard band is to prevent one track's information from 'bleeding' over to an adjacent track. This guard band will also erase data written to a disk from a drive that was out of alignment. The guard band will also erase any information that was previously written on either of the two adjacent half tracks. The area erased by the guard band will ensure that there is no track to track interference of the data.

Let's look at a 'standard' disk, that was formatted on a 'standard' drive. For the purposes of our illustration we must make some assumptions:

1). The rotational speed of the disk is exactly 200 milliseconds (five revolutions per second). The speed will not vary from 200 ms. during the format procedure.

2). The disk drive is in perfect mechanical and electrical condition.

3). The media (disk) used is flawless and certified to the highest standards.

4). There will not be any 'glitches' during the format procedure.

It is quite unrealistic that any disk drive will be able to maintain the standards that we are going to use for our illustration. If it was possible to maintain these standards let's see what we might find on our standard disk.

STANDARD DISK DATA (TYPICAL)

FIELD	HEX BYTES	GCR BITS	
HEADER SYNC	5	40	(NOT CONVERTED INTO GCR)
HEADER	8	80	
HEADER GAP	8	64	(NOT CONVERTED INTO GCR)
DATA SYNC	5	40	(NOT CONVERTED INTO GCR)
DATA BLOCK	260	2600	(BLK ID, DATA, CHKSUM & 00'S)
TAIL GAP	8	64	(VARIES PER CALCULATIONS)

TOTAL BITS PER SECTOR		2888	(= 361 GCR BYTES PER SECTOR)

Now that we have established the length of a 'standard' sector on a disk, let's use this information to determine the total number of bytes used on a given track. To determine the total bytes used, we only have to multiply the number of sectors per track by the number of bytes used per sector. Remember, the bytes per sector is a typical value and may vary.

TRACKS	SECTORS PER TRACK		GCR BYTES PER SECTOR		TOTAL GCR BYTES USED PER TRACK
1-17	21	X	361	=	7581
18-24	19	X	361	=	6859
25-30	18	X	361	=	6498
31-35	17	X	361	=	6137

Let's examine a few more of the relationships that exist on the 'standard' 1541 format.

TRACKS	BIT DENSITY	BYTE DENSITY	SECTORS /TRACK	GCR BYTES REQUIRED PER TRACK	GCR BYTES AVAILABLE PER TRACK	% OF BYTES USED PER TRACK	UNUSED BYTES PER TRACK
1-17	3.25 us	26 us	21	7581	7692	98.5%	111
18-24	3.50 us	28 us	19	6859	7142	96.0%	283
25-30	3.75 us	30 us	18	6498	6666	97.7%	168
31-35	4.00 us	32 us	17	6137	6250	98.2%	113

In order to determine the GCR BYTES AVAILABLE PER TRACK, all that is necessary to do is divide the time required for one rotation (200 milliseconds or 0.2 seconds) by the time required to write one byte to the disk (byte density). In the above example it will be found that track 1 byte density equals 26 us/byte (26 microseconds or 0.000026 seconds per byte). If we divide 0.2 seconds (one rotation time) by 0.000026 sec/byte (byte density) we will obtain a value of 7692 bytes available per track. The difference between the bytes available and the bytes required may be referred to as the unused bytes per track. These unused bytes are $55's that were written to the disk just prior to the actual format. They just provide a gap on the disk to prevent over writing of data on the disk due to speed variations. During the normal operation of the DOS these bytes will not be used again, nor will the tail gap bytes.

Before we can begin to understand how to modify the data on the disk for use in a protection scheme, it is essential to have a grasp of how the data would normally stored on a track. Review the above data before proceeding further.

CUSTOM DOS ROUTINES

In this chapter we'll present several custom DOS routines. These will allow you to examine disks for various types of formatting irregularities such as data written on half-tracks, altered density, extra sectors, extra tracks, long data blocks and nonstandard sync marks. The routines are similar to what you might find in a protected program, and so may be adapted for use in your own protection schemes.

In order to use these routines, you will need to have DRVMON64 from the DI-SECTOR package. The ease with which we can transfer a routine to the drive's memory, execute it and examine the results with DRVMON makes it alone well worth the price of the entire DI-SECTOR package. If you don't have DRVMON you can still learn about the workings of the disk drive from this chapter, but you'll miss out on the hands-on experience. If you are serious about exploring the 1541, DRVMON is essential (there may be other programs similar to DRVMON which can be used, but all our examples will be based on DRVMON).

Make sure you have read the preceding chapters on using DRVMON, on GCR and on the standard 1541 format. A reference book such as INSIDE COMMODORE DOS or ANATOMY OF THE 1541 will also prove invaluable. When you experiment with these routines be sure to use disks with expendable information, since we will operating at a very fundamental level of the drive. Any mistake could be disastrous to the data on the disk.

STANDARD FORMAT

Let's very briefly review the standard disk format. Each standard sector consists of two parts: the header block (containing header block identifier and checksum, sector and track numbers, disk ID and padding bytes) and the data block (containing the data block identifier, 256 data bytes, data checksum and padding bytes). This information is not stored on the disk in ASCII characters but rather in a special form called GCR. The header block is preceded by a SYNC mark (usually 40 1-bits = 5 $FF's) and followed by the header gap. The data block is next, also preceded by a sync mark and followed by the inter-sector (tail) gap. This pattern is repeated throughout a track; the tail gap at the end of the last sector on a track is usually larger than the other tail gaps on that track. There are 35 tracks on a standard 1541 disk, spaced approximately .020 inch apart, center to center (48 tracks per inch). The outer tracks are written at a higher density than the inner tracks.

The first program we'll look at is called "READ GCR". We have given you both the source (assembler) and object (binary) versions of the program. The source code is compatible with the Commodore, PAL and other assemblers. This is handy if you want to make changes to the code yourself. The source code version is called "READ GCR.ASM" and the object code (executable) version is "READ GCR.OBJ".

Figure DOS-1 shows a combined listing of both the object code (in hex) and the source code. This routine loads into the computer at $4300, but is designed to reside in the drive's memory at $0300. In the source code on disk, all the instruction locations are given in the $4300 range. For convenience we have changed the instruction locations in figure DOS-1 to the $0300 range. All the program examples in this chapter will follow this pattern.

Let's assume for now that we have turned on the drive motor, moved to the proper track, etc. and are ready to begin executing our routine (we'll see in a minute just how to accomplish this). Refer to figure DOS-1. The area from $0400-04FF, which we'll call the storage buffer, will receive the data read in from disk by our routines. After clearing the storage buffer, the code at $030B-0315 sets the R/W circuitry into read mode. In order to start at the beginning of a block (header or data), we have to wait until a sync mark is found. This is the purpose of the code at $0316-0328. We loop around waiting for a sync mark to be detected (Bit 7 of location $1C00 is used to detect sync marks). If one isn't detected within about .5 second (a very generous margin - over 2 revolutions!) we store a 'no SYNC mark error' (code $03) in location $0000 and call it quits. If and when we do find a sync mark, the first byte read is always an image of the sync, so we will discard it. The bytes themselves appear at location $1C01, so we simply load a byte from that location and then go on.

Now we are finally ready to read valid data bytes. Since bytes are clocked in one bit at a time, we have to wait until the entire byte has been read in. This is signalled by the overflow flag V being set to 1 (BYTE READY). We start by clearing the V flag (CLV) and then waiting at GETBYTE ($032F) as long as it is still clear. Once BYTE READY is signalled, we simply load the data byte from $1C01 and save it to our storage buffer, indexed by the Y-register. We continue getting bytes by looping back to GETBYTE until the buffer is full. At that point we will stop. We store a 'no error' (code $01) into location $0000. Note that the error code is placed into the same location our 'E0' command was originally put. This is very important, since if we don't wipe out the execute command when done, the DOS may keep re-executing our routine. Finally, we terminate the routine by jumping to the DOS ROM error handling routine. This handles turning off the drive motor and resuming normal disk controller functions.

FIGURE DOS-1: READ GCR

```
1030:  0300
1040:  0300                    *=   $0300        ;READ GCR V1.26
1050:  0300 78                 SEI
1060:  0301 A0 00              LDY  #$00        ;FILL WORK SPACE WITH 00
1070:  0303 A9 00              LDA  #$00
1080:  0305           CLEARIT  =    *
1090:  0305 99 00 04           STA  $0400,Y      ;STORE 00 AT $0400-$04FF
1100:  0308 C8                 INY
1110:  0309 D0 FA              BNE  CLEARIT
1120:  030B 20 00 FE           JSR  $FE00       ;SET PCR TO READ MODE
1130:  030E AD 0C 1C           LDA  $1C0C
1140:  0311 09 0E              ORA  #$0E
1150:  0313 8D 0C 1C           STA  $1C0C
1160:  0316 A2 00              LDX  #$00        ;SET UP TIMER FOR SYNC
1170:  0318 A0 00              LDY  #$00
1180:  031A           TIMEOUT  =    *
1190:  031A 88                 DEY
1200:  031B D0 07              BNE  WAITSYNC
1210:  031D CA                 DEX
1220:  031E D0 04              BNE  WAITSYNC
1230:  0320 A9 03              LDA  #$03        ;03=NO SYNC   IF NO SYNC THEN END
1240:  0322 D0 19              BNE  ENDIT
1250:  0324           WAITSYNC =    *        ;CHECK FOR SYNC
1260:  0324 2C 00 1C           BIT  $1C00
1270:  0327 30 F1              BMI  TIMEOUT
1280:  0329 AD 01 1C           LDA  $1C01       ;SKIP FIRST BYTE
1290:  032C B8                 CLV
1300:  032D A0 00              LDY  #$00
1310:  032F           GETBYTE  =    *        ;READ DATA FROM DISK
1320:  032F 50 FE              BVC  GETBYTE     ;WAIT FOR BYTE READY
1330:  0331 B8                 CLV
1340:  0332 AD 01 1C           LDA  $1C01       ;LOAD BYTE FROM DATA PORT
1350:  0335 99 00 04           STA  $0400,Y     ;STORE DATA FROM $0400-$04FF
1360:  0338 C8                 INY
1370:  0339 D0 F4              BNE  GETBYTE
1380:  033B A9 01              LDA  #$01        ;01=NO ERROR
1390:  033D           ENDIT    =    *         ;FINISH UP AND END
1400:  033D 85 00              STA  $0000       ;STORE ERROR CODE IN COMMANAD QUEUE
1410:  033F 4C 6E F9           JMP  $F96E       ;ROM ROUTINE TO END
```

FIGURE DOS-2: READ GCR 1K

```
1050:  0300
1060:  0300                    *=   $0300        ;READ 1K OF GCR V3
1070:  0300 78                 SEI
1080:  0301 A0 00              LDY  #$00        ;FILL WORK SPACE WITH 00
1090:  0303 A9 00              LDA  #$00
1100:  0305           CLEARIT  =    *
1110:  0305 99 00 04           STA  $0400,Y      ;STORE 00 AT $0400-$04FF
1120:  0308 C8                 INY
1130:  0309 D0 FA              BNE  CLEARIT
1140:  030B 20 00 FE           JSR  $FE00       ;SET PCR TO READ MODE
1150:  030E AD 0C 1C           LDA  $1C0C
1160:  0311 09 0E              ORA  #$0E
1170:  0313 8D 0C 1C           STA  $1C0C
1180:  0316 A2 00              LDX  #$00        ;SET UP TIMER FOR SYNC
1190:  0318 A0 00              LDY  #$00
1200:  031A           TIMEOUT  =    *
1210:  031A 88                 DEY
1220:  031B D0 07              BNE  WAITSYNC
1230:  031D CA                 DEX
1240:  031E D0 04              BNE  WAITSYNC
1250:  0320 A9 03              LDA  #$03        ;03=NO SYNC   IF NO SYNC THEN END
1260:  0322 D0 23              BNE  ENDIT
1270:  0324           WAITSYNC =    *        ;CHECK FOR SYNC
1280:  0324 2C 00 1C           BIT  $1C00
1290:  0327 30 F1              BMI  TIMEOUT
1300:  0329 AD 01 1C           LDA  $1C01       ;SKIP FIRST BYTE
1310:  032C B8                 CLV
1320:  032D A0 00              LDY  #$00
1330:  032F           GETBYTE  =    *        ;READ DATA FROM DISK
1340:  032F 50 FE              BVC  GETBYTE     ;WAIT FOR BYTE READY
1350:  0331 B8                 CLV
1360:  0332 AD 01 1C           LDA  $1C01       ;LOAD BYTE FROM DATA PORT
1370:  0335 99 00 04           STA  $0400,Y     ;STORE DATA FROM $0400-$04FF
1380:  0338 C8                 INY
1390:  0339 D0 F4              BNE  GETBYTE
1400:  033B EE 37 03           INC  $0337       ;USED TO FILL BUFFER FROM $0500-$07FF
1410:  033E AD 37 03           LDA  $0337       ;WITH GCR DATA FROM THE DISK
1420:  0341 C9 08              CMP  #$08
1430:  0343 D0 EA              BNE  GETBYTE
1440:  0345 A9 01              LDA  #$01        ;01=NO ERROR
1450:  0347           ENDIT    =    *          ;FINISH UP AND END
1460:  0347 85 00              STA  $0000       ;STORE ERROR CODE IN COMMAND QUEUE
1470:  0349 4C 6E F9           JMP  $F96E       ;ROM ROUTINE TO END
```

A couple of notes are in order about this routine. First, since we simply waited for any sync mark before reading, we will come in at a random point on the track. We may find a data block after the sync or a header block. Second, the bytes we read in are raw GCR bytes. This is useful since you can 'see' exactly what is on the disk. If you want the corresponding hex bytes instead, you must convert the bytes yourself as outlined in the chapter on GCR recording (or use the DOS ROM conversion routine to do this for you). Third, we only read in 256 bytes after the sync. Recall from the chapter on standard disk format that since GCR is 25% longer than hex, a typical sector takes about 361 GCR bytes (header and data block together). Depending on which sync we pick up first, we will either read in just the first part of a data block, or else a whole header block followed by part of a data block (including the intervening sync mark, which will appear as only a single byte).

Now that we have a good idea of what the read routine itself entails, let's look at how we will get set up to execute it and how we will examine the results. In order to read from or write to the disk, a number of things must be done first. First of all, the drive motor must be turned on and brought up to the normal speed (300 rpm). Then the stepper motor must be directed to move the R/W head to the correct track. Based on the track we are moving to, the proper bit density must also be selected. The job of controlling the drive and stepper motors and setting the density could be done directly by our routine, but fortunately the DOS (Disk Operating System) has a built-in command called the EXECUTE command that will do this for us automatically and then jump into our routine.

At this point, the motor will be up to speed and the head will be sitting on the track, ready for our routine to do the actual reading or writing. In other words, from here on we're on our own! Depending on what we are doing, we may want to change the density, move to a half-track, search for a sync mark, etc. before reading. In our first routine we just wait for a normal sync mark and read the next 256 bytes. Let's execute it.

Start by loading into the computer the version of DRVMON that resides at $C000. Execute it with SYS 49152. You will see the start-up message and then the '.' command prompt. At this point you are operating DRVMON in the computer. All our work will done in the disk drive, so we have to switch over to working in the drive now by using the 08 command (that's the LETTER O, not the digit 0; the 8 is for device 8). You should be rewarded with the ']' prompt (to get back to working in the computer, type just O). All commands will now reference the drive's memory rather than the computer's, except for a load or save. The load command 'L' still loads the routine into the computer; we'll have to transfer the code to the drive with the transfer command 'TC'. To do a save we just reverse the process; transfer the data to the computer with 'TD' and save with 'S'.

Assuming you have DRVMON operating in the drive, load the first DOS program into the computer with:

```
]L "READ GCR.OBJ",08
```

Note that DRVMON should already have put the ']' prompt on the line, so don't type it. This program will load into the C64 at $4300, like all of our examples in this chapter.

Having loaded the program into the computer, you must now transfer it to the drive. The routine takes up much less than one block of memory, but we will just transfer a whole block so that all our routines can be done with the same transfer command, namely:

```
]TC 4300 43FF 0300
```

This transfers the memory from $4300-43FF in the computer to $0300 in the drive. Check to see that the routine has been transferred by disassembling part of the drive memory at $0300 with:

```
]D 0300
```

You should see the following code:

```
0300 78       SEI
0301 A0 00    LDY #$00
0303 A9 00    LDA #$00
0305 99 00 04 STA $0400,Y
etc.
```

Remove the program disk and insert a blank formatted disk. Next, initialize the drive with:

```
]@I
```

(DRVMON has a DOS mini-wedge built into it). The initialize command forces the drive to read the BAM at track 18, sector 0. Initializing is necessary in some of the experiments we'll do, and a good habit in general.

To execute the routine we will use the job queue command 'E0'. The job queue was discussed in the chapter on 'Working Inside the Disk Drive'. To use 'E0', we first put the number of the track we wish to examine into location $0006 of drive memory and then put $E0 into location $0000 (with DRVMON we can do them both at the same time). The next time the disk controller is looking for something to do it will see our command, get us positioned on the proper track at the proper speed with the proper read density set, and then jump to location $0300 and begin executing. To display the first 8 bytes of memory, type the following and hit RETURN:

```
M 0000 0000
```

Now cursor back up to the first byte displayed (after the]:0000) and type E0 over it. DON'T HIT RETURN. Cursor over to memory location $0006 (2nd byte from end) and type in your track number. Let's use track 18 ($12) for our example. After doing this your line will look like:

```
]:0000 E0 00 00 00 00 00 12 00 ........
```

(The 00 bytes may be different depending on previous operations) Take a deep breath and press RETURN. The drive should come alive, the head should step to track 18 and the red access light should come on briefly. Then everything shuts off again almost immediately - it's done.

The first thing you should do is check that the operation proceeded normally. The job error code returned by our program is automatically put back into location $0000 by the error handling routine, so display $0000 again with M 0000 0000. A normal termination will result in an $01 code at location $0000. If you don't have the $01 code, an error has taken place (a list of the error codes is given in the chapter on working inside the drive). In this case the storage buffer will be filled with $00 bytes. If you do have the $01 code, you are ready to examine the GCR data at $0400-$04FF. Display the beginning of this area with M 0400 0400. You should see either a $52 or a $55 as the first byte. These are part of the GCR codes for a header block identifier ($08) and a data block identifier ($07) respectively. The rest of the code can be deciphered with the aid of the GCR chapters. It might be instructive for you to stop here and do that for practice. I'll wait.

Now that you're back, let's look at an abnormal track. With the 'E0' command we can position the head at any track we wish, even past track 35! Some drives can go out to track 40, but many have trouble beyond track 37. If you send the head too far, it may become physically stuck. If this happens, you have two options. Remove the cover and back the head up by hand, or try to execute a 'BUMP request 'C0' through the job queue (OUCH!). We all know what that can do to your drive, so let's just stick to track 36. Execute the routine with track 36 ($24) specified. This time the job error code should be $03 (no sync). This corresponds to DOS error number 21 as reported normally through the error channel to the computer (in fact, by adding 18 to the job error code we get the DOS error number in most cases). We get this error because on a normal disk track 36 has never been formatted. Occasionally, a disk may come from the manufacturer with some random garbage out there that happens to correspond to a sync mark (10 consecutive 1-bits). Also, some commercial disks may use tracks 36-37 as part of a protection scheme. If you get a sync mark there, you might want to examine the storage buffer. If you don't get a sync mark, no data was read and the storage buffer will be all 00's.

A modified version of 'READ GCR' is also included on the program disk. This one is called 'READ GCR 1K' (.ASM and .OBJ). As the name implies, it is just like 'READ GCR' except that it reads 1024 consecutive bytes (1K) from the disk after finding a sync mark. These bytes are placed into drive memory at $0400-07FF. The program listing appears in figure DOS-2.

HALF-TRACKS

The next routine we'll present will allow you to position the head on a half-track and attempt to read the data there. We saw earlier that the standard tracks are about .020 inch apart (48 tracks per inch). However, the stepper motor is capable of positioning the head midway between two standard track positions. This is called half-tracking (96 tpi). We can read and write data on a half-track just as if it were on a whole-track, with one catch. Writing data on a half-track will render both of the adjacent whole-tracks unreadable, and vice versa (writing to a whole-track will wipe out the half-tracks on either side). To position the head on a half-track, we have to control its movement directly. This is done by stepping the two low order bits (bits 0 & 1) of location $1C00 sequentially through their four possible combinations (cycling). This was explained in the chapter on working inside the drive. The order in which you proceed through the sequence determines whether you move the head inward or outward.

The program is called 'HALF TRACK' (.ASM and .OBJ) and a listing is given in figure DOS-3. This routine works exactly like our first example except for half-tracking. To use it you load the routine into the computer, transfer it to drive memory with 'TC', initialize the drive with '@I0', enter the track number and execute the routine with 'E0'. The 'E0' command positions us to the whole-track nearest to where we want to be. Once there we will simply step outward or inward half a track and begin reading. The section of the program from $0316 to $0332 does the stepping for us. Basically, we get the value from $1C00, cycle the low bits to the next value in the sequence and store it back at $1C00. Using an INX statement at $0319 cycles forward in the sequence. This steps the head inward (forward; towards higher-numbered tracks). To step outward, simply replace the INX with a DEX. After stepping, we have to wait a little bit to give the head time to move. That's all there is to it. We can proceed with our regular read process. Load, transfer and execute this routine exactly like the previous one. If your disk does not have information written on the half-track, the results will be unpredictable. You may get an error returned, or you may seem to terminate normally. If you examine the storage buffer, however, you will see garbled data there. This comes from 'bleed-over' from the adjacent whole-tracks.

FIGURE DOS-3: HALF TRACK

```
1040:  0300
1050:  0300                    *=   $0300          ;ROUTINE TO READ HALF TRACKS V3
1060:  0300 78                 SEI
1070:  0301 A0 00              LDY  #$00          ;FILL WORK SPACE WITH 00
1080:  0303 A9 00              LDA  #$00
1090:  0305           CLEARIT  =    *
1100:  0305 99 00 04           STA  $0400,Y       ;STORE 00 AT $0400-$04FF
1110:  0308 C8                 INY
1120:  0309 D0 FA              BNE  CLEARIT
1130:  030B 20 00 FE           JSR  $FE00         ;SET PCR TO READ MODE
1140:  030E AD 0C 1C           LDA  $1C0C
1150:  0311 09 0E              ORA  #$0E
1160:  0313 8D 0C 1C           STA  $1C0C
1170:  0316 AE 00 1C           LDX  $1C00         ;STEP HEAD IN HALF TRACKS
1180:  0319 E8                 INX              ;CHANGE TO DEX TO MOVE HEAD OTHER WAY
1190:  031A 8A                 TXA
1200:  031B 29 03              AND  #$03          ;CYCLE BITS 0 & 1
1210:  031D 85 14              STA  $0014
1220:  031F AD 00 1C           LDA  $1C00
1230:  0322 29 FC              AND  #$FC
1240:  0324 05 14              ORA  $0014
1250:  0326 8D 00 1C           STA  $1C00         ;STEP HEAD 1/2 TRACK
1260:  0329 A2 AA              LDX  #$AA          ;WAIT FOR HEAD TO SETTLE
1270:  032B           DELAY1   =    *
1280:  032B A0 00              LDY  #$00
1290:  032D           DELAY2   =    *
1300:  032D 88                 DEY
1310:  032E D0 FD              BNE  DELAY2
1320:  0330 CA                 DEX
1330:  0331 D0 F8              BNE  DELAY1        ;HEAD SETTLED AT HALF TRACK
1340:  0333 A2 00              LDX  #$00          ;SET UP TIMER FOR SYNC
1350:  0335 A0 00              LDY  #$00
1360:  0337           TIMEOUT  =    *
1370:  0337 88                 DEY
1380:  0338 D0 07              BNE  WAITSYNC
1390:  033A CA                 DEX
1400:  033B D0 04              BNE  WAITSYNC
1410:  033D A9 03              LDA  #$03          ;03=NO SYNC   IF NO SYNC THEN END
1420:  033F D0 19              BNE  ENDIT
1430:  0341           WAITSYNC =    *          ;CHECK FOR SYNC
1440:  0341 2C 00 1C           BIT  $1C00
1450:  0344 30 F1              BMI  TIMEOUT
1460:  0346 AD 01 1C           LDA  $1C01         ;SKIP FIRST BYTE
1470:  0349 B8                 CLV
1480:  034A A0 00              LDY  #$00
1490:  034C           GETBYTE  =    *          ;READ DATA FROM DISK
1500:  034C 50 FE              BVC  GETBYTE       ;WAIT FOR BYTE READY
1510:  034E B8                 CLV
1520:  034F AD 01 1C           LDA  $1C01         ;LOAD BYTE FROM DATA PORT
1530:  0352 99 00 04           STA  $0400,Y       ;STORE DATA FROM $0400-$04FF
1540:  0355 C8                 INY
1550:  0356 D0 F4              BNE  GETBYTE
1560:  0358 A9 01              LDA  #$01          ;01=NO ERROR
1570:  035A           ENDIT    =    *           ;FINISH UP AND END
1580:  035A 85 00              STA  $0000         ;STORE ERROR CODE IN COMMANAD QUEUE
1590:  035C AD 0C 1C           LDA  $1C0C         ;KILL PCR
1600:  035F 29 FD              AND  #$FD
1610:  0361 8D 0C 1C           STA  $1C0C
1620:  0364 4C 6E F9           JMP  $F96E         ;ROM ROUTINE TO END
```

There is one thing to note about using the 'EO' job queue command to move the head initially. Before moving the head with 'EO', the DOS consults a stored value to see what track it is currently on. If it is being told to go to the same track it thinks it is already at, it will not move the head at all. This sounds reasonable until we realize that when we move the head with our routine, we don't change the stored value. Try this experiment. First, issue an '@I' command to initialize the drive. Set the job queue track to 30 ($1E) and execute the routine. The head will step out to track '30-1/2'. Now if you execute the routine again, the DOS still thinks the head is on track 30, so the 'EO' command WILL NOT move it back to 30. When the routine steps out a half-track again, the head will end up at track 31! What's more, it will stay a whole track off until you either power-off, initialize, or perform a job queue sector operation like read. To top it off, track 31 is written at a lower density than track 30. If we try to read at this point we may find that all we get is some garbled data (see the next section on density). The simplest way of avoiding problems like this is to initialize the drive with an '@IO' before execution.

CHANGING DENSITY

The next routine we will look at is 'DENSITY' (.ASM and .OBJ). This routine is also a variation of the 'READ GCR' routine, except that we change the density before attempting to read data. Density corresponds to the rate at which bits are written out to (and read back in from) the disk. This process is controlled by a clock which can be switched to one of four rates. These four different clock rates give us four different densities to choose from. Each density is normally used only in a particular 'zone' (range of tracks) on the disk. This explains why different tracks have different numbers of sectors on them. Within a zone the number of sectors per track is the same, but from zone to zone the number varies. The outer tracks (1-17) are written at the highest rate and have the most sectors; the inner tracks (31-35) are written at the lowest rate and have the fewest sectors. In other words, the higher the track number, the lower the density and number of sectors.

Density is controlled through bits 5 and 6 of location $1C00. The four possible values for these two bits correspond to the four available densities. The higher the value of these two bits, the higher the density selected will be. When you use the 'EO' or other job queue commands to move the head, the proper density is selected automatically based on the track you are moving to. Once control is transferred to our routine, we can alter the density by simply changing location $1C00. We have to be careful what densities we choose, however.

FIGURE DOS-4: DENSITY

```
1050:  0300
1060:  0300                    *=   $0300          ;ROUTINE TO READ CHANGED DENSITY 3
                     ;THIS ROUTINE WILL READ ANY TRACK AT TRACK 1 DENSITY
                     ;USE THIS ON THE INNER TRACKS FOR THE 'BEST' RESULTS
1090:  0300 78               SEI
1100:  0301 A0 00            LDY  #$00          ;FILL WORK SPACE WITH 00
1110:  0303 A9 00            LDA  #$00
1120:  0305          CLEARIT =    *
1130:  0305 99 00 04         STA  $0400,Y       ;STORE 00 AT $0400-$04FF
1140:  0308 C8               INY
1150:  0309 D0 FA            BNE  CLEARIT
1160:  030B 20 00 FE         JSR  $FE00         ;SET PCR TO READ MODE
1170:  030E AD 0C 1C         LDA  $1C0C
1180:  0311 09 0E            ORA  #$0E
1190:  0313 8D 0C 1C         STA  $1C0C
1200:  0316 AD 00 1C         LDA  $1C00         ;GET CURRENT DENSITY
1210:  0319 09 60            ORA  #$60          ;SWITCH DENSITY TO TRACK 1-17 DENSITY
1220:  031B 8D 00 1C         STA  $1C00         ;STORE CHANGED DENSITY AT $1C00
1230:  031E A2 00            LDX  #$00          ;SET UP TIMER FOR SYNC
1240:  0320 A0 00            LDY  #$00
1250:  0322 .        TIMEOUT =    *
1260:  0322 88               DEY
1270:  0323 D0 07            BNE  WAITSYNC
1280:  0325 CA               DEX
1290:  0326 D0 04            BNE  WAITSYNC
1300:  0328 A9 03            LDA  #$03          ;03=NO SYNC   IF NO SYNC THEN END
1310:  032A D0 19            BNE  ENDIT
1320:  032C          WAITSYNC =   *         ;CHECK FOR SYNC
1330:  032C 2C 00 1C         BIT  $1C00
1340:  032F 30 F1            BMI  TIMEOUT
1350:  0331 AD 01 1C         LDA  $1C01         ;SKIP FIRST BYTE
1360:  0334 B8               CLV
1370:  0335 A0 00            LDY  #$00
1380:  0337          GETBYTE =    *         ;READ DATA FROM DISK
1390:  0337 50 FE            BVC  GETBYTE       ;WAIT FOR BYTE READY
1400:  0339 B8               CLV
1410:  033A AD 01 1C         LDA  $1C01         ;LOAD BYTE FROM DATA PORT
1420:  033D 99 00 04         STA  $0400,Y       ;STORE DATA FROM $0400-$04FF
1430:  0340 C8               INY
1440:  0341 D0 F4            BNE  GETBYTE
1450:  0343 A9 01            LDA  #$01          ;01=NO ERROR
1460:  0345          ENDIT   =    *           ;FINISH UP AND END
1470:  0345 85 00            STA  $0000        ;STORE ERROR CODE IN COMMANAD QUEUE
1480:  0347 4C 6E F9         JMP  $F96E        ;ROM ROUTINE TO END
```

Through experimentation we have found that as long as you read or write a track at a LOWER than normal density, you should not have any trouble, even if you go to the lowest possible density. When WRITING at a lower than normal density, you are just giving the bits more 'room' on the disk than they need (you won't be able to write as many bits as normal on that track, however). When READING a track at a lower density than it was written, you should also be successful. The drive is able to compensate when bits come in a little faster than normal.

If you try to raise the density above normal, however, you begin to run into trouble. When WRITING at a higher than normal density, you may force the bits too close together for the disk to handle. The bits could 'blur' together and the information may not be written reliably. When READING, if you increase the density only one level higher than normal, you can usually manage to read a standard-sized sector. If you raise the density two levels above normal, you can sometimes read successfully, but you can rarely read at three levels above normal. In this situation the read circuitry is not prepared to wait as long as necessary for a bit to register. It comes to a premature conclusion about what the bit is, and you get false results. To summarize, you can lower the density to any level you desire for both reading and writing, but raising the density higher than normal can cause trouble.

Figure DOS-4 is the source code listing for 'DENSITY'. Again you load, transfer and execute this routine just like our first example (don't forget to initialize the drive first). If you compare this routine to 'READ GCR' you will see that a short section of code has been inserted at $0316. This code changes the density to the value for tracks 1-17 (highest density) by ORA'ing the current density value at $1C00 with $60. This sets bits 5 and 6 to 1's.

Different density levels might be used on the same track of a disk. If you want to examine such a disk, set the density to the lowest value and you will be able to read everything perfectly. Reproducing the disk presents a problem, however. Even though you can read the information, you can't tell directly what density it was written at originally. You may be able to get around this by timing how long the sector takes to be read, which will be less as the density gets higher.

NYBBLE COUNTING

A nybble is half of a byte, i.e. 4 bits or 1 hex digit. Nybble counting refers to counting the number of nybbles on a track or between two points on a track. Actually, 'nybble' is a misnomer; on the 1541 all we can read are whole bytes. The term comes from the Apple drives, which do allow true nybble counting. Since the term has become so entrenched in the computer jargon, we'll continue to use it.

The routine on the program disk to do nybble counting is called 'NYBBLE' (see figure DOS-5). The actual counting is done at lines $032E-0351. This routine finds a sync mark, waits for it to end, and then begins counting bytes (GCR of course). It counts until another sync is detected, waits until that sync is past, and resumes counting again. It continues this until it has passed all the sync marks on the track. In other words, it counts the total number of bytes on the track, EXCLUDING most sync bytes (each sync mark is counted as one byte by the routine).

The routine uses location $0014 in the drive (normally unused) to keep track of how many sync marks it has seen so far on the track, so it can tell when to stop. This sync count is initially set at $032E to the number of syncs on the track PLUS 1. The value there is correct only for tracks 1-17. There are 21 sectors on these tracks, and each sector has 2 sync marks (header and data block). Therefore the value used is 43 (21x2+1 = 43 = $2B). If you want to nybble count on a different track, be sure to change this value.

The 'nybble' count itself is kept in the X and Y registers while the routine is executing. This allows a 2-byte (16-bit) count, which is necessary since the number of bytes possible on a track is well over 256 (around 6000-7000). The count is stored at $0400-0401 in lo-byte/hi-byte order when the routine ends. The count normally varies from track to track on a disk, even on tracks in the same zone (e.g. 1-17). The count may even vary by one or two bytes when reading the same track repeatedly, due to variations in the media, drive speed and operation of the DOS. If this kind of routine was used in a protection scheme, you would have to allow a margin for error of a few bytes plus or minus. Note that the count includes the tail gap bytes put on the track during formatting, which can change from disk to disk. Because of this, the best idea when protecting a program would be to actually count the relevant track on each disk after creating it, and then store the count on the disk somewhere. The protection code in your program would nybble count the track and compare the result with the stored value. If the difference was outside the acceptable range, your program would crash itself.

There are many variations on the basic idea of nybble counting. You could count just gap bytes, or even the sync bytes. Another idea is to slow down the drive when creating a disk. This will allow us to get more bytes on a track than is normally possible, as we have already seen. This kind of disk is hard to duplicate with a copy program, since it just can't put that many bytes on a track without having your drive slowed down too.

FIGURE DOS-5: NYBBLE

```
1050:  0300                          P#
1060:  0300                  *=      $0300          ;ROUTINE TO NYBBLE COUNT A TRACK
1070:  0300 78               SEI
1080:  0301 A0 00            LDY     #$00         ;FILL WORK SPACE WITH 00
1090:  0303 A9 00            LDA     #$00
1100:  0305          CLEARIT =       *
1110:  0305 99 00 04         STA     $0400,Y      ;STORE 00 AT $0400-$04FF
1120:  0308 C8               INY
1130:  0309 D0 FA            BNE     CLEARIT
1140:  030B 20 00 FE         JSR     $FE00        ;SET PCR TO READ MODE
1150:  030E AD 0C 1C         LDA     $1C0C
1160:  0311 09 0E            ORA     #$0E
1170:  0313 8D 0C 1C         STA     $1C0C
1180:  0316          FINDSYNC =      *
1190:  0316 A2 00            LDX     #$00         ;SET UP TIMER FOR SYNC
1200:  0318 A0 00            LDY     #$00
1210:  031A          TIMEOUT  =      *
1220:  031A 88               DEY
1230:  031B D0 08            BNE     WAITSYNC
1240:  031D CA               DEX
1250:  031E D0 05            BNE     WAITSYNC
1260:  0320 A9 03            LDA     #$03         ;03=NO SYNC   IF NO SYNC THEN END
1270:  0322 4C 69 F9         JMP     $F969
1280:  0325          WAITSYNC =      *        ;CHECK FOR SYNC
1290:  0325 2C 00 1C         BIT     $1C00
1300:  0328 30 F0            BMI     TIMEOUT
1310:  032A AD 01 1C         LDA     $1C01        ;SKIP FIRST BYTE
1320:  032D B8               CLV
1330:  032E A9 2B            LDA     #$2B         ;NUMBER OF SYNCS+1 ON THE TRACK
1340:  0330 85 14            STA     $14          ;CONTER IS SET UP FOR TK 1-17
1350:  0332 A2 00            LDX     #$00
1360:  0334 A0 00            LDY     #$00
1370:  0336          SYNC1BEG =      *
1380:  0336 2C 00 1C         BIT     $1C00        ;FIND SYNC MARK BEGINNING
1390:  0339 30 FB            BMI     SYNC1BEG
1400:  033B          DECCOUNT =      *        ;DECREMENT COUNTER
1410:  033B C6 14            DEC     $14
1420:  033D F0 13            BEQ     ENDIT
1430:  033F          SYNC1END =      *
1440:  033F 2C 00 1C         BIT     $1C00        ;WAIT FOR END OF SYNC MARK
1450:  0342 10 FB            BPL     SYNC1END
1460:  0344          GETBYTE  =      *        ;START COUNT NOW
1470:  0344 50 FE            BVC     GETBYTE
1480:  0346 AD 00 1C         LDA     $1C00        ;CHECK FOR START OF SYNC BYTE
1490:  0349 10 F0            BPL     DECCOUNT
1500:  034B B8               CLV
1510:  034C E8               INX           ;INCREMENT BYTE COUNTER - LO BYTE
1520:  034D D0 F5            BNE     GETBYTE      ;COUNT 256 BYTES
1530:  034F C8               INY           ;INCREMENT BYTE COUNTER - HI BYTE
1540:  0350 D0 F2            BNE     GETBYTE
1550:  0352          ENDIT    =       *          ;FINISH UP AND END
1560:  0352 A9 01            LDA     #$01
1570:  0354 85 00            STA     $0000
1580:  0356 8E 00 04         STX     $0400
1590:  0359 8C 01 04         STY     $0401
1600:  035C 4C 69 F9         JMP     $F969
```

SYNCHRONIZED TRACKS AND TRACK ARCING

These two schemes are variations of the same basic idea. Rather than putting special information on a track, information is put on the track in a SPECIAL POSITION, RELATIVE TO ANOTHER TRACK. That is what we mean when we say the tracks are 'synchronized'. This works as follows: you find a reference sector on one track, say sector 0, then immediately step out to the next track and read in the first sector that comes along. Let's say it is sector 5. If you attempt to copy these tracks with a copy program, it may duplicate all of the information on the two tracks, but it won't reproduce the relative positions of the sectors (unless it is a very smart copy program). When you try to repeat the process of find-step-read, you'll jump into the second track at a different place than sector 5. This is a very subtle protection scheme since there are no abnormalities on the disk per se.

This basic idea can be expanded considerably. We could read a sector, step out to the next track and read a sector, step out and read again, etc. many times. We could even step back to a previous track. We could pick up a value from each sector as we go, maybe just the sector number itself. All the values would have to be right before the program will run. Synchronizing many tracks multiplies the difficulty of reproducing the disk.

If we add half-tracking to the idea of synchronized tracks, we get what is called track arcing. In this kind of scheme, we align the sectors on adjacent half-tracks. We said earlier that you can't write to two adjacent half-tracks without one interferring with the other. This problem is avoided by not writing complete tracks. We could write only 3 sectors on the first track, then step out a half-track and write another 3 sectors, then step out again and write 3 more. If you pay close attention to the relative time it takes to step out versus the time it takes for the disk to turn, and you don't write too many sectors on each track, you can successfully avoid placing any sectors side-by-side on adjacent half-tracks. You could even write a few sectors, step out a half-track and write again, then step BACK a half-track and write a few more sectors on the first track, if you are very careful.

When checking the synchronization of sectors in a protection scheme, you should allow a certain margin for error due to speed variations, etc. For instance, after finding the reference sector 0 and stepping out, you might accept either sector 5 or 6 as the first sector encountered on the next track. If you did find 6, you might wait until 5 comes around on this track before stepping out again. This would prevent differences from building up over the course of several tracks.

FIGURE DOS-6 SYNCHRONIZED TRACKS

```
1070:  0300                    *=   $0300          ;ROUTINE TO READ SYNCHRONIZED TRACKS V3
1080:  0300 78                 SEI
1090:  0301 A0 00              LDY  #$00           ;FILL WORK SPACE WITH 00
1100:  0303 A9 00              LDA  #$00
1110:  0305           CLEARIT  =    *
1120:  0305 99 00 04           STA  $0400,Y        ;STORE 00 AT $0400-$04FF
1130:  0308 C8                 INY
1140:  0309 D0 FA              BNE  CLEARIT
1150:  030B 20 00 FE           JSR  $FE00          ;SET PCR TO READ MODE
1160:  030E AD 0C 1C           LDA  $1C0C
1170:  0311 09 0E              ORA  #$0E
1180:  0313 8D 0C 1C           STA  $1C0C
                     ;SET UP TO FIND TRACK AND SECTOR
1200:  0316 20 10 F5           JSR  $F510          ;FIND PROPER SECTOR-ERROR #2 IF CAN'T FIND
1210:  0319 20 22 03           JSR  STEPHEAD       ;MOVE 1/2 TRACK
1220:  031C 20 22 03           JSR  STEPHEAD       ;MOVE 1/2 TRACK
1230:  031F 4C 40 03           JMP  FINDSYNC       ;JUMP TO READ THE DATA FROM THE ADJACENT TRACK
1240:  0322           STEPHEAD =    *
1250:  0322 AE 00 1C           LDX  $1C00          ;STEP HEAD IN HALF TRACKS
1260:  0325 E8                 INX              ;CHANGE TO DEX TO MOVE HEAD OTHER WAY
1270:  0326 8A                 TXA
1280:  0327 29 03              AND  #$03           ;CYCLE BITS 0 & 1
1290:  0329 85 14              STA  $0014
1300:  032B AD 00 1C           LDA  $1C00
1310:  032E 29 FC              AND  #$FC
1320:  0330 05 14              ORA  $0014
1330:  0332 8D 00 1C           STA  $1C00          ;STEP HEAD 1/2 TRACK
1340:  0335 A2 AA              LDX  #$AA           ;WAIT FOR HEAD TO SETTLE
1350:  0337           DELAY1   =    *
1360:  0337 A0 00              LDY  #$00
1370:  0339           DELAY2   =    *
1380:  0339 88                 DEY
1390:  033A D0 FD              BNE  DELAY2
1400:  033C CA                 DEX
1410:  033D D0 F8              BNE  DELAY1         ;HEAD SETTLED AT HALF TRACK
1420:  033F 60                 RTS
1430:  0340           FINDSYNC =    *
1440:  0340 A2 00              LDX  #$00           ;SET UP TIMER FOR SYNC
1450:  0342 A0 00              LDY  #$00
1460:  0344           TIMEOUT  =    *
1470:  0344 88                 DEY
1480:  0345 D0 07              BNE  WAITSYNC
1490:  0347 CA                 DEX
1500:  0348 D0 04              BNE  WAITSYNC
1510:  034A A9 03              LDA  #$03           ;03=NO SYNC   IF NO SYNC THEN END
1520:  034C D0 1C              BNE  ENDIT
1530:  034E           WAITSYNC =    *           ;CHECK FOR SYNC
1540:  034E 2C 00 1C           BIT  $1C00
1550:  0351 30 F1              BMI  TIMEOUT
1560:  0353 AD 01 1C           LDA  $1C01          ;SKIP FIRST BYTE
1570:  0356 B8                 CLV
1580:  0357 A0 00              LDY  #$00
1590:  0359 A2 0A              LDX  #$0A
1600:  035B           GETBYTE  =    *           ;READ DATA FROM DISK
1610:  035B 50 FE              BVC  GETBYTE        ;WAIT FOR BYTE READY
1620:  035D B8                 CLV
1630:  035E AD 01 1C           LDA  $1C01          ;LOAD BYTE FROM DATA PORT
1640:  0361 99 00 04           STA  $0400,Y        ;STORE DATA FROM $0400-$04FF
1650:  0364 C8                 INY
1660:  0365 CA                 DEX
1670:  0366 D0 F3              BNE  GETBYTE
1680:  0368 A9 01              LDA  #$01           ;01=NO ERROR
1690:  036A           ENDIT    =    *            ;FINISH UP AND END
1700:  036A 48                 PHA              ;SAVE ERROR CODE
1710:  036B A0 04              LDY  #$04           ;CONVERT GCR TO HEX
1720:  036D 84 31              STY  $31            ;SET UP POINTERS FOR GCR TO HEX CONVERSION
1730:  036F A0 00              LDY  #$00
1740:  0371 84 30              STY  $30
1750:  0373 20 E0 F8           JSR  $F8E0          ;CONVERT GCR TO HEX
1760:  0376 68                 PLA              ;GET ERROR CODE
1770:  0377 85 00              STA  $00
1780:  0379 4C 6E F9           JMP  $F96E          ;ROM ROUTINE TO END
```

The routine on the program disk to read synchronized tracks is called 'SYNCHRONIZED'. The listing is shown in figure DOS-6. At line $0316 we call upon a DOS ROM routine to find the reference sector initially. This sector number must be placed in location $0007 in drive memory (next to the track number at $0006) before issuing the 'E0' command to execute the routine. The ROM routine attempts to find the given sector on the selected track. If it doesn't succeed, it registers an error code $02 in location $0000 and does not return to our routine. If it does find the reference sector, it returns to our routine, which then steps out a track (two half-tracks) by calling on its own subroutine. Note that there is a small delay in the subroutine after stepping. This is to allow the head time to move and get settled. You may want to change this a bit, but don't reduce it too much.

If you want to step out more or less than one whole track, simply change the number of times you call the stepping subroutine. Once we get to the proper track, we simply enter our normal read process. We wait for the first sync to come along and read in the first 10 GCR bytes. This will capture an entire header block if it is encountered next (if the tracks are truly synchronized, you probably WILL find a header next). Now we do something that we didn't do in our previous routines. We save our error code on the stack and call on a DOS routine to convert the GCR into hex. This puts the decoded data back into our storage buffer at $0400 (note that it will only occupy the first 8 bytes of the buffer now). Finally we terminate as usual.

Based on what you find in the buffer, you may want to adjust the step delay. For instance, if you are consistently landing on a data block, you could reduce the delay a little to pick up the preceding header too. Or you could increase the delay to pick up the following header. As with all our routines, this routine is a starting point for you to experiment with.

EXTRA SECTORS

It is possible to add an extra sector to tracks 18-24. The chapter on the standard 1541 format explains how the drive stores a sector. Here is a quick summary of the size of a 'normal' sector:

INFORMATION	# OF BYTES HEX	GCR	x8= # BITS	
HEADER SYNC		5	40	
HDR BLOCK	8	10	80	
HDR GAP		8	64	
DATA SYNC		5	40	
DATA BLOCK	260	325	2600	
TAIL GAP		8	64	(typical)
TOTAL		361	2888	

Standard number of bytes/track:	7142
- 361 bytes/sector x 19 sectors :	- 6859
= Extra bytes per track	= 283

The only variable factor in the above setup is the length of the tail gap after the data block. When formatting a disk, the DOS actually 'calibrates' itself to the speed of your drive. It starts on each track by writing a long section of sync marks ($FF's) and then a section of contrasting bytes ($55's). By reading these bytes back and counting the length of the two sections, it can calculate how many bytes will actually fit on that track, assuming the drive speed stays exactly the same, which it doesn't. On a 'perfect' drive (at a constant 300 RPM) it will be able to get 7142 bytes on tracks 18-24. Since this is more than it needs for the sectors on the track, it splits these extra bytes among the tail gaps of the sectors. Typically, it might put 8 bytes in each tail gap on tracks 18-24 (it will make the gap after the last sector on the track even longer).

This gives us a typical sector length on these tracks of 361 GCR bytes (see table above). Multiplying 361 times the number of sectors normally on the track (19) gives us a total of 6859 bytes needed. This leaves 283 bytes leftover. Unfortunately, this is not enough to do a whole extra sector (361 bytes). There are two possible ways around this problem if we want to squeeze an extra sector on the track. We can reduce the size of the tail gaps or slow down the speed of the drive (Or both). Let's look at reducing the tail gap size first.

The DOS has a table built into it containing the number of sectors on each track of the disk. We can change the preset number on tracks 18-24 to 20 instead of 19 by burning a new ROM. When the DOS tries to format the track, it will proceed as follows. Since we now need 20 sectors instead of the normal 19, and each needs a minimum of 353 bytes (not counting the tail gap) we will need at least 7060 bytes. There are 7142 bytes available on the track (ideally), so we have 82 bytes left over to split among the tail gaps. Since there are 20 tail gaps, this leaves 4 bytes for each, with a whopping two bytes left to spare (which will make the last tail gap that much longer than the others). Thus each sector will now take 353+4 = 357 bytes. This is right at the limit that the DOS format routine will allow. In fact, it will not put less than 4 bytes in a tail gap of its own accord when formatting (it will signal an error instead). If this worries us (it shouldn't) we could get some more breathing room with our other option, reducing the speed of the drive.

If we reduce the drive speed (by adjusting a potentiometer in the drive), each revolution will take a little longer. Since bytes are clocked out at a constant amount of TIME per byte, more time per revolution means more bytes per track. Since we have 283 bytes left over on tracks 18-24 (see the chart above), but we need 361 bytes to fit in another whole sector, we have to pick up 78 bytes. If we slow the drive down about 1.1% when formatting, we can manage this. This will put the bits slightly closer together on the track than normal, but the diskette will be able to handle it. The bits will come in a little faster when reading at normal speed (with 'READ GCR', for instance), but we saw in the density section above that this is no problem. In fact, reducing the drive speed amounts to increasing the density. The change in density is much smaller than you get by going to another clock rate, however. Thus by reducing the drive speed we can fine-tune the density more flexibly than possible otherwise.

We will also be able to write to the disk at normal speed without trouble. Since the disk is turning a little faster now than it was when formatted, the sector will take up slightly more space when rewritten than it did originally, by about 1% or 4 bytes. This will reduce our tail gap from 8 bytes to 4 (in theory), but that is enough room. We are not in danger of writing over the sync mark of the next sector. In fact, a sync mark only needs to be 10 BITS long, and the DOS normally writes 40 bits (5 $FF bytes). Thus we could even overlap some of the normal sync and get away with it. This might happen on a drive that was turning faster than normal.

There is one other way to get an extra sector on these tracks - let a 2040 or 3040 drive do it for us! These old Commodore drives were set for 20 sectors on tracks 18-24, and they are read-compatible with the 1541.

MODIFIED FORMATS

Another type of nonstandard format that can be used on a disk is to change the information that is written on a track rather than the way it is written. This includes altering the information in the sector headers as well as varying the number of gap bytes and their values, using custom sync marks, etc. Altering the sector header information can result in mis-numbered sectors of various types as well as some of the regular 'bad block' errors. These types of changes are 'softer' errors than changing density, adding extra sectors and half-tracking.

At this point you should look back to the chapter on standard 1541 format and review the standard sector setup. We'll start our discussion of these modified formats with altered sector header information. Any of the sector header bytes can be changed to yield a protection method. A normal read will yield an error in most cases, but the header is easily read by a custom DOS routine. You can just as easily read a data block after it. By combining more than one change you can come up with virtual do-it-yourself headers.

Let's start with the first byte of the header, the header block identifier (not to be confused with the disk ID). This byte is always $08 (hex, not GCR) on a standard disk to distinguish header blocks from data blocks, which start with $07. If some other value is used instead, this will fool the disk into giving an error 20 (header block not found) on a normal attempt to read the sector. A custom DOS routine can simply wait for a sync mark to finish and then check to see if the next byte is the special value it expects. If not, the routine can try again. If it doesn't find the special identifier after a certain number of attempts, it can assume the disk is a copy. Another way to use this would be to read a particular normal sector, and then expect the next sector header after it to contain the nonstandard identifier.

The second byte of the header is the header block checksum. This is simply an EOR of the following four bytes (sector and track numbers and disk ID bytes). Changing this value will result in an error 27 (checksum error in header block) on a normal read. Again, a custom routine can find the header on its own and verify not only that there is an error 27, but also that the checksum contains the particular value it is looking for, without ever 'bump'ing the head.

The next two bytes are the sector and track numbers, in that order. By manipulating these bytes we can make all sorts of strange things appear on the disk, such as: duplicate sector numbers, sectors numbered out of order (displaced sectors), incorrect track numbers (not the same as the track they are on) and illegal track and sector numbers (values outside the normal range). These seem to be popular currently, but many copy programs can already handle them easily. The errors yielded by

these changes range from none at all (duplicate or out-of-order sectors) to errors 20 and 27 (since they are included in the header checksum).

There is an interesting story connected with the duplicated sector idea. We know of one company which paid $5000 for the rights to a protection scheme based on duplicating sectors 0-10 on a track twice! At that moment in time there were no copy programs that could handle this scheme. The protected software had barely hit the market when suddenly there appeared several copy programs that could handle it easily. Looks like somebody sure wasted their money!

Back to our sector header. The two bytes after the sector and track numbers are the disk ID bytes, in reverse order (ID2 then ID1). Changing these bytes will give an error 29 (disk ID mismatch) normally, although an error 20 is possible in certain operations. These two bytes can be used in much the same way as the preceding ones; e.g. we can check not only for an error 29 but for the actual value of the incorrect bytes.

The last two bytes of the header are just filler bytes, value $0F (hex). These are used to pad out the header block to eight hex bytes so it can be handled conveniently by the DOS ROM 4-byte hex to 5-byte GCR conversion routine at $F7E6. Normally these have a value of $0F but since the DOS never examines them again after formatting, they can be set to any value without showing up as an error.

The DOS formats a track by preparing all the headers for the track in advance and converting them to GCR. It also creates a dummy data block in GCR. It then goes around the track, writing a header sync, block and gap, followed by a data sync, block and tail gap (previously calculated as explained in EXTRA SECTORS). If you patch into the format routine or write your own, you can easily write any sort of headers you wish on a track, as well as changing gap and sync byte values, varying gap lengths and creating extra long header and data blocks. Reading this information is easy with the 'READ GCR 1K' program we looked at earlier.

On the program disk is another program called 'READ HEADERS'. Figure DOS-7 lists the program code. This program allows you to look at all the headers on a track, decoded into HEX for ease of reference. You'll need DRVMON to examine the results conveniently.

When you execute the routine with 'E0' it moves the head to the track you specify and begins reading headers (minus syncs and gaps) into a storage buffer at $0400. For simplicity it only takes those blocks with a valid header identifier byte ($52 GCR, $08 hex). It reads in 25 headers. Normal tracks contain a maximum of 21 headers, so there will be a duplication of the first few headers at the end of the buffer. This feature gives the routine the ability to detect extra sectors on the track.

After reading in the 25 headers, it converts the entire buffer from GCR to hex in one shot by calling on a DOS ROM routine. This replaces the hex bytes into the buffer, where you can examine them with DRVMON.

Referring to figure DOS-7, we see that the routine begins in the same way as our previous examples. It finds a sync mark and then at $033D checks the next byte to see if it is a normal header block identifier byte ($52 GCR). If not, the routine goes back and waits for the next sync mark. Once it has found a valid header identifier, it reads that byte and the next nine bytes into the buffer, indexed by the Y-register (header blocks consist of 10 GCR bytes, which convert to 8 hex bytes). Next it checks at $0357 to see if 250 ($FA) bytes have been read into the buffer yet. This value represents 25 headers at 10 bytes each. If there is more to do, the routine branches back up to look for a sync and starts the whole process over again (it saves the Y-register in $03FF before branching because it is used by the FINDSYNC code).

Once the buffer has been filled with headers, the no-error code ($01) is saved on the stack before the GCR-HEX conversion routine is called. First it sets a DOS pointer at $30-31 to point to the storage buffer at $0400, and then jumps to the conversion routine which is at $F8E0. This routine puts the converted bytes back into the buffer at $0400-04FF, replacing the GCR version. Upon returning, the error code is retreived from the stack and the routine is terminated.

Use an M 0400 command from DRVMON to examine the buffer after this routine has terminated. Since the M command displays memory 8 bytes at a time, each line will contain exactly one header. However, the conversion routine strips off the very first byte of the buffer, so each line starts with the second header byte (checksum). At the end of the line you will see the header identifier ($08) from the next header. Here's an example of what you might see:

```
:0400 0A 00 23 44 49 0F 0F 08  ..#DI..
:0410 89 01 23 44 49 0F 0F 08  ..#DI..
```

From the second and third bytes you can see that these are the headers for sectors 0 and 1 from track 35 ($23), and the disk ID is 'ID'. The checksums in the first bytes of each line are not correct, so that you'll get an error 27 on these sectors with a normal read.

FIGURE DOS-7: READ HEADERS

```
                                *=   $0300          ;ROUTINE TO HEADERS V3
1080:  0300 78                  SEI             ;WILL READ 25 HEADERS AND DECODE
1090:  0301 A0 00               LDY  #$00           ;FILL WORK SPACE WITH 00
1100:  0303 A9 00               LDA  #$00
1110:  0305           CLEARIT   =    *
1120:  0305 99 00 04            STA  $0400,Y        ;STORE 00 AT $0400-$04FF
1130:  0308 C8                  INY
1140:  0309 D0 FA               BNE  CLEARIT
1150:  030B 20 00 FE            JSR  $FE00          ;SET PCR TO READ MODE
1160:  030E AD 0C 1C            LDA  $1C0C
1170:  0311 09 0E               ORA  #$0E
1180:  0313 8D 0C 1C            STA  $1C0C
1190:  0316 A9 00               LDA  #$00           ;SET COUNTER FOR THE # OF SECTORS
1200:  0318 8D FF 03            STA  $03FF          ;COUNTER
1210:  031B           FINDSYNC  =    *
1220:  031B A2 00               LDX  #$00           ;SET UP TIMER FOR SYNC
1230:  031D A0 00               LDY  #$00
1240:  031F           TIMEOUT   =    *
1250:  031F 88                  DEY
1260:  0320 D0 07               BNE  WAITSYNC
1270:  0322 CA                  DEX
1280:  0323 D0 04               BNE  WAITSYNC
1290:  0325 A9 03               LDA  #$03           ;03=NO SYNC   IF NO SYNC THEN END
1300:  0327 D0 34               BNE  ENDIT
1310:  0329           WAITSYNC  =    *          ;CHECK FOR SYNC
1320:  0329 2C 00 1C            BIT  $1C00
1330:  032C 30 F1               BMI  TIMEOUT
1340:  032E AD 01 1C            LDA  $1C01          ;SKIP FIRST BYTE
1350:  0331 B8                  CLV
1360:  0332 AC FF 03            LDY  $03FF
1370:  0335           READHDER  =    *          ;FIND A HEADER
1380:  0335 A2 0A               LDX  #$0A           ;COUNTER FOR TEN HEADER BYTES
1390:  0337           FINDHDR   =    *          ;READ DATA FROM DISK
1400:  0337 50 FE               BVC  FINDHDR        ;WAIT FOR BYTE READY
1410:  0339 B8                  CLV
1420:  033A AD 01 1C            LDA  $1C01          ;LOAD BYTE FROM DATA PORT
1430:  033D C9 52               CMP  #$52           ;IS THIS A HEADER BLOCK
1440:  033F D0 DA               BNE  FINDSYNC       ;IF NOT, FIND NEXT HEADER BLOCK
1450:  0341 99 00 04            STA  $0400,Y
1460:  0344 C8                  INY
1470:  0345 CA                  DEX
1480:  0346           GETBYTE   =    *          ;READ DATA FROM DISK
1490:  0346 50 FE               BVC  GETBYTE        ;WAIT FOR BYTE READY
1500:  0348 B8                  CLV
1510:  0349 AD 01 1C            LDA  $1C01          ;LOAD BYTE FROM DATA PORT
1520:  034C 99 00 04            STA  $0400,Y        ;STORE DATA FROM $0400-$04FF
1530:  034F C8                  INY
1540:  0350 CA                  DEX
1550:  0351 D0 F3               BNE  GETBYTE
1560:  0353 8C FF 03            STY  $03FF
1570:  0356 38                  SEC
1580:  0357 C0 FA               CPY  #$FA           ;DONE 25 SECTORS YET
1590:  0359 90 C0               BCC  FINDSYNC       ;IF NOT DO AGAIN
1600:  035B A9 01               LDA  #$01           ;01=NO ERROR
1610:  035D           ENDIT     =    *            ;FINISH UP AND END
1620:  035D 48                  PHA             ;SAVE ERROR CODE IN STACK
1630:  035E A0 04               LDY  #$04            ;SET UUP TO DECODE GCR INTO HEX
1640:  0360 84 31               STY  $31             ;POINTERS TO BLOCK TO BE DECODED
1650:  0362 A0 00               LDY  #$00
1660:  0364 84 30               STY  $30
1670:  0366 20 E0 F8            JSR  $F8E0          ;ROM ROUTINE TO DECODE GCR
1680:  0369 68                  PLA             ;GET ERROR CODE FROM STACK
1690:  036A 4C 69 F9            JMP  $F969
```

Other types of modified format besides altered sector header information are possible. The number of header gap bytes (normally 8) can be changed or a nonstandard value used (gap bytes are normally $55 GCR). The data block format can also be altered. Its sync mark could be missing or replaced by a custom value. The data block identifier could be changed from the normal $55 GCR ($07 hex) to cause an error 22. The data block checksum could be wrong (error 23). The OFF bytes ($00) used to pad out the data block could be replaced with another value. The tail gap bytes can be replaced with a custom value or varied from sector to sector. These types of changes can all be detected with the 'READ GCR 1K' routine presented earlier.

Using the tools in this chapter, you can examine a disk on the most fundamental level to see what type of protection has been placed on it. You may also see routines similar to these in a protected program. Even better, you can use these routines or modified versions of them as part of your own protection scheme. Once the routine has looked for the special format you've used, a simple 'M-R' memory read command can read the drive memory to determine the result. These routines also give you a glimpse into the internal workings of the disk drive, to get you started on your own investigations. We hope that you have found this chapter useful and illuminating.

GENERAL NOTES

1) THESE ROUTINES ARE DESIGNED TO WORK ONLY ON A 1541 DISK DRIVE. OTHER DRIVES MAY REQUIRE MODIFICATION TO THE ROUTINE.

2) ALWAYS USE A DISK FORMATTED ON A 1541 DRIVE FOR YOUR INVESTIGATIONS. THE RESULTS WILL BE MORE PREDICTABLE.

3) ALWAYS INITIALIZE THE DRIVE WITH THE 'IØ' COMMAND PRIOR TO EXECUTING A CUSTOM DOS ROUTINE (do this faithfully). It is also necessary to do a 'UI' or 'UK' (RESET) command BEFORE the 'IØ' with HALF TRACK, SYNCHRONIZED and TRACK ARCING. These routines leave the drive in a nonstandard state (i.e. on the half track). Prior to performing the routine for a second time you should do a UI (or UK) then an IO before execution of another operation. If you don't use the above commands it may be necessary to have your custom DOS routine take over complete control of the head stepping.

4) THE RESULTS OF A ROUTINE CAN BE AFFECTED BY THE MECHANICAL AND ELECTRICAL CONDITION OF YOUR DISK DRIVE AND DISKETTE. If these routines are to be used as part of a protection scheme, you must test them on as many drives as possible to see what (if any) variation you can expect 'in the field'.

5) CHECK THE JOB ERROR CODE RETURNED IN LOCATION $0000 BEFORE EXAMINING THE RESULTS. A value of $01 indicates no error. Any other value may reflect trouble with a particular routine and/or disk.

6) GIVE THE ROUTINES ENOUGH TIME TO EXECUTE BEFORE CHECKING THE RESULTS. Certain routines take a perceptible amount of time, e.g. READ HEADERS.

7) DRVMON 'LOCKS UP' OCCASIONALLY FOR REASONS UNKNOWN. Usually it is necessary to reset the drive and sometimes the computer too. Occasionally it is necessary to power-off to recover (the drive's memory will be wiped out in any case). Also, DRVMON sometimes puts '00, 0K, 00, 00' into the storage buffer ($0400-$04FF); if this happens just re-execute the routine and try again.

8) All the routines include a check to find a sync mark on a particular track. This check is timed, so if there is not a sync mark on the track the routine will not search forever. If the routine 'times out' prior to finding a sync mark an error will be reported in the job queue and the routine will be terminated.

HALF TRACK

Requires a UI then an IØ before execution. Also, you may get what seems like valid data on a half track from 'bleed-over' from an adjacent whole track. The amount of bleed-over depends upon the physical and electrical condition of the disk drive used to create the disk and upon the condition of the drive used to read it. After this routine has executed and the data has been read from the drive you should perform a UI and IØ command to reset the drive.

NYBBLE

Use a 1541-formatted disk only. This routine is preset for tracks 1-17 density; it must be altered (see text) for other tracks.

DENSITY

In the text we mention that it is possible to read data that was written at different densities. This is true only for small amounts of data. The more data read and/or the greater the difference between density, the greater the chance of error when reading the data. So, if you are going to read data from a disk, it is important to know at what density it was written. Otherwise the data read may not be reliable.

SYNCHRONIZED/TRACK ARCING

Requires a UI then an IØ before execution. TRACK ARCING is an additional program not mentioned in the text. It is identical to SYNCHRONIZED except it only steps out half a track. Synchronized tracks, Track arcing and Spiral tracking are all related to one another. The only variation of these routines is how far to step and when to step the head. After this routine has executed and the data has been read from the drive you should perform a UI command to reset the drive.

READ HEADERS

Give it time to work (1-2 seconds) before you try to read data from the buffer. This routine will decode the data read from the disk (from GCR to hex). The decoding is part of the routine and will be done automatically for you.

The Commodore 64 contains the 6510 microprocessor, a new member of the 65xx family. The 6510 contains 6 I/O lines which it uses for external communication. The I/O lines and a chip called the address manager (PLA) enable the 6510 to perform memory bank selection. The 64 contains 64K of RAM, 20K of ROM, 4K of I/O devices and 8K of character ROM, as well as the ability to directly control up to 16K of external ROM. Since the 6510 can communicate with only 64K of memory at a time, it uses bank selection to select certain banks or areas of memory.

The Commodore 64's memory reconfiguration system can be controlled externally by the hardware configuration of a cartridge. The memory configuration necessary to accommodate cartridges is controlled by two of the six I/O lines. These two lines are the EXROM and the GAME lines. Normally, these lines are at logic one (+5 volts). When a cartridge is inserted either one or both of these lines will be set to logic zero (grounded, 0 volts). This results in a hardware reconfiguration of the 64's memory. The following four figures depict the possible memory schemes.

FIGURE 1

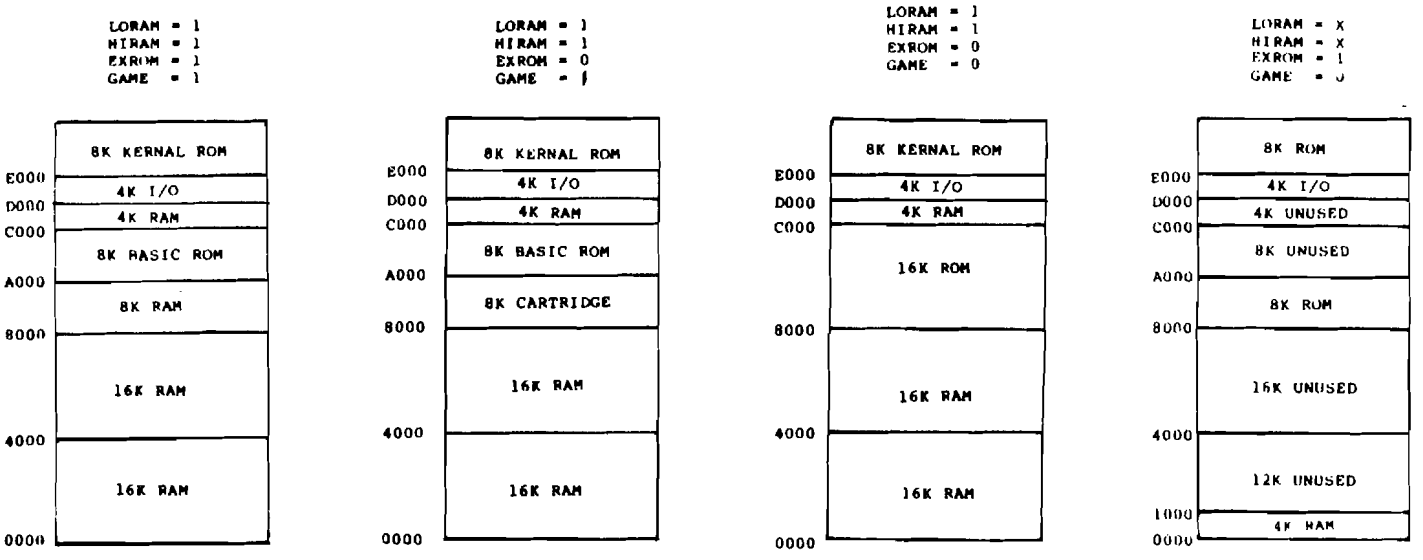


Figure 1a shows the configuration which appears when the 64 is turned on. The EXROM and GAME lines are at logic one giving the computer the familiar 38911 bytes of free RAM. Figure 1b shows the standard BASIC system with an 8K expansion cartridge. This memory arrangement is achieved by grounding (logic zero) the EXROM line. The expansion cartridge can either be an extension of BASIC, or a video game. With the exception of Commodore game cartridges, very few game cartridges are only 8K. Figure 1c depicts the memory

configuration when both the EXROM and GAME lines are grounded. This is the result of inserting a 16K cartridge, which is the most common of the four memory reconfigurations. It shows ROM area available from hex $8000 to $BFFF. Finally, figure 1d shows the memory reconfiguration resulting from a very special type of cartridge. This cartridge was originally manufactured by Commodore for the ULTIMAX computer. Rearrangement of the 64's memory enabled the cartridges to also be used with the 64. The cartridge grounds the GAME line which switches out the KERNAL ROM. The KERNAL is replaced by the 8K ROM of the cartridge. A more in depth explanation of these cartridges and how to identify them can be found in the next chapter.

The Commodore 64 searches for a special character sequence from $8004 to $8008 (CBM80) to determine if there is an autostart routine present in the cartridge area. The 64 checks these five bytes every time the computer is turned on or reset. When the computer executes its cold-start KERNAL routines it runs the code at $FCE2. The following sections of code are executed by the 64:

```
FCE2  A2 FF     LDX #$FF     :RESET ROUTINE
FCE4  78        SEI
FCE5  9A        TXS
FCE6  D8        CLD
FCE7  20 02 FD  JSR $FD02    :CHECKS CBM80
FCEA  D0 03     BNE $FCEF
FCEC  6C 00 80  JMP ($8000)  :AUTO-START
FCEF  8E 16 D0  STX $D016
FCF2  20 A3 FD  JSR $FDA3
FCF5  20 50 FD  JSR $FD50
FCF8  20 15 FD  JSR $FD15
FCFE  58        CLI          :BASIC
FCFF  6C 00 A0  JMP ($A000)  :cold-start
.
.
.
FD02  A2 05     LDX #$05
FD04  BD 0F FD  LDA $FD0F,X  :THIS ROUTINE
FD07  DD 03 80  CMP $8003,X  :CHECKS CBM80
FD0A  D0 03     BNE $FD0F    :(WORKING
FD0C  CA        DEX          :BACKWARDS)
FD0D  D0 F5     BNE $FD04
FD0F  60        RTS
.
.
.
FD10  C3        'C'  :THE ROUTINE AT $FD02
FD11  C2        'B'  :CHECKS THIS AND
FD12  CD        'M'  :JUMPS TO THE ADDRESS
FD13  38        '8'  :AT $8000 AND $8001
FD14  30        '0'  :IF THE VALUES MATCH
```

If there is a 'CBM80' at $8004, the computer jumps to

the address contained in $8000 and $8001. This is called the cold-start vector. The following example will help to illustrate:

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

In the above example, the cold-start address would be $8009. Remember, the vector is stored LO-BYTE, HI-BYTE (reverse) order. The next two bytes contain the warm-start vector. In our example this would be $8070. The warm-start address is used when the 'RESTORE' key is pressed. Both of these vectors point to the area of the program which must be executed depending on the state of the computer. Sometimes these addresses are identical, but most of the time they are different. The next five bytes are the shifted letters 'CBM80'. The auto-start sequence will work whether the routine (CBM80) is in ROM or RAM. If you used a monitor to place the example into memory and then reset the computer, the computer would attempt to execute code at $8009. Now that we've covered the fundamental operation of cartridges it's time to examine some of the coding techniques which are used to protect cartridges.

Before we continue, you must be aware of a number of assumptions we'll make about the individual attempting to copy cartridges (that's you). First, you must have a switchable expander board, preferrably one that has LED's and a reset button. Currently, there are two boards ideally suited for copying cartridges. The first is the Cartridge Backer (CB) board which is available from CSM SOFTWARE INC. This board can be purchased with the Cartridge Backer software for $54.95, or separately for $24.95. The second board is Cardco's Cardboard/5 which retails for $79.95. Either of these boards have the capability to bank-select the three cartridge configurations. They also have cartridge ENABLE and power switches which allow cartridges to be inserted or removed while the computer is on. Finally, both have a reset button and LED's. There are two LED's used to indicate cartridge size and type. The LED's on the CB board are red, and the LED's on the Cardco board are yellow. Throughout the rest of the chapter, references will be made to the use of the boards. If you have another type of board, you must determine the cartridge size and type in another manner.

Next, it is assumed that you are familiar with the use of a ML monitor. Most of the time, you should be using HIMON, but sometimes LOMON will be used because not all types of cartridges reside at $8000. Don't use HESMON if you are doing 16K cartridges since it uses the BASIC ROM. The following section is a brief review of the commands you will be using.

There are seven commands which will be used to copy cartridges. They are: S, M, I, D, H, C and T.

The 'S' command is used to save sections of code. For example, if you wished to save a 16k cartridge you would do the following:

S'name',08,8000,C000

The 'M' command displays the hexadecimal contents of any address or range of addresses in memory. For example, to examine address 0001 you would do the following:

M 0001

This would display the next 8 bytes of memory starting at address 0001.

The 'I' command is used to interrogate memory locations. Interrogation converts the hexadecimal memory contents to their ASCII equivalent and display them on the screen. You will use the 'I' command to check for the 'CBM80' at $8000. For example:

I 8000

If an auto-start cartridge was inserted, you would see the letters 'CBM80' from addresses $8004-$8008.

The 'D' command is used to disassemble the contents of a series of memory locations. This command displays the mnemonic equivalent of the hexadecimal contents at a given location. For example:

D 8000 8100

This would disassemble the memory contents from $8000-$8100.

The 'H' command is used to locate a byte or series of bytes within a specified memory range. For example:

H 8000 C000 85 01

This would search the address range from $8000-$C000 for the byte sequence 85 01. If there was a match, the address of the match would be printed on the screen.

The 'C' command is used to compare the contents of two sections of memory. For example:

C 8000 BFFF 2000

This would compare the contents of the address range from $8000-$BFFF against the contents of address range $2000-$5FFF. The address of any mismatch is printed on the screen.

Finally, the 'T' command is used to transfer sections of memory from one part of the computer to another. For example:

T 8000 A000 2000

This would transfer 16k of code from $8000-$BFFF to $2000-$5FFF.

The last assumption is that you are familiar with hexadecimal arithmetic, and have a means to convert numbers from decimal to hexadecimal to binary. This is especially important when trying to determine the effect of storing numbers at address 0001. It is suggested you purchase a scientific calculator capable of arithmetic conversion. A good calculator at a reasonable price is the Casio FX-450. This calculator is capable of arithmetic conversion of all three number bases and only costs $25.00.

Now that we have covered the basic tools and skills necessary to copy cartridges, it's time to examine some cartridges.

We'll start with an 8K cartridge. A cartridge is 8K if only the right LED is lit. Remember, on the Cardco 5, there are 4 LED's per slot. You should only be concerned with the two yellow LED'S. Insert your expander board into the computer and turn on the computer. On the CB board, the switches should be set as follows: sw1, sw2, and sw4 should be OFF; sw3 should be ON. On the Cardco board, you have two master switches for the whole board and two for each of the 5 slots. We'll call the master switches sw1 (left) and sw2 (right). The switches for the slot your cartridge is plugged into will be called sw3 (l) and sw4 (r). All the switches should be OFF on the Cardco 5.

The above switch settings should give you 38911 BYTES FREE on the main screen. Next, load HIMON and SYS 49152 to activate it. Insert the cartridge into the slot and check the LED's. If only the left LED is lit, go to the section on MAX cartridges. If both LED's are lit, you have a 16K cartridge. 16K cartridges are similar to 8K except they reside from $8000-$BFFF in the computer's memory. 16K cartridges will be covered later, but be sure to read the section on 8K cartridges or else you won't understand how to copy a 16K cartridge. If only the right LED is lit, you have an 8K cartridge which resides in the computer from $8000-$9FFF. With the cartridge inserted in the slot, enable the cartridge by turning ON sw4 and turning OFF sw3 on the CB board, and turning ON sw4 and sw3 on the Cardco 5. The first thing which must be done is to transfer the contents of the ROM cartridge to the RAM underneath. Use the 'T' command to do this:

T 8000 9FFF 8000

After the transfer is complete, disable the cartridge by reversing the positions of sw3 and sw4 (both boards). The contents of the cartridge must be copied to RAM so it can be altered. If you try to alter the program while it is still in ROM, you will get a '?' from the monitor because it was unable to execute the command. Next, save the cartridge using

the 'S' command:

S 'program name',08,8000,A000

Remember, you must save from the starting address to the ending address plus one. Now examine the 5 bytes from $8004-$8008 using the 'I' command. Make sure there is a 'CBM80' at this location. If there is a 'CBM80', try executing the program by typing G FCE2 (software RESET) from the monitor. Occasionally, a cartridge will run without any modification, but most of the time you must make changes to the program so it can run in RAM when loaded from the disk.

There are 5 types of conditions which prevent a cartridge from running when it is in RAM. They are:

1. KERNAL routines (initialization)
2. BASIC ROM switch-ins (16K cartridges only)
3. Stores to cartridge area (8K:$8000-$9FFF, 16K:$8000-$BFFF)
4. Vectors in ZERO PAGE (program checks for particular values)
5. CIA TIMER 'A' running (location DC0E nonzero)

The first type of routine which must be removed from almost all cartridges are the KERNAL initialization routines. These routines are not actually intended as protection, but they can prevent the cartridge from running when it is in RAM. The routines are used in a cartridge because the computer must be initialized when it is first turned-on. There are 8 commonly used KERNAL routine:

HEX CODE	NMEMONIC	PURPOSE
20 81 FF	JSR $FF81	initialize video
20 5B FF	JSR $FF5B	"
20 84 FF	JSR $FF84	initialize TIMERS
20 A3 FF	JSR $FFA3	"
20 87 FF	JSR $FF87	clear RAM
20 50 FD	JSR $FF50	"
20 8A FF	JSR $FF8A	initialize I/O
20 15 FD	JSR $FD15	"

If the cartridge you copied didn't run when the computer was RESET, you must examine the code to determine the problem. After you have RESET the computer and the cartridge 'crashed', you must flip-out the RAM to disable the program. Do this by turning ON the EXROM switch (on both boards) and press the RESET button. You should regain control of the computer with 30719 BYTES FREE. Turn OFF the EXROM switch,

and SYS 49152 to activate the ML monitor. Some cartridges overwrite the monitor area and the monitor must be reloaded.

The first thing which must be done is to transfer the code from $8000 to $2000.

```
T 8000 9FFF 2000
```

This moves the 'crashed' version down to $2000 so it can be compared with the original version. Next, load the original version you saved to disk:

```
L"program name",08
```

This will load the original code into the computer at $8000 (the place from where it was saved). With the 'crashed version at $2000, and the original at $8000, it is now possible to compare the two to check for differences. Use the 'C' command to compare the two sections of code:

```
C 2000 3FFF 8000
```

The 'crashed' version will be compared to the original byte-for-byte and any differences will be reported on the screen. The first byte ($2000) will be reported as being different because the computer places a $55 at address $8000 when it is undergoing a RESET. If no other addresses are reported as being different, the cause of the 'crash' is more than likely due to the KERNAL initialization routines. Most cartridges will operate properly once the KERNAL routine calls are removed. Remember, this is only true for 8K cartridges. With a 16K cartridge, a BASIC switch-in can cause a 'crash' without altering any code in the program (more on this later). Assuming the cause of the 'crash' is due to the KERNAL routines, you can use the 'H' command to search for references to them. For example:

```
H 8000 9FFF 20 81 FF
```

This will search from $8000-$9FFF for the KERNAL routine $FF81, and report the address at which the routine is used if it is present. (Remember, make all changes to the original version of the program at $8000, not the version at $2000. The changes should be made to the version at $8000 so the program runs in the correct area when reloaded into the computer).The same procedure should be repeated for all 8 of the KERNAL routine references. If there were a KERNAL routine present in the program it must be replaced with NOP's ($EA). This instruction means No OPeration. FIGURE 2 is an example of a cartridge which 'crashed' because of KERNAL routines.

FIGURE 2

```
8E72  20 84 FF JSR $FF84 :KERNALS
8E75  20 87 FF JSR $FF87 :   ''
8E78  20 8A FF JSR $FF8A :   ''
8E7B  20 81 FF JSR $FF81 :   ''
8E7E  78       SEI
8E7F  A9 7F    LDA #$7F
8E81  8D 18 03 STA $0318
8E84  A9 97    LDA #$97
8E86  8D 19 03 STA #$0319
8E89  A9 00    LDA #$00
```

If you found the code shown in FIGURE 2, you would use the 'M' command to display the code from $8E72-$8E7D. Change the bytes by placing the cursor over byte and change then to $EA. At the end of each row of bytes you must press the 'RETURN' key to actually make the changes. The altered code should look like FIGURE 3:

FIGURE 3

```
8E72  EA       NOP
8E73  EA       NOP
8E74  EA       NOP
8E75  EA       NOP
8E76  EA       NOP
8E77  EA       NOP
8E78  EA       NOP
8E79  EA       NOP
8E7A  EA       NOP
8E7B  EA       NOP
8E7C  EA       NOP
8E7D  EA       NOP
8E7E  78       SEI
8E7F  A9 7F    LDA #$7F
8E81  8D 18 03 STA $0318
8E84  A9 97    LDA #$97
8E86  8D 19 03 STA #$0319
8E89  A9 00    LDA #$00
```

The altered code should now be saved to disk:

```
S "program name",08,8000,A000
```

After the code is saved, try running the program by typing G FCE2 from the ML monitor. If the cartridge was only 8K, there were no addresses reported different on comparison,and you correctly changed the KERNAL routines, the cartridge should run.

The previous example was extremely simple. If there had been an address reported when you compared the 'crashed' version with the original, the job of copying the cartridge becomes much more involved. There are a number addressing methods a cartridge can use to attempt to 'write' to itself. These include: direct or absolute; indexed; indirect,indexed;

and indexed,indirect. For the rest of the manual, with the exception of the MAX cartridges, all cartridges will be assumed to be 16K. The rest of this chapter will cover the BASIC ROM switch-in. The next chapter will cover the addressing methods for 'writes', vectors at ZERO page, CIA TIMERS, and MAX cartridges.

When a 16K cartridge is inserted, both the EXROM and GAME lines are grounded (logic zero) which makes the area from $8000-$C000 available for cartridge ROM. It is also possible to reconfigure the computer's memory using address $0001. The two most common methods to protect a 16K cartridge are KERNAL routines (previously covered) and BASIC switch-ins. The BASIC ROM resides from $A000-C000 and is switched-in over the top half of a 16K cartridge. The first 3 bits of address $0001 are used to control the memory configurations. FIGURE 4 shows the values used to reconfigure the computer's memory (also see pp. 260-267 of the Programmer's Reference Guide).

FIGURE 4

VALUE at ADDRESS $0001

HEX	BINARY	DEC	CONFIGURATION
37	00110111	55	NORMAL
36	00110110	54	BASIC-out
35	00110101	53	KERNAL and BASIC out
34	00110100	52	64K RAM (all out)
33	00110011	51	I/O out, BASIC/KERNAL in
32	00110010	50	I/O and BASIC out
31	00110001	49	I/O and KERNAL out
30	00110000	48	64K RAM (all out)

The most common result of a BASIC switch-in is the computer returning to 'READY' when the program is run.

A 16K cartridge is copied in the same basic manner as an 8K. The only difference is that the BASIC ROM must be switched out before you save the cartridge. The BASIC ROM can be switched out using the 'M' command:

```
M 0001
```

The normal value at address $0001 is $37 ($07 on the SX-64). This must be changed to $36 to switch-out the BASIC ROM. Once this is done, you can save the code from $8000-$C000. FIGURE 5 is an example of a cartridge which

stores a value at address $0001 to switch in the BASIC ROM.

FIGURE 5

```
BE36  A5 01    LDA $01
BE38  29 FB    ORA #$01  :BASIC ROM on
BE3A  85 01    STA $01
.
.
.
BE57  A5 01    LDA $01
BE59  09 04    AND #$FB  :CHAR ROM on
BE5B  85 01    STA $01
```

When working with address $0001, you must remember that not every store to address $0001 is detrimental to the operation of the cartridge. (This is why it is important to have a calculator to convert the hex numbers to binary so you can see which bits are being changed). In FIGURE 5, only the first store to address $0001 will switch in the BASIC ROM and 'crash' the cartridge. There are two methods which can be used to fix this type of protection. The first is to NOP addresses $BE3A and $BE3B, removing the STA $01(store the accumulator at address $0001). This method of repair is very unreliable because it can prevent other needed ROMs in the computer from being switched in or out. The second method is the suggested method of repair. This method requires altering the value which is AND'ed or ORA'ed with the accumulator. Changing the value at $BE39 from $FB to $FA results in the BASIC ROM remaining switched out and the other ROMs left unaltered. The second store to address $0001 doesn't affect the cartridge area. The second example switches-in the character set. In order for the cartridge to run correctly, the second store to address $0001 must be left unaltered. Only change those 'stores' to address $0001 which attempt to switch-in the BASIC ROM. A quick way to search for 'stores' to address $0001 is to use the 'H' command of the ML monitor. FIGURE 6 is a list of possible BASIC switch-ins.

FIGURE 6

```
85 01     STA $01
84 01     STY $01
86 01     STX $01

8D 01 00  STA $0001
8C 01 00  STA $0001
8E 01 00  STA $0001

99 01 00  STA $0001,Y
9D 01 00  STY $0001,X
```

If you suspect a BASIC switch-in, search the code from $8000-$C000 for a 'store' to address $0001. For example:

```
H 8000 C000 85 01
```

Remember, after loading a 16K cartridge the BASIC ROM must be switched out before examining the code from $A000-$C000. Use the 'M' command to switch out the BASIC ROM. After locating a store to address $0001, it must be NOP'ed or the value must be changed. FIGURE 7 is a list of values which can be used to enable or disable the different combinations of ROMs.

FIGURE 7

OR (enable)	AND (disable)	CONFIGURATION (disabled)
01	FE	BASIC out
02	FD	BASIC & KERNAL out
03	FC	64K RAM (all out)
04	FB	I/O out, CHAR ROM in
05	FA	BASIC & I/O out
06	F9	I/O & KERNAL out
07	F8	64K RAM (all out)

REVIEW

This first cartridge chapter covered the fundamental operation of the 64 with external ROMs. You have been introduced to some of the common methods of cartridge protection including KERNAL routines and BASIC switches. The following outline lists the areas that were covered and the steps necessary to copy a 16K cartridge.

1. Use the 'M' command to switch out BASIC.
2. Use the 'H' command to search for KERNAL calls and BASIC switch-ins.
3. NOP KERNALs and modify BASIC switch-ins.
4. Save altered code to disk.
5. Type G FCE2 from the ML monitor to run the program.
6. If the cartridge crashes, use the 'C' command to compare the two versions.
7. If the cartridge 'writes' to itself, see the next chapter.
8. If there are no 'writes' see the section on CIA TIMERS in the next chapter.

ADVANCED CARTRIDGE PROTECTION

The previous chapter outlined the fundamentals of cartridge operation, copying cartridges to disk, and simple protection removal. This chapter will cover advanced cartridge protection, CIA TIMERS, MAX cartridges, and auto-start boots.

The last chapter dealt with the removal of protection schemes that did not alter or ''write'' to the cartridge area ($8000-$C000). This chapter concentrates on protection schemes which attempt to alter themselves, resulting in a program ''crash'' if the program is run in RAM.

Cartridge-based programs can be considered a permanent form of computer memory. The program is ''burned'' into 1-4 ROMs and cannot be altered by the user (see the advanced section on EPROMs). When a program attempts to write values to the cartridge area ($8000-$C000), it is checking to see if the program is still in ROM and hasn't been copied to RAM. Since data in RAM can be modified, a program which writes to itself can cause a system ''crash''. For this reason, a program which attempts to alter itself must be changed in order to operate properly when copied from ROM to RAM.

Remember, it is assumed that all cartridges are 16K. If you have an 8K cartridge, you must change the ending address used with the monitor from $C000 to $A000. It is also assumed you have copied and run the cartridge, reset the computer, and are looking at a computer screen which reads 30719 BYTES FREE. In another words, it is assumed that you have read and understood the previous chapter!

First, SYS 49152 to enter the ML monitor and use the 'M' command to change address $0001 from $37 to $36 (flip-out BASIC). The crashed version must be transferred from $8000 to $2000:

```
T 8000 C000 2000
```

Next, load the original version from disk:

```
L''program name'',08
```

After loading the original version, use the 'C' command to compare the two versions:

```
C 2000 5FFF 8000
```

This time let's assume some addresses were reported as changed in the ''crashed'' version. This means the program wrote to itself, and it probably also contains some KERNAL and BASIC switches. Write down the addresses reported and use the 'H' command to search the code from $8000-C000 for the code which altered the reported addresses. FIGURE 8 is an example of a simple store to a series of addresses. This type of addressing is called ABSOLUTE addressing.

FIGURE 8

```
8E69  A2 00     LDX #$00
8E6B  8E 00 82  STX $8200
8E6E  8E 57 86  STX $8657
8E71  8E 00 90  STX $9000
```

In this example, the compare would have returned addresses $8200, $8657 and $9000 as being different from the original program. All of these addresses would have been different because address $8E69 loads the X-register with a $00 which is then stored at $8200, $8657, and $9000. If the program had been in ROM, no changes would have occurred at these addresses (or rather, the RAM under the ROM would be altered). Since the program was in RAM, the addresses were changed, and the program ''crashed''. In this example, you should use the 'H' command to locate the area of the program which altered the three reported addresses. Since this type of addressing can use the Accumulator, X-register, or Y-register, you must search the code for all three types. FIGURE 9 shows how you would search for the code that altered address $8200.

FIGURE 9

```
H 8000 C000 8D 00 82 for STA $8200

H 8000 C000 8E 00 82 for STX $8200

H 8000 C000 8C 00 82 for STY $8200
```

The use of these 'H' commands on the code in FIGURE 8 would have returned address $8E6B. Since the example used the X-register, only the search which specified the X-register would have returned the address $8E6B. The other two searches wouldn't have reported anything. Using the same technique, the search for addresses $8657 and $9000 would have returned the addresses $8E6E and $8E71 respectively.

After locating the addresses, use the 'M' command to replace the code with NOP's ($EA). FIGURE 10 shows how FIGURE 8 should be altered so the program can run in RAM.

FIGURE 10

```
8E69  A2 00     LDX #$00
8E6A
-8E73 EA        NOP
```

Remember, always work on the version of the program which resides at $8000. The version at $2000 has a $55 at the first byte(due to the RESET), as well as altered code caused by the writes. Also, if you repair and save the version at $2000, the program will not run because it will load back in at $2000, instead of $8000 where the cartridge normally operates.

After making the changes, be sure to save the code from $8000-$C000. Try running the cartridge by typing G FCE2 from the ML monitor (same as SYS 64738 from BASIC). If you have properly repaired the addresses and removed any KERNAL calls and BASIC switch-ins the cartridge should run. If not, there is one type of protection which is extremely effective, yet doesn't always modify program code. This type of protection checks CIA TIMER A (more on this later).

FIGURE 8 altered the program by directly changing the values in memory. FIGURE 11 uses a technique to store values in memory with an addressing technique called indexed addressing. This type of addressing uses the X- or Y-register to hold a INDEX (OFFSET) value which is added to a BASE address to arrive at the actual address to be altered. FIGURE 11 shows two examples of how indexed addressing can be used.

FIGURE 11

(A)

```
9000 A9 00     LDA #$00
9002 A0 05     LDY #$05
9004 99 00 81  STA $8100,Y
```

(B)

```
9000 A9 00     LDA #$00
9002 A0 05     LDY #$00
9004 99 00 81  STA $8100,Y
9007 C8        INY
9008 D0 FA     BNE $9004
```

This type of addressing can be used in two ways. In FIGURE 11a a single address ($8100+$05=$8105), is altered when line $9004 is executed. This is similar to FIGURE 8 because only a single address is altered by the statement.

FIGURE 11b produces a very different result when it is run. It uses two extra instructions to erase an entire section of memory - BNE and INY. Line $9004 stores a $00 byte to an address calculated by adding Y to $8100. By increasing Y step-by-step with the INY (INcrement Y) instruction and executing line $9004 repeatedly, we can write to many addresses with one piece of code. When the Y-register returns to $00 (remember, a single byte can only hold values from $00 to $FF; if you try to exceed $FF it 'wraps around' back to $00) the Branch if Not Equal to zero instruction fails and so the program continues past this point. This net result is to write zeroes to locations $8100-$81FF, erasing a whole section of the program. It is fairly obvious when this technique is used because a comparison will return a large consecutive series of altered addresses, using containing the same value.

Use the 'H' command as in our previous example to locate the address or addresses which alter the code, and then replace the stores with NOP's. In FIGURE 11a, you would replace addresses $9004-$9006 with $EA. In FIGURE 11b addresses $9004-$9009 should be replaced with $EA. In general, try to change as little as possible. Replacing self-modifying code with NOP's and removing any KERNAL calls or BASIC switch-ins should allow these examples to be run in RAM.

The next type of addressing is called indirect, indexed. This type of addressing uses a VECTOR (POINTER) to hold the actual address to be altered. The vector must be stored as two consecutive bytes in zero page, in low-byte, hi-byte format. FIGURE 12 shows an example using this type of addressing.

FIGURE 12

```
802E  A2 1F  LDX #$1F       Fill byte / page number
8030  A0 00  LDY #$00       Lo-byte of start address
8032  84 B8  STY $B8        Lo-byte of vector
8034  A9 80  LDA #$80       Hi-byte of start address
8036  85 B9  STA $B9        Hi-byte of vector
8038  8A     TXA           Use $1F as filler byte
8039  91 B8  STA ($B8),Y    Store filler to RAM
803B  B1 B8  LDA ($B8),Y    Read back from ROM/RAM
803D  91 B8  STA ($B8),Y    Store filler/code to RAM
803F  88     INY           Next byte
8040  D0 7F  BNE $8039      Continue if page not done
8042  E6 B9  INC $B9        Next page
8044  CA     DEX           Reduce page number
8045  10 F2  BPL $8039      Loop back if not done
8047  A5 01  LDA $01        Get current memory map
8049  29 FE  AND #$FE       Turn off BIT 0 (LORAM)
804B  85 01  STA $01        Replace map
```

FIGURE 12 is actually an example of two protection techniques. The first is the use of indirect, indexed addressing to disguise the address of the area of memory being changed. The second involves transferring the ROM to RAM and switching out the ROM in order to run the program from RAM. This routine has a couple twists, so take it slowly. The routine starts by setting X to $1F, which serves a dual role as a filler byte and page counterNext, Y is set to $00 for indexing purposes. Addresses $B8 and $B9 are set up to contain a pointer to the start of the area to be written to, in this case $8000. Then X ($1F) is transferred to A to be used as a filler value. This completes the set-up for the routine, which then enters its main loops starting at $8039. Leaving the code from $8039 to $803E aside for a moment, the code from $803F to $8046 controls the looping. We step through each byte using Y, as in the previous example. In this case we are going to write multiple pages, so at the end of a page we increase the high-byte of the pointer ($B9), decrease the page number (X) and check to see if we're done yet. If so we reconfigure memory as discussed later.

Now let's look at the tricky part: the LDA and STA's. We have to bear in mind that there are two possible cases, namely that the program is running in RAM or in ROM. This routine either erases the RAM copy and crashes or downloads the ROM copy to RAM and starts it. Let's take the RAM case first. Initially, the first STA will place a $1F at $8000. The $8000 is determined by taking the CONTENTS of $B8-B9 (zero page vector = $8000) and adding the current value of Y ($00). Next, A will be loaded back from $8000, getting the $1F back. Then this value will be stored out to $8000 again by the second STA. Since A still has $1F in it, when we loop back to $8039 after incrementing Y, we'll be storing $1F in $8001. This process continues for a while, but notice that the routine itself is in the first page after $8000. Soon Y will reach $2E and it will begin writing over itself at $802E. This will crash the RAM copy.

Now let's take the ROM case. First of all, remember that all stores will still take place in RAM but now the load will come from ROM. The first time it reaches $8039 the $1F will go into RAM $8000, but the LDA at $803B will pick up the value from ROM $8000, say $09. Since it then stores A back to RAM $8000, our first byte of ROM program will be transferred to RAM! Incrementing Y and looping back, we will then store the $09 in RAM $8001, but then we pick up the correct ROM $8001 byte and store it to RAM. Eventually, we will transfer all of the ROM copy to RAM ($00 to $1F = 32 pages = 16K). Since no problems are encountered, the loops will eventually finish and the code at $8047 will be executed.

This code will turn off BIT 0 of address $0001 (LORAM line). If both the EXROM and GAME lines are grounded, as they will be in a 16K cartridge, turning off LORAM will flip out the cartridge area from $8000 to $9FFF, but will leave the cartridge ROM at $A000-$BFFF flipped-in (replacing the BASIC ROM). Thus, if you're working with a 16K cartridge, it's

possible that the lower half could be running in RAM and the upper half could be running in ROM! So how do we modify this routine so a copy completely in RAM will work? Simply NOP addresses $8039 and $803A to prevent overwriting (as well as changing any BASIC switch-ins and KERNAL calls).

The final type of addressing is called indexed, indirect. This type of addressing uses zero page vectors like indirect,indexed. However, the contents of the X-register is used for indexing instead of the Y-register. Even more important, in this type of addressing the indexing takes place BEFORE the indirect access. The index might be used to select from a table of vectors, and then the selected vector is used as a pointer for indirect access. FIGURE 13 is an example of indexed, indirect addressing.

FIGURE 13

```
8040  A9 00  LDA #$00      Lo-byte
8042  85 82  STA $82
8044  A9 80  LDA #$80      Hi-byte
8046  85 83  STA $83
8048  A2 03  LDX #$03      Select vector no. 3
804A  A9 00  LDA #$00
808C  81 FB  STA (82,X)    Store indirect
```

In FIGURE 13, the Accumulator is stored to a location specified by the contents of addresses $85-$86, since $82+$03=$85. Thus the actual vector to be used indirectly is stored at addresses $85 and $86 in zero page, not $82 and $83. This type of addressing can be used to erase a single byte or entire section of memory. It is an extremely rare type of addressing which has yet to found in any cartridges, but it is useful for certain types of control processing in which you need to be able to select a particular vector from a list of vectors.

Some cartridges use the 64's CIA TIMERS to determine if a program is running in RAM or ROM. Each of the two 6526 CIA's is an I/O chip that contains 16 I/O lines, 2 linkable timers (A and B), a 24-hour clock with a programmable alarm, and an 8 bit shift register for serial I/O. Our example will use TIMER A of CIA #1. It is located at $DC04-$DC05 but is controlled by location $DC0E.

When the 64 is turned on, TIMER A of CIA #1 will be reset, that is, it will be set to zero and stopped. During normal initialization, the timer is set running again. When a cartridge is present on power-up, however, normal initialization is bypassed and the cartridge takes over. If desired, the cartridge can purposely NOT start the timer running. Later on, it can check the timer to see if it has been running (by whether it is nonzero) and crash the program if it is. When you go through normal initialization and load your copy from disk, the program will not work, since the

timer has been running. This is a extremely effective way of determining whether the program was started from cartridge. To defeat this type of protection you must use one of the auto-start boots at the end of this chapter. The boot programs store the value $00 at address $DC0E which resets the timers, making the program think the computer was just turned-on. In many cartridges that use this, it is the only type of protection. If you find a cartridge which doesn't alter any code but still won't run, try initializing the CIA TIMERS. DON'T INITIALIZE THE CIA TIMERS FOR EVERY CARTRIDGE. It actually prevents some cartridges from running. Only use it as a last resort.

The next type of cartridge to examine is the ULTIMAX cartridge (MAX for short). This cartridge was originally manufactured for the Commodore ULTIMAX computer, which was never distributed in the U.S. This cartridge reconfigures the 64's memory as shown in FIGURE 1d in the last chapter. This cartridge grounds the GAME line, which flips out the KERNAL and the cartridge area from $8000-A000. Most MAX cartridges are 8K and operate in the KERNAL area ($E000-$FFFF).

It takes a special technique to copy a MAX cartridge. Insert the cartridge into the expander board and turn ON the switches in the following order: POWER (sw4 on CB board); EXROM (sw2 on CB); and then GAME (sw1 on CB). If the switches are operated in any other order the computer will lock up and you'll have to start over. Once the switches are activated, use the "M" command to flip-out the BASIC ROM. The MAX cartridge will appear in memory from $A000-$BFFF. Keep in mind that the cartridge will not run from this area, it only resides here when the computer is powered-up. In order for a MAX cartridge to properly relocate and run when loaded, a special routine must be added to the end of each MAX cartridge. A routine that will relocate and run a MAX cartridge is shown in FIGURE 14.

FIGURE 14

```
C000  A9 36    LDA #$36
C002  85 01    STA $01        Switch out BASIC ROM
C004  A0 00    LDY #$00       Initialize lo-bytes:
C006  84 FA    STY $FA        Copy from address
C008  84 FC    STY $FC        Copy to address
C00A  84 FE    STY $FE         ''  ''    ''
C00C  A9 A0    LDA #$A0
C00E  85 FB    STA $FB        From $A000
C010  A9 20    LDA #$20
C012  85 FD    STA $FD        To $2000
C014  A9 E0    LDA #$E0
C016  85 FF    STA $FF        And $E000
C018  B1 FA    LDA ($FA),Y
C01A  91 FC    STA ($FC),Y    Copy a byte
C01C  91 FE    STA ($FE),Y     ''     ''
C01E  C8       INY            Next byte
C01F  D0 F7    BNE $C018      Done with page?
```

```
C021  E6 FB     INC $FB         Next page
C023  E6 FD     INC $FD
C025  E6 FF     INC $FF
C027  D0 EF     BNE $C018       Done with copying?
C029  78        SEI             Disable IRQ interrupt
C02A  A9 35     LDA #$35
C02C  85 01     STA $01         Turn off ROMs
C02E  6C FC FF  JMP ($FFFC)     Activate MAX reset
```

The code in FIGURE 14 must be added to the end of every MAX cartridge. The routine relocates the cartridge to $E000 and $2000. MAX cartridges require an image of the program to appear at $2000 or they will not run. To add the relocate routine to the end of a MAX cartridge, operate the switches as outlined above and use the monitor to assemble the code shown in FIGURE 14. The relocate routine should be assembled starting at $C000. Once the routine is entered into memory after the MAX cartridge, save the memory from $A000-$C031. This will save the cartridge and the relocate routine. You can save the code from $C000-$C031 as a separate file to use later if you need to copy another MAX cartridge. Once the cartridge and relocate routine are saved, it can be activated with a SYS 49152 or JMP $C000.

There are a few MAX cartridges which are protected with a KERNAL switch-in. Once the cartridge is relocated the program attempts to switch in the KERNAL by turning on bit 1 of address $0001 (HIRAM). To check for this type of protection in a MAX cartridge, use the 'H' command to search the code from $A000-$C000 for a store to address $0001.

```
H A000 C000 85 01
```

This command hunts for STA $01; you may have to look for STX and STY too. If you find an instruction where the value stored has bit 1 on (e.g. $E7) you must replace it with a value that has bit 1 off (e.g. $E5). This is the only type of protection that has ever been found in a MAX cartridge to date.

This concludes the discussion of cartridge protection and how to remove it. The next sections will cover auto-boots, and special types of cartridges.

We will cover two types of auto-boots: one which loads a single program and one which loads up to four separate sections of code. Either of these boots can be built using the program called SUPERBOOTER on the accompanying disk. FIGURE 16 is a disassembly of a typical single program boot built by SUPERBOOTER.

FIGURE 15

```
02A7  20 44 E5 JSR $E544   :CLEAR SCREEN
02AA  A9 01    LDA #$01
02AC  8D 20 D0 STA $D020   :BORDER COLOR
02AF  8D 21 D0 STA $D021   :SCREEN COLOR
02B2  A2 80    LDA #$80
02B4  8E 84 02 STX $0284
02B7  86 38    STX $38
02B9  20 53 E4 JSR $E453   :RESTORE BASIC VECTORS
E2BC  A9 01    LDA #$01
02BE  A6 BA    LDX $BA     :CURRENT DISK NO.
02C0  A8       TAY
02C1  20 BA FF JSR $FFBA
02C4  A9 06    LDA #$06    :6 LETTER NAME
02C6  A2 FA    LDX #$FA    :AT $02FA
02C8  A0 02    LDY #$02
02CA  20 BD FF JSR $FFBD
02CD  A9 00    LDA #$00
02CF  20 D5 FF JSR $FFD5   :LOAD
02D2  20 E7 FF JSR $FFE7   :CLOSE ALL CHANNELS
02D5  86 A2    STX $A2
02D7  A5 A2    LDA $A2
02D9  D0 FC    BNE $02D7
02DB  8D 0E DC STA $DC0E   :INITIALIZE CIA TIMERS

02DE  EA       NOP
-02F2

02F3  A9 36    LDA #$36    :SWITCH-OUT BASIC ROM
02F5  85 01    STA $01
02F7  6C FC FF JMP ($FFFC) :JUMP TO RESET VECTOR
02FA  58                   :X
02FB  58                   :X
02FC  2E                   :.
02FD  4F                   :O
02FE  42                   :B
02FF  4A                   :J

0300-0301  8B E3           :NORMAL BASIC ERROR VECTOR
0302-0302  A7 02           :SET BASIC WARM-START
                           :VECTOR TO JUMP TO $02A7
```

As you can see, the boot in FIGURE 15 initializes the CIA TIMERS (STA $DC0E). This must be added with a ML monitor after the boot is built. SUPERBOOTER will build a boot for a cartridge or a SYS address. This boot will auto-start because addresses $0302 and $0303 (BASIC warm-start vector) are modified to jump to the beginning of the autoboot at $02A7. Normally, the vector is $A483, but it can be modified to jump to a ML program. In order to auto-start, the boot must be loaded with LOAD''boot name'',8,1. After the boot is activated it restores the warm-start vector by running a subroutine at $E453.

In order for this boot to work properly, the main program must be saved in the XX.OBJ name format. The boot program should be saved with the full program name. For example, if you had a cartridge called''Cartridge Backer'', the boot would be called ''Cartridge Backer'' and the main program would be called ''CB.OBJ''.

SUPERBOOTER will ask you for the name of the program, the two letters in the XX.OBJ file name, the number of boots, whether you want BASIC switched out, and whether the boot is for a cartridge or a SYS start. If you wish to SYS a particular address it must be entered in decimal. Also, if you wish to initialize the CIA TIMERS, you must load the boot with a ML monitor and add the store to address $DC0E as shown in FIGURE 15. Once you have answered SUPERBOOTER's questions, it will build a boot and save it to a disk. It is assumed that you are saving the boot to the disk that contains the main program. If you are using SUPERBOOTER to build a boot to load multiple files, the sections of the program must be saved as XX.OBJA-XX.OBJD.

The final area which will be covered is cartridges which are larger than 16k and special ROM types. Cartridges which are larger than 16K use a technique known as bank-switching or bank-selection. The cartridge uses a chip called 74LS74 to ''flip'' ROMs in and out as needed. The usual addresses for the selection register is $DC00 or $DFFF. The bank-selection technique allows a cartridge to be used in conjunction with the 64's ROMs.

A few MAX cartridges contain special ROMs which cannot be read properly unless the data is read several times with a monitor. You can identify this type of ROM by reading the memory at the beginning of the MAX cartridge several times. If the data that is read is different each time, the cartridge contains these ROMs. If you have this type of cartridge, you must read the data until it doesn't change (usually 10-15 times). If you don't read the data several times, when you save the cartridge you will get all $FF's.

This concludes the basic techniques for copying and removing the protection from cartridges. It can take from a few minutes to several hours to find and remove the protection from a particular cartridge. The CARTRIDGE BACKER

system available from CSM SOFTWARE INC. will automatically remove the protection from 99% of all cartridges, build an autoboot, and save everything to disk in less than 90 seconds! The Cartridge Backer system will save the experienced programmer time by locating KERNAL calls, BASIC switch-ins, writes to RAM, indirect addresses, etc. The system includes the software on disk, an expander board, and a user's manual. The software package is entirely menu driven, making it very easy to use. If you have access to a number of cartridges, the CARTRIDGE BACKER system is well worth the investment of $54.95

Here's an experience most of us can relate to: You've got a new program that you would like to investigate. You load it for the first time and find that it is an "auto-load" program. You let it go through its protection scheme, you listen and you watch carefully. The program runs. Now you are ready to 'dig in'. Since the program autoboots, you use your machine language monitor to load the program into memory (modifying the autoboot if necessary - see the chapter on autoboots). You put your ML monitor into Interrogate mode (or Memory mode) and expect perhaps to see some code. Hmmm... nothing looks familiar. Must be in machine language. You now switch to D(isassemble) in order to check the machine code and find to your surprise that 99% of the program looks like garbage!! The chances are that you are looking at a program which has been encrypted or one that uses undocumented opcodes.

Why would someone want to "code" their code? The answer is obvious - they don't want you to analyze it and figure out how to disable the protection scheme. The purpose of this chapter is to review some methods of program encryption and how to get around them. For our purposes in this chapter, whenever we refer to coded, encrypted or undocumented opcodes you may consider them all to mean the same technique.

One thing is certain; there must be some valid machine language somewhere. The C-64 doesn't know how to decode someone's program and therefore there must be a routine which performs the decoding somewhere in memory. This is the obvious place to begin your quest.

You may wish, at this point, to try a different approach. Why not simply reset the computer after the program has self-decoded (or gone past the undocumented opcodes) and then begin your examination? This may work in some cases, but most of the time a number of things have occurred during the decoding process which you are unaware of. A clever programmer will have wiped out all traces of the decoding before you have even gotten this far. For example there may be values stored on zero page or under the Kernal which are used as the basis for an indirect jump. True, if you are young you may have several years to spend carefully tracing the program, instruction by instruction, looking for these values and finding where the 'extra stuff' is hidden. Anything can be broken given enough time and energy but isn't there an easier way? Many times the answer is yes.

First of all, many program authors feel that since the program is encrypted or uses undocumented opcodes it will be impossible for anyone to figure out what is happening. They often leave a trail which is very easy to follow. Here is a method for dealing with coded programs which has worked often on programs.

1). Use your monitor to load the autoboot program.

2). Once you have determined that it is an encrypted or coded program start over and let it load normally through its autoboot routine (from BASIC).

3). Once the program is up and running, we assume it is past the encoded portion. Use your RESET button to get control of the computer and "fire up" your ML monitor. (Note: HESMON is nice here since it is a cartridge monitor. You still need a cartridge switch installed or an expander board. See Vol. 1 of the PROGRAM PROTECTION MANUAL from CSM)

4). Begin by looking for calls to Kernal routines. We are assuming here that you know what to look for. Again, Vol. 1 of the PROGRAM PROTECTION MANUAL has thoroughly explained this.

5). Locate the protection scheme in the decoded program. List out the code around this point on your printer if you have one.

6). Figure out how to beat the protection scheme. Often this will amount to changing a CMP #$32 to CMP #$30 or some similar change. If more elaborate changes are necessary you have to work harder.

The trick now is to figure out where in the original (coded) program the changes must be made. We must make some assumptions here. Most coded programs have a 1-to-1 relationship with the decoded program. In other words, the coding scheme changes each byte according to some pattern (discussed later), but doesn't change the byte's position in the program. If this assumption is true, you can count down the coded program and locate the exact byte or bytes you need to change. Here is a more specific example:

Suppose that the load address of the main program (encrypted) is $1000. You have located the protection scheme and you decide that you need to change the byte at address $2356 from $32 to $30 (as in CMP #$32). You need to figure out which byte in the coded program corresponds to byte $2356 in the decoded program. Since there is $1356 bytes difference ($2356-$1000=$1356), you simply go $1356 bytes into the disk based version of the program to locate the corresponding position in your coded program. Thus the byte at $1356 from the beginning of the program needs to be modified. We now have a new problem. Since we are looking at a coded byte, how do we know what to replace it with so that it will decode into a $30 and make the program bypass its protection scheme?? If you have no idea of how the encryption works you will have to use a bit of trial and error. Let me share a personal experience that may give you some insights.

I reached the exact point described above once while investigating an encrypted program. At the time I didn't know anything about coding and decoding. Some thoughts on this will be discussed a bit later. I did know that at worst I had a 1 in 255 chance of guessing the value that would decode into what I needed ($30). I was very sure that all that needed to happen was for a CMP #$32 to become a CMP #$30. I had invested hours on the program and decided to "go for it". I was willing to try all 255 possibilities. Using a track and sector editor I counted down the bytes on the disk itself until I found the one to change. What happened next was pure luck. I accidentally changed the byte on the disk which contained the machine code for the CMP instruction itself instead of its operand, the $32. I crossed my fingers as the autoboot began. The program booted successfully with no banging - no bad blocks. It worked! I had gotten lucky. I could have spent the rest of the day trying 255 possible values, hoping one would work.

In the above example the critical part was locating the area on the disk that must be modified. When the code was located on the disk the 'lucky' change that was made actually turned out to be the proper place to change the code. By changing the CMP instruction we have many more possibilities of finding a instruction that will work "by guess and by gosh". In most instances it is more desirable to change either the CMP or the BNE (BEQ) instruction than the value itself (#$32).

I am sure we have all had similar experiences where through some lucky happening we have succeeded in our quest. We can't rely on luck however and so an understanding of how a program can be coded will allow you to analyze the decoding routine and figure out how to make changes which will work.

EXCLUSIVE-OR

There is a 6502/6510 machine language instruction known as EOR or exclusive-OR. It is generally used in calculating checksums (as well as other purposes). In the context of coding schemes the EOR instruction can be of great value. The next few paragraphs are a review of the EOR and its use in program protection.

Remember, whatever system we use to code the information must be reversible. We have to be able to get back to our original program from the coded version. The EOR instruction is useful here because it performs a reversible change on a given byte. Let's look at a couple of examples.

You are probably already aware of the definitions of OR and AND as 'logic operators'. Below are 'truth tables' for OR, AND and EOR:

OR Truth Table	AND Truth Table	EOR Truth Table
0 OR 0 = 0	0 AND 0 = 0	0 EOR 0 = 0
0 OR 1 = 1	0 AND 1 = 0	0 EOR 1 = 1
1 OR 0 = 1	1 AND 0 = 0	1 EOR 0 = 1
1 OR 1 = 1	1 AND 1 = 1	1 EOR 1 = 0

As you can see, OR and EOR are similar. They differ only in the 1-1 value. Now let's try using the EOR operation between two hex numbers expressed in binary. As an example we will perform $A7 EOR $6C:

```
$A3 = 10100111
$6C = 01101100
--------------
$CB = 11001011
```

Check the EOR truth table in order to see how this result comes about.

Now let's see what happens when we EOR the result $CB (11001011) with $6C again:

```
$CB = 11001011
$6C = 01101100
--------------
$A3 = 10100111
```

But this answer is exactly what we started with - $A3!! In fact this "reversibility" always happens. Thus EOR is an excellent way of coding bytes in a way which is reversible. We would pick a value to use as our "coding value". In this case we have used $6C. We would then perform an EOR operation between $6C and each byte of the program. This would make the program look like complete garbage. The beauty of the EOR operation is its reversibility.

In order to get our program back again (decode it), we must EOR each byte with $6C again. This will bring back the original program, byte for byte.

This coding scheme is easy to understand and also easy to beat. There are only 255 different values which could be used for the "coding value" ($00 wouldn't change anything). This means that a simple trial and error program should be able to find the proper "coding value" and reconvert the program into good code.

Software protectors may use a more complex coding scheme than this. The saving grace here is that the decoding routine itself must be uncoded. There must be valid code in the program so that it can begin executing. When you have found that a program is coded you should be able to find good code right at the beginning. Very likely this part of the program is the decoding routine. These routines are usually very short and generally relocatable. It is often possible to relocate the decoding routine and execute it, causing it to decode the main program without autobooting.

You would load the autoboot program into the computer and then disassemble it. You may find the decoding routine. Analyze this short routine and modify it so that it decodes but doesn't execute the main program (i.e. change the JMP instruction at the end of it). Now you can execute the decoding program, causing the main program to decode but not to execute. You should now be able to locate the protection scheme in the main program and disable it. You can relocate the decoded program in its proper place in memory and then begin execution at the point specified at the end of the decoding routine. If you can do it this way, you got off easy. Some clever protectors have gone beyond this...

Several programs (SUPERBASE, for one) have elaborate coding and decoding systems built into them. They are designed to drive you nuts if you try to analyze them. Here's what you are up against with these 'super coded' programs:

Naturally there will be an autoboot routine. These are easy to analyze. In the case of coded programs, the decoding routine is executed immediately after the main program loads. Unfortunately, we now discover that many parts of the decoded program have been modified or totally obliterated during the decoding process. A careful examination of the autoboot routine reveals that it only decodes part of the main program and then transfers control to this newly decoded routine. Our next step is to disable the transfer of control so that we can get a look at the decoded routine. The obvious method would be to place a BRK instruction into the autoboot routine at the point that it was going to JMP to the main program. What we find is that 100 or so bytes of the main program have been decoded. Naturally we are curious about this routine and begin a careful analysis of the machine code. It is 'good code' in the sense that there are no ???'s populating it. It is truly an executable machine language routine. However, as we begin our analysis we are getting the feeling that this routine is not an essential part of the main program. The routine is running us through every possible weird operation imaginable. We are ANDing, ORing, SBCing, ADCing, CMPing, branching here and there based on outcomes which are very easy for us to miscalculate. In other words we are being asked to perform all sorts of wild calculations and decisions perfectly. One mistake in one bit is enough to send us off into oblivion.

It is painfully obvious that this routine is for us. Its purpose is to drive us crazy. We find out after half an hour of careful analysis that this routine decodes the next 100 or so bytes of the program and then self-destructs. Oh no... here we go again! More crazy calculations designed to drive us nuts. The day is yet young - let's go for it... After another hour of careful calculating we find that once again the purpose of this second part is to decode the next section of code and then self-destruct. And so on and so on.... Hey, why not use the machine language monitor to single-step through the routine carefully? This is a good idea but not as simple as it sounds. It is pretty easy to hang up the monitor when it is single-stepping through the program and these little routines are designed to do just that.

The moral of this little story is that a protection scheme can be very well hidden in a program. Clever programmers are making it much more difficult to find those KERNAL calls which check the disk protection. The "end of the rainbow" is very far away. Undoubtedly you will find that if you stop the program after it has passed the protection scheme, the protection scheme has self-destructed. You won't find a trace of it left in the program.

Is this the end of the line? The answer is no. You have two alternatives: 1) You can batten down the hatches and get ready for a long haul. Here you plan to carefully analyze the program, find the protection scheme and disable it. This could be a big project. OR 2) Elsewhere in this book several programmers discuss alternate methods - methods which may allow you to "lift" a working copy of the program from memory. If you can do this, you will avoid the problem of coded programs entirely.

It is hoped that this discussion of encrypted programs has given you some ideas as to what is possible in the area of program encryption and what you might be up against if you try to locate the protection scheme and disable it. Sometimes coded programs are not difficult to beat. However, programmers often have spent many hours creating routines for you to analyze which are designed to give you nightmares. In any event, it is worthwhile to spend some time analyzing coded programs. You may find that you CAN 'break' some of them without losing your sanity.

For all practical purposes it is easier to let the program load into memory, decode itself and execute than it is to try to decipher the encryption scheme. Lifting a program out of memory will be covered in the next few chapters. After all, it is not important how the working program got into memory. What is important is that a working version of the program may usually be "lifted" out of memory after it has passed all its "checks".

THE BACKDOOR APPROACH

Program protection has come a long way in the last year. We have seen the introduction of non-standard sectors, extra track and sectors, nibble counting, altered density bits, and much more. Each new scheme results in the creation of a "99% EFFECTIVE COPY PROGRAM" designed to defeat that scheme. We later find that this copy program is only good until a new protection scheme is developed. What all of this means is that you are continually frustrated in your attempts to exercise your lawful right to create an archival copy of your valued software. It is our opinion that there is no 'perfect' scheme. Every program may be "unprotected". The newer schemes may require a little more investigation than others, but the result is the same.

In this chapter, we will explore the "BACK-DOOR" approach to "unprotecting" a program. We will analyze some of the more recent schemes, and the ways to defeat them. The code we will analyze is typical of that being used today. Often you will find that the techniques explored in the PROGRAM PROTECTION MANUAL VOLUME I are still being used, but the disguise is better.

The theory behind the "back-door" approach is to capture the code once it is in memory. We allow the program to run normally and then reset the computer, with the thought that the error checking will no longer be utilized once the program is in memory. This saves us the trouble of tracing through some of the more sophisticated protection schemes. Once the program is in the computer's memory, the only thing we have to do is find where the code is stored in memory and the proper entry point to start the program. Up to now, this approach has worked extremely well and most programs are still being captured in this manner. Recently, we have noticed a few "curves" in the back-door method. Programmers are now utilizing areas of memory that up to now have been avoided, error checking is done once the program is in memory, and programs are encrypted. In these programs, we find that a combination of the "front and back-door" is required to capture a program. We will illustrate this process later in this chapter.

GETTING STARTED:

As stated earlier, the "back-door" is meant to be a time-saving technique. What we attempt to do is allow the program to load in the normal fashion. Once the program is in memory and running properly, we assume it has passed the protection scheme. We now have the program code without protection. We no longer have to be concerned about tracing down and defeating the protection scheme. All that's left is to find a proper entry point for the program. Before you attempt this procedure, we recommend a bit of preparation and investigation. You may

find that tracing a one block boot program will reveal a protection scheme. If this is all that is needed to 'unprotect' a program, then there is no need to use the "back-door" method. Don't be in a hurry when you begin your study of a program, because you may miss something simple. We recommend that you follow the steps below before you proceed to utilize the "back-door" approach.

1). Record the starting addresses and program lengths for each program on the disk you are examining. This will give you an idea of what area of memory the program will occupy. This will also help you determine which machine language monitor should be used. When attempting to locate the ending address, don't forget about the END OF PROGRAM LOAD VECTOR at $00AE. When you load a program into memory through a machine language monitor or from BASIC, using M 00AE will reveal the ending address (plus 1) of the program that was loaded. Keep in mind that a RESET will erase this address. Many disks directories have been altered to prevent listing. Make a BACK-UP disk and alter the DIRECTORY, so that you may get a proper listing. The P.P.M. VOL I describes this procedure in depth.

2). Make a note of the track and sector location for each file on the disk. If after examining a program, you find that the necessary alterations could be made quickly on the disk, this record could be a handy reference.

3). Examine the "BOOT" or LOADER program first. Check your directory notes to determine where the boot resides in memory. Examine the program through a machine language monitor in INTERPRET MODE (I). This should reveal the name of the next program to be loaded. Make a note of the order in which the programs are to be loaded.

4). Once you have determined the order of the program loads, examine the "boot" program in DISASSEMBLY MODE with you machine language monitor. Use the HUNT feature of your monitor to search for KERNAL CALLS (FF..), CMP#$30, and CMP#$32. Trace the entire "boot" program through DISASSEMBLY MODE. If the protection does not appear to be stored in the boot program, check the program that is to be loaded next. Follow this same procedure until all files have been examined.

5). If the program you are examining is stored in USER FILES, you may find it easier to make the correction directly on the disk. Begin your investigation on the disk with the "boot" program. You may find that the protection scheme is stored in the first or second block of the boot. Use a TRACK AND SECTOR EDITOR such as Di-Sector or Peek A Byte that includes a DISASSEMBLY feature. Make a record of your changes so that you may restore the disk to its original condition, if the correction does not work.

6). When you examine a file through the DISASSEMBLY mode, attempt to determine if the code is "ENCRYPTED". If you see a lot of ???'s in the DISASSEMBLY, the code is probably designed to be modified when executed, or else contains undocumented opcodes.

What we are attempting to do here is to determine if the "fix" will be an easy one. If the procedure described above does not reveal the protection scheme, you are ready for the "back-door" approach.

TOOLS AND TECHNIQUES:

We still find that 99% of the programs we "unprotect" merely require the right tools and techniques. If after examining a program, we find that the error routine will require too many hours to trace, we attempt to get the program once it is in memory. In most cases, it becomes a matter of knowing when to stop the program and how. If all goes well, all that's left is to find a suitable entry point.

How and where you stop a program can be crucial to capturing the code intact. Many of the protection schemes will program the CBM80 COLD-START vectors to erase the code when a RESET occurs. Refer to the section on INTERRUPTS for a detailed explanation of the RESET process. Determining if the program you are working on utilizes this technique is rather easy to discover. You need only fill memory with 00's or (99's), run the program from the original disk and perform a RESET once it is in memory. If the screen fills with 'garbage' or locks up, the program has probably been sent to a self-destruct sequence. Other programs may not be so obvious. The program may appear to perform a normal RESET, returning you to the normal blue screen. Once the program has been loaded and your RESET has been performed, load in HIMON to examine the code from $0801 through $BFFF. If you find repeated patterns throughout the code, or a large section of BRK's (00), you can assume that the program was altered through the COLD-START vector.

If you have a cartridge power switch such as the one described in the PPM VOL I, or a cartridge board that allows you to control the EXROM line, you may capture a program of this type. Through the technique described below, we will attempt to regain control of the computer without losing any of the program code by a cold-start via the CBM80 and associated code.

If you are using the cartridge power switch, load the original program in the normal manner. Once the program is in memory, insert a cartridge-based machine language monitor, with your cartridge switch OFF, and RESET your computer. This is the same as grounding the EXROM line. Remove the cartridge monitor. At this point, you should load a high monitor such as HIMON to examine and save out the program code. Don't forget to flip out BASIC to determine if there is any code stored in the RAM beneath. Once you have saved out this section of code, we

recommend that you repeat the procedure so that you may determine if some of the program code has been stored from $C000 through $CFFF. Load a low monitor, such as LOMON, to examine this area of memory. Once you have saved out the code, you must now find an entry point for the program.

To capture a program using an expander board, load the program normally. Once the program is in memory, set your EXROM line to LOW and RESET your computer. You will be returned to the familiar blue screen. Set your EXROM line to high and examine the code. For a detailed description of the effects of the EXROM line, refer to the chapter on INTERRUPTS.

FINDING THE CODE:

When we choose the "back-door" for our approach, we are faced with the problem of determining where the code is stored. Before you load the program, you should clean-up as much memory as possible through the use of a machine language monitor. We recommend that you use a high monitor such as HIMON for this procedure. Since this monitor resides at $C000, you can clear-up the memory from $0800 through $BFFF utilizing the F command. Follow the procedure described below for this process:

1). First flip out the BASIC interpreter by storing a 36 at $0001. Using the M command at 0001, change the 37 to a 36. This allows us access to the RAM under BASIC.

2). Using the F command, fill the memory from $0800-$BFFF with $00's. F 0800 BFFF 00. This will fill the memory with BRK's ($00). Now when you load the program you are working on, you will be able to determine where the code begins and ends.

3). When the program loads, it will over-write the BRK's (00) that we inserted. The end of code may be determined from where the BRK's (00) begin again.

When you load the program and examine it through your high monitor, with the I command, you should be able to determine where the code is stored in memory. The program code will over-write the BRK's (00's) that we inserted. Now when we examine the code using the I command, we will find the end of the code by searching for BRK's (00) that we inserted. It is usually good practice to save out all of memory when examining a program through the back-door. It will take two attempts to do this. On our first attempt, we will load the program as usual, perform the RESET, and load in HIMON. Since HIMON locates at $C000, we will be able to save out the program code from $0801 through $BFFF. Don't forget to flip-out BASIC to save out the RAM area beneath. Starting over again, load the program in the normal manner, perform your RESET, and then load in LOMON. LOMON resides at $8000. Now you may save out the code from $C000 through $CFFF.

The first area of memory to be checked is $8000. Check this area using the I command. If you see a CBM80 stored in this area, you probably have a program that utilizes an AUTO-START feature. This type of program will emulate a CARTRIDGE start when the RESTORE key is pressed or a RESET is done. Use the M command to examine the memory at $8000. The first two bytes at $8000 contain the address of the cold start vector, stored in low byte/high byte order. The next two bytes contain the warm-start vector, which are also stored in low byte/high byte. Then comes the "CBM80" itself. Once the program is in memory, save yourself a lot of time by trying these vectors. Activate the program with a GO to the warm-start vectors (G XXXX). If all goes well, all you need do is find the beginning and ending address of the program. Save out the code and activate with the decimal equivalent for the WARM-START VECTORS (EX. If the WARM-START VECTORS were $8E72, the decimal equivalent would be SYS 36466). You may find that if the CBM80 warm-start vector is used, pressing the RESTORE key once the program is in memory will activate the program. Following is an example of what you may see at $8000 using the M feature of your machine language monitor:

```
.:8000 D9 8D 72 8E C3 C2 CD 38
.:8008 30 20 F7 82 B0 03 4C 30

D9 8D - COLD START VECTORS - G 8DD9 WILL CAUSE A COLD START  AT
                                $8DD9.
72 8E - WARM START VECTORS - G 8E72 WILL CAUSE A WARM START  AT
                                $8E72.
C3    - C
C2    - B
CD    - M
38    - 8
30    - 0
```

SYS 36466 (HEX 8E72) would execute this program after a load with ,8,1.

For a detailed explanation of the CBM80, refer to the chapter on INTERRUPTS in this manual and the PROGRAM PROTECTION MANUAL VOL. I.

KERNAL STORAGE:

Many of the latest programs are now storing code to the RAM from $E000 through $FFFF. If you take a look at your memory map, you will find that this is the KERNAL ROM area of the operating system. As with BASIC, though, there is alternate RAM underneath. A few more steps will be necessary to gain access to this code. Programs that utilize this area of memory will run normally when the proper entry point is found after RESET. All seems well until you save out the code from $0800-CFFF, power-down, reload, and start the program from the entry point that you used when the original program was in memory. If you

have experienced this problem, you may find that there is additional code from $E000-$FFFF. Since we have found so many programs lately using the RAM under the KERNAL ROM, we have made this a part of our normal routine. It may require a few extra steps now, but will save a great deal of time in the long run. In order to access the code in this area, we must utilize a program to transfer this code to another area of memory. The program called MOVE KERNAL on your PROGRAM DISK will transfer the code from $E000-$FFFF to $2000-$4000. To save out the code in this area, follow the procedure described below:

1). Load the original program from your disk and perform a RESET.

2). Load and execute LOMON. Clean-up the work space from $1000-$4000, with F 1000 4000 00.

3). Load "MOVE KERNAL" (L "MOVE KERNAL",08) from your PROGRAM DISK. This program will be located at $1000. This program will set the interrupt flag, which will prevent IRQ's from occurring, and transfer the code stored under the KERNAL to $2000 through $4000. Then it will clear the IRQ flag and BRK to the monitor. To execute the program you would type G 1000. Using the I command, scroll through the code at 2000. Save this code with S "KERNAL CODE",08,2000,4000.

4). We will encounter a problem when the "KERNAL CODE" program is loaded back into memory. When loaded, KERNAL CODE will locate at $2000-$4000, because we saved it out from that area of memory. The program we are working on would expect to find this code at $E000. We could write a loader program to relocate this code, or we could do it the easy way. With a track and sector editor, we can locate where the first block of the KERNAL CODE program is stored on the disk. Go to TRACK 18 and locate KERNAL CODE in the directory. Once you locate that file, go to that TRACK and SECTOR. Byte 3 of that block will contain a 20, which is the high byte of the load address. If we change this to E0, the program will locate at $E000 when it is loaded with ,8,1.

5). With the main section of code and the additional KERNAL CODE section loaded into memory, you should now try your entry point. If the program still does not function properly, there is probably still some other code that must be captured.

FLIPPING-OUT BASIC WITHIN YOUR PROGRAM:

Before we look into other areas which may contain program code, we should point out that some programs require that you flip-out BASIC before the program will run properly.

To determine if you have a problem of this kind, load the program code in and load a machine language monitor that will not over-write the program. Use a monitor that allows you to flip-out the BASIC interpreter (LOMON, LLMON, HIMON). Change the 37 at $0001 to a 36, and try your entry point. If the program runs properly, you will have to write a section of code that will flip-out BASIC for you and then JMP to your entry point. You may also use AUTOBOOT1 from your PROGRAM DISK. There is a section in this program that will allow you to store a 36 at $01. If you prefer to do this within your program, follow the procedure below:

1). Determine where there is space within your program code. Scroll through the code using the D command, and search for a small section of BRK's (00).

2). Once the area has been determined, input the following code using the M command:

```
0A00 A9 36 85 01 4C 00 80
```

In our example, we have inserted the code at $0A00. You would store this piece of code wherever you have room. Our JMP was to $8000. Your JMP would be to whatever entry point you are using. The disassembly is as follows:

```
0A00 A9 36    LDA #$36  - LOAD THE ACCUMULATOR WITH 36, WHICH
                          IS THE VALUE THAT MUST BE PLACED AT
                          $0001 TO FLIP-OUT THE BASIC
                          INTERPRETER
0A02 85 01 00 STA $01   - STORE THE 36 AT $01
0A05 4C 00 80 JMP $8000 -  JUMP TO THE PROGRAM ENTRY POINT
                          ($8000 IN THIS CASE)
```

To activate our example program, we would use SYS 2560. This is the DECIMAL equivalent for HEX $0A00, which is where we stored our program to flip-out BASIC.

OTHER AREAS TO STORE CODE:

If you refer to your memory map, you will find that there are many other areas of memory available for program storage. If you have saved out the code from $0800-$CFFF and have followed the procedure to access the code from $E000-$FFFF but still find that the program will not execute properly, you may have to experiment with saving out the code from these other available memory locations. We'll pass along a few hints for determining where the code may be stored.

UNDER I/O DEVICES ($D000-DFFF)

Beside the Kernal and Basic ROM's, another area of memory which has "hidden" RAM under it is $D000-DFFF. This area is usually occupied by the I/O devices (VIC, SID, CIA's) and color RAM. The character ROM also occupies this area when it is switched in. Beneath all of these, there is also 4K of free RAM, accessible only when a special memory configuration is chosen. See the chapter on THE 6510 AND THE PLA for the details on how this is done.

LOW MEMORY:

There are lots of little areas in low memory available for storing protection values, especially if the program isn't going to use RS-232, tape, etc. We recommend that you investigate the program for a new entry point first. If this fails, we suggest that you use an altered machine language cartridge monitor that will not RESET so much of low memory. We use a cartridge of this type here and have been able to capture code in the CASSETTE BUFFER, and other areas of memory that were RESET through the standard cartridge monitor. The procedure for altering a cartridge monitor is explained elsewhere in this manual.

WHEN TO STOP THE PROGRAM:

When to stop a program can be crucial in the "BACK-DOOR" approach. We have encountered several programs of this type. Experiment with your program. Try timing the program to determine how long it takes to execute. Try stopping the program just after it has loaded and done its error checking but before it executes. Save out the code and try various entry points. It may take several trys before you find the right spot to stop the program. This can be time consuming, but it's still better than spending all that money for a copy program that probably won't work on the next program you purchase.

ENTRY POINTS:

This is the most difficult concept to teach. We can and will give you suggestions on what to look for, but the key to success here is experimentation. Through practice you gain experience. This can be a time-consuming process. There are some things that you should get in the habit of checking, but often finding the proper entry point is a matter of trial and error. Following is the procedure we use to locate an entry point along with examples of what to look for:

SCREEN COLOR ENTRY POINT:

1). Load the original program and allow it to run normally. Take particular notice of screen color changes, menu or title screens. The first thing you should check for is a re-start feature through the RESTORE KEY. This may be determined by striking the RESTORE key after the program is in memory and running. If this is included in the program, check the code at $8000 for the CBM80.

2). Check for stores (STA) to border and background color with the HUNT feature of your machine language monitor. Try H 0800 9FFF 8D 20 D0 and H 0800 9FFF 8D 21 D0 (hunt for STA $D020 and STA $D021). If these commands are found, investigate the areas with your D command. You may only have to go back to the LDA instruction that precedes the STA to try your entry point. Remember, before we can store the accumulator (STA), we have to load something into it. This is where the LDA comes in. This should occur just prior to the STA. The example below should give you an idea of what to look for.

```
., 0A00  A9 01    LDA #$01  - CODE FOR THE COLOR WHITE
., 0A02  8D 20 D0 STA $D020 - STORE WHITE TO BORDER COLOR
```

After the H 0800 9FFF 8D 20 D0, the memory address returned would have been $0A02. When we disassemble the code in that area we find the LDA at $0A00. In this example we would try a G 0A00.

3). Attempting a G immediately after an RTS (RETURN SUBROUTINE), will probably not be successful, because the second routine is probably part of a subroutine. Every subroutine ends with an RTS. When the program encounters this instruction, it will NOT know where to return to and will probably "crash". When you find a section that looks promising, scroll through the code with your D command to determine if this section of code ends with an RTS. If it does scroll up through the code until you find another RTS. Just after that RTS is the beginning of this subroutine. Instead of trying a G here, HUNT for the section of code that calls this subroutine. When this memory location is returned from the monitor, scroll through this code as described. This may be the place to enter. If we were to find a section of code at $0A00 that included a load (LDA) and store (STA) to border color (D020) that came between two return subroutines (RTS), we would HUNT for the section of code that called this subroutine (H 0801 9FFF 4C 00 A0 or H 0801 9FFF 20 00 A0). If, after the HUNT, we found this call at $0B00, we would try a G 0B00.

FLIPPING-OUT BASIC'S INTERPRETER:

1). Another good place to try looking for an entry point is at a section of code that will flip-out the BASIC INTERPRETER.

2). Try H 0800 9FFF A9 36. We are now searching for a section of code that will load the accumulator (LDA) with a 36. If a memory address is returned, scroll through that section of code to determine if a STA $01 follows the LDA instruction. This is usually a good place to try an entry point.

```
., 4000 A9 36    LDA #$36
., 4002 85 01    STA $01
```

In this example, the memory address $4000 would have been returned after the HUNT. In this example we would try a G 4000.

FILE LOAD ENTRY POINTS:

It is often possible to enter a program from a file load. One of the last programs to be loaded in game programs is usually the top scores file. These are usually easy to spot in the program through the I command. Programs such as these are usually called something like SCORE, HIGH, etc.. If you cannot locate a file of this type, try loading the last file that was listed on the directory.

1). Attempt to identify the file name through the I command.

2). Using the D command, locate where the file is being opened and try a G at that section of code.

```
.,193B A9 01     LDA #$01
.,193D A2 3F     LDX #$3F
.,193F A0 1A     LDY #$1A
.,1941 20 BD FF  JSR $FFBD
.,1944 20 C0 FF  JSR $FFC0
```

In this example we are setting a file name and opening a file. We would attempt a G 193B in this example.

TITLE SCREEN OR INSTRUCTION MENUS:

Use your I command to locate a section of code that contains title screen or menu options. Once located use your D command to scroll through the code. You will usually find various JMP (JUMP) instructions within this code. Investigate these areas for possible entry points. Try the G command at each JMP. This is where luck may lend a hand.

FINAL COMMENTS ON ENTRY POINTS:

The suggestions offered here should give you an idea of where to begin, but as we stated earlier, experience is the best teacher. Do not attempt an entry point within an area that is littered with question marks (?). This type of code is often data, such as graphics code, that cannot be interpreted by the monitor. Trying a G in an area of this kind will usually lock-up your computer. Look for your entry point in solid areas of code. Before attempting an entry, write down the address you are trying. If it works, you may forget the address and spend two hours trying to find that spot again.

Don't be afraid to try a spot that looks promising. The only thing you can lose is a little time.

BACK AND FRONT DOOR TECHNIQUES

Many of the newer schemes will not only check for a specific value, but will store the values returned from the error channel within the program. This code is then recalled and utilized for proper program execution. The value being checked could be a standard error, a non-standard sector, or whatever the programmer has stored on the disk and will later check for. A scheme of this type presents two problems. First we must find the area where the code will be stored and store the necessary values. Our second task is to keep the program from overwriting the values we stored with the new values that will be returned through the error-checking. We will try to make this clearer through the example that follows. Although this example is checking for an error 21, your program could be checking for other values. In the interest of space, the entire section of code will not be presented here. We have only included the code that is of interest to us.

ORIGINAL CODE:

```
CC26 4C B1 2C JMP $CCB1   - JUMPS TO THE SECTION OF CODE THAT
                            WILL LOAD AND MANIPULATE THE VALUES
                            RETURNED FROM THE ERROR CHANNEL
CC29 EA       NOP         - WILL BE CHANGED TO A 32 AFTER THE
                            ERROR CHANNEL IS CHECKED
CC2A EA       NOP
CC2B EA       NOP
CC2C EA       NOP
CC2D EA       NOP         - WILL BE CHANGED TO A 31 AFTER THE
                            ERROR CHANNEL IS CHECKED
CC2E 20 32 37 JSR $3732
CC31 85 FB    STA $FB
CC33 8E AB CC STX $CCAB
CC36 8C AC CC STY $CCAC
CC39 A9 00    LDA #$00
CC3B 20 BD FF JSR $FFBD   - SET FILE NAME
CC3E A9 0F    LDA #$0F    - DECIMAL EQUIVALENT OF 15 - THE
                            ERROR CHANNEL IS BEING OPENED IN
                            THIS SECTION OF CODE (15,8,15)
```

```
CC40 A2 08     LDX #$08     - DECIMAL 08
CC42 A8        TAY          - WILL SET Y REGISTER TO 15
CC43 20 BA FF  JSR $FFBA    - SETS LOGICAL 1ST AND SECOND ADDRESS
CC46 20 C0 FF  JSR $FFC0    - OPEN A LOGICAL FILE
CC49 A9 01     LDA #$01
CC4B A2 A2     LDX #$A2
CC4D A0 2C     LDY #$2C
CC4F 20 BD FF  JSR $FFBD    - SET A FILE NAME
CC52 A9 05     LDA #$05
CC54 A2 08     LDX #$08
CC56 A8        TAY
CC57 20 BA FF  JSR $FFBA    - SETS LOGICAL 1ST AND SECOND ADDRESS
CC5A 20 C0 FF  JSR $FFC0    - OPEN A LOGICAL FILE
CC5D 20 CC FF  JSR $FFCC    - CLOSE I/O CHANNELS
CC60 A2 0F     LDX #$0F
CC62 20 C9 FF  JSR $FFC9    - OPEN CHANNEL FOR OUTPUT
CC65 A0 00     LDY #$00
CC67 B9 A3 CC  LDA $CCA3,Y
CC6A F0 06     BEQ $CC72
CC6C 20 D2 FF  JSR $FFD2    - OUTPUT A CHARACTER TO CHANNEL
CC6F C8        INY
CC70 D0 F5     BNE $CC67
CC72 20 CC FF  JSR $FFCC
CC75 A2 0F     LDX #$0F
CC77 20 C6 FF  JSR $FFC6    - OPEN A CHANNEL FOR INPUT
CC7A 20 CF FF  JSR $FFCF
CC7D A4 FB     LDY $FB
CC7F 99 29 CC  STA $CC29,Y -  THE VALUE RETURNED THROUGH THE
                              ERROR CHANNEL WILL BE STORED AT
                              CC29,Y
CC82 20 CF FF  JSR $FFCF    - ANOTHER VALUE WILL BE RETURNED
CC85 99 2D CC  STA $CC2D,Y - THE VALUE RETURNED WILL BE STORED
                              AT $CC2D,Y
CC88 20 CF FF  JSR $FFCF    - GET ANOTHER VALUE
CC8B C9 0D     CMP #$0D     - LOOKING FOR A CARRIAGE RETURN
CC8D D0 F9     BNE $CC88    - IF NOT FOUND GO BACK AGAIN
CC8F A9 0F     LDA #$0F
CC91 20 C3 FF  JSR $FFC3    - CLOSE A LOGICAL FILE
CC94 20 E7 FF  JSR $FFE7    - CLOSE ALL FILES AND CHANNELS
CC97 A4 FB     LDY $FB
CC99 B9 29 CC  LDA $CC29,Y
CC9C 19 2D CC  ORA $CC2D,Y
CC9F C9 30     CMP #$30
CCA1 60        RTS
CCA2 23        ???
CCA3 55 31     EOR $31,X    - U1 COMMAND
CCA5 3A        ???
CCA6 20 35 20  JSR $2035      U 1 : 5 5 + 0 5 0 1 5 9 <>
CCA9 30 20     BMI $CCCB
CCAB 30 31     BMI $CCDE
CCAD 20 39 0D  JSR $0D39
CCB0 00        BRK
```

```
CCB1 AD 29 CC LDA $CC29   - WILL LOAD THE ACCUMULATOR WITH THE
                            VALUE FOUND AT $CC29 - AT THE
                            PRESENT TIME THERE IS A NOP STORED
                            AT THIS LOCATION
CCB4 4D 05 08 EOR $0905   - MANIPULATE THE VALUE FOUND AT $CC29
CCB7 8D 05 08 STA $0905   - STORES THE VALUE RETURNED AFTER THE
                            LDA AND EOR OPERATIONS
CCBA AD 2D CC LDA $CC2D   - LOAD THE VALUE AT $CC2D - AGAIN, AT
                            PRESENT, THE ONLY VALUE STORED IS
                            AN EA
CCBD 4D 11 08 EOR $0811   - EXCLUSIVE OR THAT VALUE
CCC0 8D 11 08 STA $0811   - STORE THE RESULT
CCC3 AD 2D CC LDA $CC2D   - SAME PROCESS AS EXPLAINED ABOVE
CCC6 4D 0B 08 EOR $090B
CCC9 8D 0B 08 STA $090B
CCCC AD 29 CC LDA $CC29   - SAME AS ABOVE
CCCF 4D 08 08 EOR $0908
CCD2 8D 08 08 STA $0908
CCD5 4C 5F 09 JMP $095F   - JUMP TO MAIN PROGRAM EXECUTION
```

One of the first steps in the "unprotection" process is to hunt for an error checking routine. In the code above, we find the U1 command located at $CCA3. Once the U1 is located, we begin examining the code around it and any JSR's or JMP's. We also look for any file-handling routines involving KERNAL calls (JSR $FF..). As you may see from the commented code above, the error channel is being opened (15,8,15). When we encountered the code above, we became immediately suspicious, because of the NOP's. Although NOP's are not unheard of in a finished program, they are a bit unusual. At $CC26, we find a JMP to $CCB1. It is this section of code that is used to load the values returned from the error channel.

Since we did not know what values were expected by the program, we ran the program from the original disk, and performed a RESET. After the RESET, we examined the code in this area and found that the NOP's had been changed. $CC29 now contained a 32, and $CC2D contained a 31. The program was checking and storing an error 21. Knowing what the values are is only half the battle. We must now find a way to insert these values into the program. Keep in mind that whenever this program is executed, it will check the error channel. If it does not find the error 21, it will not execute properly. We must also keep the new values from disturbing the code we will insert. To solve these problems, we changed the code to the following.

```
CCB1 A9 32    LDA #$32     - WILL LOAD THE ACCUMULATOR DIRECTLY
                             WITH THE VALUE IT EXPECTED TO FIND
                             AT $CC29
CCB3 EA       NOP
CCB4 4D 05 08 EOR $0905    -
CCB7 8D 05 08 STA $0905    -
CCBA A9 37    LDA #$31     - WILL LOAD THE ACCUMULATOR DIRECTLY
                             WITH THE VALUE IT EXPECTED TO FIND
                             AT $CC2D
CCBC EA       NOP          -
CCBD 4D 11 08 EOR $0811    -
CCC0 8D 11 08 STA $0811    -
CCC3 A9 37    LDA #$31     - LOAD THE ACCUMULATOR WITH THE VALUE
                             IT EXPECTS TO FIND AT $CC2D
CCC5 EA       NOP          -
CCC6 4D 0B 08 EOR $090B
CCC9 8D 0B 08 STA $090B
CCCC A9 32    LDA #$32     - LOAD THE ACCUMULATOR WITH THE VALUE
                             IT EXPECTS TO FIND AT $CC29
CCCE EA       NOP          -
```

By loading the values directly into the accumulator, it will not matter what is returned from the error channel. We have given the program the values it requires to run properly.

Another approach would be to insert a $32 at $CC29 and a $31 at $CC2D. We would then have to change the STA's at $CC7F and $CC85 to another area of memory. This would keep the program from storing the new values in place of the values we inserted.

This example illustrates the need to work with both forms of "unprotection". We use the "back-door" to find out what the program expects, and the "front-door" to make the necessary changes.

We have found that replacing the subroutine (JSR) to check the error channel with a NOP is all that is necessary in most programs that store an error code. This bypasses the error checking entirely. In a case like this you would allow the program to run normally, find the subroutine that checks the error, insert a NOP in place of the JSR, and save out the altered code. To locate the subroutine, look for a U1 or B-R, or file handling of any kind (KERNAL calls). This will often expose the error-checking routine. Don't assume that a program is going to be difficult to "unprotect". Often when you begin with this thought in mind, you tend to miss the obvious. If you find a CMP#$32 and change it to a CMP#$30 and the program still does not work, you are probably in the right area but have missed some code that would alter that area of memory. Keep an eye out for the following if bad blocks are being used:

DEC (DECREMENT) instructions that refer to the CMP instructions.

LDA#$32 (Load the accumulator) followed by a STA (Store accumulator) to the memory address that contains the #$32. Even if you altered the $32 in the CMP, this instruction would return it to the original value.

CMP#$32 and CMP#$31 - Always check for the second compare. CMP#$32 is only checking to see if an error is present, the programmer may go on to check for the specific error.

If you check the error section thoroughly, you should be able to spot this type of code.

ENCRYPTION:

Recently, we have been encountering programs that modify themselves when executed. The chapter in this manual called "ENCRYPTION" will give you an explanation of this procedure. Many of these programs utilize an AUTOBOOT that is stored in screen or stack memory. The boot is used to load a program that will do the error checking and then load and execute the main program. Some of these boot programs are difficult to capture during a run, because they take control of the computer. Normally, we would attempt to make the changes directly on the disk, but we are now finding that many of these programs are being modified once in memory. The code found on the disk is virtually useless to us.

In a situation of this kind, we find the "back-door" approach particularly useful. Most of these programs will write the new code to an area of memory that is easily accessible (0800-CFFF). We suggest the following procedure for encrypted programs which use bad blocks:

1). Load the program from the original disk.

2). Keep an eye on the disk drive error light. Once the light begins to blink, RESET your computer.

3). Use a machine language monitor to examine the code from $0800-CFFF. More often than not, you will find a complete error checking routine, along with the code necessary to load the main program.

4). At this point, you will have to alter the routine in the manner similar to the one described below. This example uses some alternate KERNAL calls that may be seen occasionally.

```
1000 20 AE FF JSR $FFAE - COMMAND SERIAL BUS TO UNLISTEN
1003 A9 08    LDA #$08
1005 20 B4 FF JSR $FFB4 - COMMAND SERIAL BUS TO TALK
1008 A9 6F    LDA #$6F  - COMMAND CHANNEL ($60 + 0F)
100A 20 96 FF JSR $FF96 - SEND SECONDARY ADDRESS AFTER TALK
100D 20 A5 FF JSR $FFA5 - INPUT BYTE FROM SERIAL PORT
1010 C9 32    CMP #$32  - HERE WE ARE CHECKING FOR A BAD  BLOCK
                          - IF WE CHANGE THE #$32  TO  A  #$30,
                          THE PROGRAM WILL  CHECK  FOR  A  GOOD
                          BLOCK INSTEAD  -  THIS  IS  THE  ONLY
                          CHANGE NECESSARY TO ELIMINATE THE USE
                          OF A BAD BLOCK ON THIS DISK
1012 D0 D9    BNE $109B
1014 20 A5 FF JSR $FFA5
1017 C9 30    CMP #$30  - HERE WE ARE CHECKING FOR A GOOD BYTE
1019 D0 D2    BNE $109B
101B 20 A5 FF JSR $FFA5
101E C9 0D    CMP #$0D  - CHECK FOR A CARRIAGE RETURN
1020 D0 F9    BNE $101B
1022 4C AB FF JMP $FFAB - COMMAND SERIAL BUS TO UNTALK
1025 60       RTS
```

5). Once the alterations to the code are made, you need only save out the altered code from memory and find an entry point for the program.

The examples above illustrate a check for bad blocks. This does not mean that all programs will be checking for these types of errors. As we indicated earlier, there are other values that may be checked. We recommend that you look for the standard errors first. The majority of the programs we have checked lately are disguising the initial code, but once it is exposed, you'll find most programmers are still checking for the standard errors.

FINAL COMMENTS:

If, after many hours of work, you are unable to "unprotect" a program, put it away for a while and work on something else Be sure to make good notes on what you tried. If you succeed in unprotecting a program, also make notes on the techniques you used. Write a check list of the procedures you have used and begin with those on each new program. Remember, experience is the key to the "back-door" approach. With each program you capture, you will learn something in the process. You may find that, armed with experience and fresh ideas, you can come back to a program that frustrated you earlier and be victorious.

HAPPY HUNTING!

I. Introduction

Allow me to introduce myself - I'm THE DOCTOR. In this chapter I'll tell you how I crack programs, and the reasons why you may wish to crack programs. I'm going to make some assumptions about what you already know. I'll assume that you've read the previous chapters in this Program Protection Manual, that you know how to use a machine language monitor program, and that you know what a disk editor is. And most importantly, you must have a sense of curiosity and a desire to learn the secrets of program protection.

Why crack programs? Cracking a program is more than just making a backup copy. It involves learning about the program - how it works and how to improve its operation. The real challenge is decyphering the program protection and then disabling or removing it. The program can be studied and modified as desired for personal use.

Other reasons for program cracking include making a backup copy of a protected program and eliminating the destructive drive head banging. Or to produce a file version of a program, allowing several programs to be put on a single disk.

II. Cracking Tools

I use a number of tools or instruments for investigating programs. These include both computer programs and hardware. Many of these are mentioned in previous chapters in this manual. Commercially available programs will be listed with the manufacturer's address at the end of this chapter.

1). Monitor program: A number of good monitor programs are available. An important requirement is that it is relocatable to different memory locations. My preference is DRVMON64 from Starpoint Software because it performs memory transfers between the computer and disk drive. Additionally, it will read data contained in the RAM underneath the computer ROMS. A second choice is MICROMON from Compute! which will do a relocatable machine language load as well as comparisons of ranges of computer memory.

2). Disk Sector Editor: A number of programs are available to edit disks and have been referenced in this manual. I use Peek A Byte 64 from Quantum Software and will give examples of its use. The key reasons I like the program are:

a. Many sectors may be stored in the computer memory, including files - I don't lose the data because I need to look at another sector.

b. The program will exit to BASIC, a monitor program, or my own decoding routines without losing my disk data or the disk editor.

c. Many monitor type features are built in.

3). Reset Switch: You can buy one or build one from the instructions in PPM Vol I. It's invaluable for recovering from crashed computers.

4). Cartridge Expansion Board: Buy one, such as Cardco's 5-slot or CSM's single slot, which allows cartridges to be turned on and off, as well as a switch to turn on the EXROM line. This will switch in a nonexistent cartridge by grounding the EXROM line and allow a reset out of programs which place the cartridge reset codes at $8000. Most boards include a reset switch.

5). Fastload Cartridge from Epyx: I like this program because it does fast loads of normal files, does not take up computer memory, and has an ML monitor with unique features built in. It can be disabled to avoid conflicts with protected programs. My version can be reactivated by SYS 57194 - this is an undocumented feature.

6). Copy Programs: I won't say much about specific copy programs because I haven't found any I really like. My favorite compendium of disk utilities is Di-Sector which includes DRVMON64. The Nibble copy program of Version 2.0 is already outdated, but the 3 minute copier is excellent. A second copy program is Omniclone from CSM Software which does a whole disk copy including most errors. One nibble copier I use is Diskmaker. Diskmaker allows one range of tracks to be copied and then requires the copy program to be reloaded. Note: it doesn't copy the data reliably when confronted with 27, 29, and 22 errors simultaneously.

7). Computers and Disk Drives: A second disk drive is helpful, but a second computer is even more helpful. I use one computer strictly for diagnostic programs such as Peek A Byte 64 and DRVMON64 and the second for loading the protected program. The disk drive can be swapped if only one is available. If the drive is swapped between computers, be sure to plug the serial bus cable into the second computer carefully so no pins on the connector are shorted. Remember that the computer and drive are on and that carelessness could damage either component.

8). Printer: A useful option for memory or disassembly listings.

III. Getting Started - The Initial Disk Examination

First I determine if the program is protected. Fortunately, many good programs are not. Load the program and follow what it does to the disk drive. Keep a notebook and take notes - you're doing an experiment. Does the drive head move over many different tracks? Does the head bang, and if so on which tracks? Does the whole program load at one time or does it access the disk while running? After the program is loaded, can you exit by pressing the STOP key and/or the RESTORE key, or can you reset back to BASIC using the reset switch? Can the directory be listed? The answer to these questions give clues to the program protection.

To examine head movement, I removed the cover from the 1541 drive (after the warranty expired) by unscrewing the (4) Phillips head screws in the bottom. The protective metal shield over the PC board is held in place by (2) Phillips screws on the left hand side. I placed a narrow strip cut from an adhesive disk label on the top and side of the support frame for the read/write head pad, just underneath the black plastic arm. I labelled every other track position after using a disk editor to move the head to each track by reading a sector on that track. Following the drive head is now easy. Put the covers back when not using this feature.

Next I usually try to make a full disk backup copy of the disk. The observations made during load help indicate what type of copy program is required. Head banging on one or two tracks usually means there are a few read errors on the disk which must be duplicated. Most weakly protected disks fall in this category. No head banging usually means no read errors are present. Omniclone or Di-Sector's 3 minute copier are my favorites for an initial backup attempt. It is important to keep a record of all the sectors or tracks on which errors occur for future use - the 3 minute copier does not copy errors, so the error writing utility must be used also. Whatever your choice of copy program, follow the instructions included with the program for the first copy attempt.

Now it's time to try the copy. First, put a write protect tab on the copy disk - some programs check for the write protect tab and erase the disk if it is not present. I use the write protect even if the original disk doesn't have one. Does it load properly and move the drive head in the same way? Does the program work or does it hang the computer? If it all works, congratulations, we're over the first hurdle. If not, more work remains which will be discussed later in this chapter. New copy protection routines download a new DOS (disk operating system) into the 1541 drive, enabling these programs to read the program disk with many errors and not bang the head. They may also read data on 1/2 tracks, past track 35, or in nonstandard format.

IV. An Examination of Disk Errors, the BAM, and Directory Sectors

OK, we've now produced a working copy of the disk. What do we do next? I normally investigate the disk further to determine what sectors are used, where the sector read errors are, and whether the BAM and directory are correct. The question is whether the disk can be used for storing any other disk files. An example is storing word processor text files on the same disk with the program. Most program disks don't allow it.

If the disk had no errors, or only a few, I first check the BAM and directory using a disk editor. The Peek A Byte program will load all the BAM and directory sectors into different memory buffers and display them in ASCII and HEX. I display the BAM sector, use the BAM function to display a map of free and used sectors for the disk tracks and sectors, and then use the printer function to print it. If the BAM is inconsistent, question marks are displayed above the track.

Next I determine what sectors on the disk have data on them. Peek A Byte will scan an entire disk and print a map showing any disk read errors for each track and sector in less than 2 minutes. In addition it will compare each sector with a single sector stored in the computer memory. I usually scan the disk for blank sectors by reading a blank sector off any disk into a free memory buffer. Typical blank sectors on 1541 disks have a 4B in byte 00 of the sector and 01's in all other positions. Track 1 may have other values in byte 00. Disks formatted on other drives usually are all 00's.

I scan the disk using Peek A Byte and generate a map of blank sectors and any read errors, and then print this also. A comparison of the free sectors in the BAM with the blank sectors gives a good indication of whether the BAM is correct. If the two maps are identical, the BAM is good. If there are non-blank sectors that don't show up in the BAM, it's likely they contain data used in copy protection or data not eliminated from the disk after files were scratched. A well known compiler program is distributed on a disk which contains company demo programs, scratched from the directory and BAM but recoverable!

After checking the BAM I look at the directory sectors. Program names are often modified by including print control characters such as the delete code $14 (appears as an inverse T in ASCII) after the shifted space code $A0. All characters after $A0 are ignored by the disk drive but are printed by the computer when listing the directory. If the directory doesn't list at all or repeats, I also check the track/sector link at bytes 00 and 01. These point to the next sector in the directory and in HEX should be 12 01 on the BAM sector (18/0) and 00 FF on the last directory sector. The 00 FF means that this is the last sector and that bytes thru FF are used. Sometimes these values are

changed to point to the first sector 12 01 so that the directory loops. See PPM Vol. I for further information.

I usually format a test or parameter disk for saving files of memory data and also individual sectors. (If I didn't do this before, I can exit from Peek A Byte without losing my sector data and format a disk using the DOS 5.1 wedge.) I write the individual directory sectors to track 1 of this test disk, using the same sector location, before altering the program copy. Then I use the editor to modify the BAM and directory sectors in memory and write them back to the disk. All we're doing here is making the directory listable by changing the inverse screen control characters to either $A0 or a normal letter or number.

After recording the BAM on the test disk, I use the Validate command to trace through all the files on the disk and generate a new BAM (the DOS wedge command for this is simply @V). The first time I do this, I keep the disk write protected so that a new BAM won't be written to the disk. If the Validate is successful, a 26,WRITE PROTECT ON,18,00 error will be generated when the error channel is read. If the light on the drive blinks, then a Validate error occurred. Read the error channel and record where the error occurred. I then restart Peek A Byte (SYS 49152), pick an unused series of eight pages or buffers, at $2000 or $3000 for example, and then use the built-in drive memory read function for pages 00 through 07 in the drive. The BAM generated is usually in drive page 07, but could be in 03 through 06. I save this sector to the test disk (and keep a log in my notebook) and then generate a new free sector map and print it. The new BAM can be compared to the one read off the disk by just changing the computer buffer displayed by Peek A Byte.

If the BAM on the disk and the one read from the disk drive match, then the BAM is correct for directory files; but be careful of nonblank sectors which show up as free on the BAM. The files are probably correct, too. If the BAMs don't match, then it's probably NOT safe to write data to the disk. I will sometimes modify the BAM of a disk using Peek A Byte to read the BAM sector (18/0). I edit this sector in the computer buffer by setting the four bytes corresponding to a particular track to 00. Track 1 starts at byte $04, track 4 at byte $10, etc. The BAM function displays the free sector map from memory, not the disk. When the tracks I wish to protect from files are correct on the map, the BAM is rewritten to the disk. The disk must be marked as an invalid BAM since a validation will free these sectors.

V. Program Cracking From Memory

Many programs are single load, that is, they do not access the disk drive after loading into memory. Before I proceed to remove read errors or protection from the disk, I check to see if the program can be 'downloaded' from memory. This is sometimes less work than removing errors from the disk. I'll be discussing primarily ML (machine language) programs, although some of the comments apply to BASIC programs. Please reread the chapters on using ML monitors, computer resets, vector addresses, and computer memory maps. It's necessary to understand how the reset affects the memory in the computer and I can't review all this material in a short chapter.

I'll go through the procedures I use step by step and explain the types of ML routines I look for. First, load a ML monitor (I use DRVMON64) at either $8000 or $C000. Use the fill command to fill most of the computer memory with a single value - I like 03 because very little ML code uses 03. If DRVMON 64 is loaded first at $C000, then I use a unique feature it has:

```
O 34              turn off Basic, Kernal and I/O devices
F 0803 BFFF 03    fill memory
F D000 FFFF 03
O 37              turn ROMs & I/O back on
```

Do NOT fill the memory where the monitor resides. Then reset the computer from the monitor using:

```
G FCE2      (same as SYS 64738 in BASIC)
```

Now load the protected program normally from the original disk. Try the STOP and RESTORE keys - do they exit to BASIC? If not, and the program is still working, try the reset button that you've installed. Many programs will reset to BASIC. Let's examine this case first. (If the computer hangs, ground the EXROM line using the expansion board switch, then press reset. Programs of this type will be discussed later.)

Reload the ML monitor to the same location originally used. Visually scan the memory and note which areas are still filled with 03 and which now have code. This can be done in blocks or continuously:

```
O 34
M 0800 FFFF    dump in HEX and ASCII
O 37
```

Now refill the memory with 03's and exit to BASIC. Then reload the original program and reset back to BASIC at the same point in the program operation. Load the ML monitor in at $8000 if $C000 was used first. Scan the $C000 - $CFFF memory range also.

If you find 03's at the $8000 or $C000 areas, you'll be OK. If other areas of memory are available instead, you may be able to use LLMON ($2000) or DRVMON ($0801 version).

Next we need a routine to transfer the memory in the range $0000-$0802, which is altered upon reset, to a free section of memory. If the area from $5000 to $7000 is free, then I use the following routine to move memory. Save to disk before running - the routine is self modifying to avoid changing zero page locations.

```
$6000: 78        SEI           ; prevent interrupts
$6001: A2 00     LDX #$00
$6003: BD 00 08  LDA $0800,X   ; save start BASIC first
$6006: 9D 00 6F  STA $6F00,X   ; store here
$6009: E8        INX
$600A: D0 F7     BNE $6003     ; loop
$600C: CE 08 60  DEC $6008     ; change pages
$600F: CE 05 60  DEC $6005
$6012: 10 EF     BPL $6003     ; continue to zero page
$6014: BD 00 DF  LDA $DF00,X   ; save I/O chips
$6017: 9D 00 5F  STA $5F00,X
$601A: E8        INX
$601B: D0 F7     BNE $6014
$601D: CE 16 60  DEC $6016
$6020: CE 19 60  DEC $6019
$6023: AD 16 60  LDA $6016
$6026: C9 D0     CMP #$D0      ; last I/O page
$6028: B0 EA     BCS $6014     ; continue until #$CF
$602A: A2 FF     LDX #$FF
$602C: 78        SEI
$602D: 9A        TXS           ; set stack pointer
$602E: D8        CLD
$602F: 4C F2 FC  JMP $FCF2     ; reset-skip CBM80 check
```

How do we jump to this routine? If we could reset from the program without hanging, then we can place the ROM reset code at $8000 to jump to our routine. Reread the chapter on resets and KERNAL vectors if you don't understand. Use the monitor to place the following code at $8000, where $6000 is the start of our routine.

```
8000: 00 60 00 60 C3 C2 CD 38 30
```

Now exit the monitor and reload the protected program. At the same point in the program operation, first try the RESTORE key. If the vector at $318 was not altered by the program, we'll jump to our save routine after the NMI saves the program counter and status register on the stack. The stack pointer is not saved by this routine. If RESTORE does NOT work and the program is still operating, try the reset button. This will correctly save memory from $2 to $8FF only. The I/O chips are altered by a reset, as are the values at $0 and $1. These values are normally 2F and 37, respectively, and should be corrected in the saved memory. Reenter the monitor program left

in memory, dump the memory as done before, and disassemble changed memory locations. Save each block of memory that appears to contain code to the test disk. Be sure to save the pages of memory moved from $0 to $8FF, and also $D000 to $DFFF if the RESTORE key was used. Look under the ROM's, too.

We must now examine the code to determine how to restart it (the entry point). If the RESTORE key jumped to our memory move routine, then the NMI vector at $318 was not altered and finding a good point to enter may be difficult. Since most programs are in a waiting loop reading the keyboard, I use to monitor HUNT command to search for either GETIN, $FFE4, or CHRIN, $FFCF and then examine the routines which call them. A good example is from a menu driven program waiting for a number from 0 to 9 to be entered:

```
$1000: 20 E4 FF  JSR $FFE4     ; read keyboard
$1003: F0 FB     BEQ $1000     ; repeat if no key
$1005: C9 30     CMP #$30      ; ASCII '0'
$1007: 90 F7     BCC $1000     ; repeat if < '0'
$1009: C9 3A     CMP #$3A      ; ASCII ':'
$100B: B0 F3     BCS $1000     ; repeat if >= ':'
$100D: 29 0F     AND #$0F      ; remove high nibble
$100F: 0A        ASL           ; double
$1010: AA        TAX           ; use X as index
```

There is more to this routine, but it gives the idea. In fact, after restoring the memory for this program, a JMP to this routine was all that was required.

It is not usually this simple. Sometimes the stack pointer must be restored first, as well as A, X, and Y. These can be saved when using the RESTORE key by putting the following at the beginning of the memory move routine - the address pointer and status registers were saved on the stack first:

```
SEI
STA $6080
STX $6081
STY $6082
TSX
STX $6083    ; save values in any convenient locations
```

If the reset button had to be used, then this information is gone since the 6510 registers are scrambled on reset. However, if the RESTORE key restarted the program after the initial program load (before we put our code at $8000), then the vector at $318 may be a good place to look for a starting address. If the vector points to a routine that restores initial memory and register values, then we may be home free.

Often values are stored either in unused color RAM, I/O, or VIC chip memory. Some of these values must be correct for the program to operate. Since this information is lost on reset, if the program does not work when the memory is restored, a search

of memory for routines which check these values is required. I like the Fastload cartridge monitor for this search because it will hunt for a range of values or disassemble only code which references a specific range of addresses. An example of such a routine is one that checked the seconds of the time of day clock at $DC09 for the value #$50, resetting the computer if different. Note that this routine would be erased by a reset since it is in the INPUT buffer ($0200-0258):

```
$0200 48          PHA           ; save A
$0201 AD 09 DC    LDA $DC09     ; TOD
$0204 C9 50       CMP #$50
$0206 D0 02       BNE $020A     ; reset if not equal
$0208 68          PLA           ; restore A
$0209 60          RTS           ; return
$020A 6C FC FF    JMP ($FFFC)   ; reset
```

Many programs now place the ROM reset codes at $8000 as a form of protection. Now a reset will typically hang the computer or erase memory and then reset to BASIC. However, the warm start vector at $8002 is often a good start address if RESTORE restarts the program. More drastic measures are now required. A modified HESMON cartridge, or a modified KERNAL ROM on a special board can be used to break out of the program. The NMI or reset vectors can be changed and the cartridge or Kernal switched in after the protected program is loaded. The use of these techniques is discussed elsewhere in this manual. I alter the NMI vector at $FFFA and reset vector at $FFFC in the RAM memory underneath the KERNAL ROM to point to my memory move routine. I first copy the ROMs into RAM with a monitor:

```
T A000 BFFF A000     ; Transfer BASIC to RAM
T E000 FFFF E000     ; Transfer KERNAL to RAM
:0001 35             ; turn off BASIC & KERNAL ROMs
:FFFA 00 60 00 60    ; change NMI & RESET vectors
X                    ; exit to BASIC
```

Now load the protected program. Typically this technique doesn't work because the program switches the ROMs back in. In this case I use a push button switch to temporarily switch in the RAM before trying either the RESTORE key or reset button. The original C-64, whose circuitry matches the Programmer's Reference Guide circuit diagram, and whose major chips are in sockets, can be bank switched easily. I connect the switch across pins 7 (HIRAM) and 14 (ground) of the PLA chip U17. The pin numbers start at 1 at the upper left, with the notch at the top. Do NOT attempt this unless you can understand the circuit diagram and the computer PC board layout - otherwise you could burn out the computer. Remember, this or any modification can cause damage, so be careful. I then press this button just before trying the reset so that the vector in RAM is used instead of ROM. Some programs place vectors in RAM or clear memory, attempting to defeat this technique. Often they do this right away in the boot, in which case you may be able to prevent it from happening.

The program memory can be restored by using a BASIC program to load the various blocks as ML files. However, since the first few pages of memory must be restored with an ML move routine with interrupts disabled, I prefer an all-ML routine. An example is shown below with comments - it was not necessary to restore the screen or I/O values. The program ended before $1A00.

```
$1C00: 78        SEI           ; prevent interrupts
$1C01: A5 BA     LDA $BA       ; use current drive no.
$1C03: 8D BA 17  STA $17BA
$1C06: A2 00     LDX #$00
$1C08: BD 00 1A  LDA $1A00,X   ; restore memory from
$1C0B: 9D 00 03  STA $0300,X   ; $0300 to zero page
$1C0E: E8        INX
$1C0F: D0 F7     BNE $1C08     ; loop through page
$1C11: CE 0A 1C  DEC $1C0A     ; hi byte of LDA address
$1C14: CE 0D 1C  DEC $1C0D     ; lo byte of STA address
$1C17: 10 EF     BPL $1C08     ; loop to page zero
$1C19: BD 00 1B  LDA $1B00,X   ; restore code
$1C1C: 9D E8 07  STA $07E8,X
$1C1F: E8        INX
$1C20: D0 F7     BNE $1C19
$1C22: AE 4F 1C  LDX $1C4F     ; unused memory location
$1C25: 9A        TXS           ; restore stack pointer
$1C26: AD 50 1C  LDA $1C50     ; status register
$1C29: 48        PHA           ; and save on stack
$1C2A: AD 51 1C  LDA $1C51     ; restore A, X, Y
$1C2D: AE 52 1C  LDX $1C52
$1C30: AC 53 1C  LDY $1C53
$1C33: 28        PLP           ; restore status reg.
$1C34: 4C 53 0C  JMP $0C53     ; use your own address
```

If the reset button was used, then all the 6510 register values were lost. It may not be necessary to restore A, X, Y or the status register if the program routines reinitialize them. The stack pointer must be reset, however, using either detective work or trial and error.

A BASIC loader less than #$E8 bytes long can be used to jump to this routine. Just type NEW from BASIC, then type in the loader, which must include a SYS to this routine. The ML monitor can be used to move the various memory blocks adjacent to the main program and the ML move routine, and to save the whole program starting with 0801. Your ML move routine should use the addresses appropriate for your program, not those in the above example.

VI. Program Cracking From Disk

Many programs today are multiple load, that is, they access the disk drive many times for data or one of a collection of programs. Other programs are heavily protected in the computer memory, or use altered DOS routines that are downloaded into the 1541 drive to read tracks with read errors. These programs must usually be tackled on the disk. Before continuing with this section, please reread the material in this manual on read errors, altered DOS, autoboot routines, and undocumented op codes.

You should already have made a best copy of the protected disk, made a list or map of all the sectors with either data or errors, and formatted a test disk for files and sector data. If you haven't done this, reread the first portion of this chapter and do the initial examinations of the protected disk.

Since many programs use altered DOS routines as a form of protection, I load the program first and wait until either the main menu comes up or the program is fully loaded. I then disconnect the drive's serial bus cable (carefully, it's still turned on) and reset the computer. Since I have two drives, I connect the second drive to the computer and load Peek A Byte 64, DRVMON64 (32768), and MICROMON (36864). If you have two computers, it's easier to leave these programs in memory in one computer all the time. I then reconnect the first drive and use the DOS 5.1 wedge to read the error channel. If it responds, we can proceed to read the drive memory. If it doesn't, I try a different drive number using the wedge @#8 or @#9 command. One program changed the drive number to 11. If it still doesn't respond, then the drive may be using altered serial bus communication, which is beyond the scope of this chapter.

I prefer using Peek A Byte or DRVMON64 to read the disk drive RAM memory from $0000 to $07FF - the I/O chips and ROM aren't important now. Peek A Byte displays whole pages of memory in HEX or ASCII, which I find easier than using only a monitor. I typically set MEMSIZ ($0283-0284) and FRETOP ($0033-0034) first to protect my ML code from BASIC, and then read the drive memory into the computer memory above BASIC. If you have only one drive and one computer, the following BASIC program can be entered from the keyboard and will put the data at $4000.

```
1 POKE55,0:POKE56,64:POKE51,0:POKE52,64
5 OPEN15,8,15 : REM  USE CORRECT DRIVE NUMBER
10 L=0:FOR H=0 TO 7
20 PRINT#15,'M-R'CHR$(L)CHR$(H)CHR$(0)
30 FOR I=0 TO 255
40 GET#15,A$
50 POKE 256*(64+H)+I,ASC(A$+CHR$(0))
60 NEXT:NEXT
```

Save the drive memory to the test disk as a ML file using the monitor. Now inspect the drive memory with Peek A Byte (or monitor). In particular look for ML routines left in the stack, typically after $150, remnants remaining after $200, and in $300 to $7FF. Usually $700 will contain a copy of the BAM sector, one page will have a sector from the directory, and one page will have the last sector loaded - $600 is typically blank. Disassemble anything that looks like ML code. If references are made to either $1800, $1C00, or $1C01, then that memory page probably contains an altered DOS routine. Write these this memory individually to any free sectors on track 2 of the test disk and make a log book entry of locations. We'll get back to this data later.

Most protected programs use an autoboot which loads in the computer either at $0102, at $02A7-$0304, or at $032C. The first type is a stack loader which places all 01's or 02's on the stack so that the RTS at the end of the load returns to $0102 or $0203. The second type of loader alters the BASIC vectors starting at $0300 so that when the load is finished and it returns to BASIC, it will jump to the ML routine instead. The third loader modifies the CLALL vector at $032C to jump to the ML routine since CLALL is called after finishing a load. See the chapter on autoboots for more info.

I use Peek A Byte to read the first directory sector which lists the autoboot file. The first track and sector of the file are given at bytes 03 and 04 of the directory entry. I then read the first file sector and note its load address, which is specified (low byte/high byte) by bytes 02 and 03 of the sector. Since the initial loader is usually only one or two sectors long, I disassemble the code and try to determine the type of loader it is, before moving it to its correct location. If the code loads at $0102 normally, I use MICROMON to load it at, for example, $1202. Then when I disassemble the code, most of each address lists correctly. If the code disassembles OK, we're golden. Many programs are now encrypted, unfortunately (see the chapters on encryption and decryption)

Another technique used is undocumented opcodes (see the chapter on these). An example is the following code which loads at $102 and uses a stack return to $102.

```
$0102: 18        CLC
$0103: A2 0A     LDX #$0A
$0105: 3F 00 01  RLAN $0100,X ; undocumented code
$0108: E8        INX
$0109: D0 7D     BNE $0288    ; incorrect branch
$010B: rest of code
```

The code won't disassemble this way since the code 3F is undocumented. It first rotates left: ROL $0100,X (=$0A), and then does an AND with the accumulator. In this case we don't care about the accumulator. The important data is that $010A is rotated from $7D to $FA since the carry is clear. Now the

branch goes to $0105. If this code is stored at $1102 instead, then transfer the code from $1102 - $110A to $1202. Change bytes $1207 to $12 and $120A to $FA. Terminate with an RTS to execute the code from Peek A Byte, or a BRK if from the monitor. After executing this code at $1202, the rest of the code at $1102 should be decyphered.

Now disassemble the resulting code and look for:

1. 'U1' or 'B-R' (or the reverse, i.e. 1U) which read a sector in
2. 'B-E' (or reverse) which reads the sector into the disk drive and executes
3. KERNAL routines which load files, open drive channels or transfer data

I use the Peek A Byte disassembler because it shows the ASCII characters. Save this decoded sector to a free sector on the test disk for further reference.

Often all the code is read from the disk by sectors and is encrypted. A routine may update the track and sector in the command sent to the drive, or a routine may be loaded into the drive using 'B-E' or 'M-W', or an '&' file may be opened. If 'M-W' is used, then the code may be found in the computer memory after doing a reset. If 'B-E' is used, I use Peek A Byte to read the sector into a computer buffer. Often it must be decoded in a fashion similar to the first loader. Again, after decoding, write the sector to the test disk. An '&' file can often be opened in order to read the data into the drive. The drive memory can then be read with Peak A Byte or DRVMON64. Put a write protect tab on the disk first, one program I've seen tried to format the copy disk. From BASIC use:

```
OPEN 15,8,15,'&*':CLOSE15: REM DOES FIRST '&' FILE
```

The routine read into the drive typically reprograms DOS so that a sector with an error can be read, both verifying the error was present and reading in new data. One must understand how the drive works before one can crack these schemes. Reread the chapters on disk protection schemes. As an example, I followed the read routine in the drive and found the last normal sector read into the drive. I read this sector with Peek A Byte and decoded it. The routine read in a page of drive memory necessary to the program. It was read from a track with different track densities by first identifying the track header, then reading in the disk nibble data. The key portion of this routine was:

```
          jsr sync          ; find sync & change density
          ldy #$bb
wait1     bvc wait1         ; for data
          clv
          lda $1c01         ; read byte
          sta $0100,y       ; store in buffer
          iny
          bne wait1         ; loop
wait2     bvc wait2
          clv
          lda $1c01         ; read byte
          sta ($30),y       ; store in buffer
          iny
          jsr sync          ; find sync & change density
wait3     bvc wait3
          clv
          lda $1c01         ; throw away
wait4     bvc wait4
          clv
          lda $1c01         ; read byte
          sta ($30),y       ; continue storing
          iny
          cpy #$83          ; don't finish buffer yet
          bcc wait4
          jsr sync          ; change density again!
          ...
          code continues
          ...
sync      lda $1c00
          sec
          sbc #$20          ; affects density bits 5 & 6
          sta $1c00         ; change density
waitsync  bit $1c00         ; test bit 7
          bmi waitsync      ; wait for sync
          lda $1c01         ; throw away byte
          clv
          rts               ; done
```

This routine was modified to eliminate the density change and to read the whole sector without intervening sync bytes. Peek A Byte was used to re-encrypt the sector data and write it back to a copy of the disk. The sector data left in the disk drive was written to the disk copy on the same track as the original error track. The disk was then fully copyable with normal backup routines.

VII. Summary of Cracking Techniques

Initial Examinations:

1). Make the best back up copy possible and format test disk.
2). Check the original disk for errors and record them.
3). Make a map of nonblank sectors.
4). Check whether the BAM is correct and compare to nonblank sectors.

Breaking Program Out of Memory:

5). Look at warm start vector for possible start address.
6). Search for keyboard routines corresponding to menu.
7). Use monitor to copy code out of memory.

Breaking Disks:

8). Use monitor and Peek A Byte to examine initial loaders.
9). Write decoder to decypher encoded sectors.
10). Look for KERNAL routines which transfer data to/from drive.
11). Look for 'U1:', 'B-R:', 'M-W', 'B-E', and 'M-E' commands.
12). Read any sectors referenced in previous routines.
13). Look for altered DOS routines.
14). Modify routines, reencode, and write back to disk.

I know this summary is cursory and does not cover many situations. I tried to give a flavor of the way I approach cracking programs, the tools I've found useful, and the types of code I look for. If you've read and understood all the other chapters in this manual, then get out and practice. That's the only way you'll learn because just about every protection routine I've seen lately has been different. Good luck!

VIII. ADDRESSES

1. DI-SECTOR and DRVMON64
   Starpoint Software
   Star Route 10
   Gazelle, CA 96034

2. PEEK A BYTE 64
   Quantum Software
   P.O. Box 12716
   Lake Park, FL 33403-0716

3. MICROMON
   Compute! Publications
   P.O. Box 5406
   Greensboro, NC 27403

4. DISKMAKER
   Basix
   Box 31209
   Santa Barbara, CA 93130

The general idea in program protection is to tie the proper functioning of a program to a physical object. This physical object is usually the disk, tape or cartridge that the program was supplied on, but it can also be some other object like a dongle (key). If the physical object is hard to duplicate, the spread of illicit copies will be severely restricted. This two-sided nature of program protection provides us with two corresponding ways to proceed when we wish to obtain an archival copy: reproduce the physical protection or disable the program code that checks for it.

Up to this point, on the Commodore 64, it has been fairly simple to copy the physical protection. We've all had the experience of buying a piece of software, only to discover that it defeats all the copy programs we can find. If we waited a little while, however, there soon appeared a new copy program which could handle our tough case. Almost simultaneously, it seems, we would also come up against a new protection scheme which our 'latest and greatest' copy program couldn't handle. While it seems that this process could go on forever, we may be nearing a time when the physical protection will be beyond the power of our home equipment to reproduce. New schemes, such as laser-encoding disks (burning a spot on the disk in a precise place) or putting special circuitry in cartridges, are a whole quantum leap ahead of current schemes.

What about our other alternative, tracing down the protection code and disabling it? Up to now, this too has often been a very simple matter. Many programs only check for the physical protection once, right at the beginning of the boot process or just before jumping to the main menu. In such cases, we can usually lift a working copy from memory after the program starts. With the right entry point we can then start the program ourselves past the protection check. We may also be able to find out precisely how the program does its check, and alter it so that it passes on an unprotected copy. If you are a PROGRAM PROTECTION NEWSLETTER subscriber, you know that this often involves changing just a few bytes, or even just a single byte.

We can't expect things to be this simple forever, though. There are already programs which check the protection each time they leave the main menu or go to a new screen. There are even programs which appear to work for a while, and only 'crash' when you reach a certain screen or try to save your work. We have also seen encrypted programs and programs using undocumented opcodes. In short, the protection code is becoming better integrated into the normal functioning of the program, more deeply 'buried', as time goes by. If we don't want this evolutionary process to leave us behind, we have to jump on the

bandwagon NOW. In this chapter we'll take an introductory look at the process of TRACING PROGRAMS.

The skill of being able to trace a program has many added benefits beyond just making straight backup copies. Many programs that take up a whole disk in their original form can be reduced to a few files once the protection method is broken. Not only can you save disk space this way, you can also organize your software in a logical fashion. You can put similar programs together, or put data on the same disk with the program that uses it. Another thing you may want to do is modify something in the regular functioning of the program. It might be as minor as changing screen colors or as major as fixing a serious shortcoming/bug. You may be able to depend on the PROGRAM PROTECTION NEWSLETTER or knowledgeable friends for help in accomplishing some things, but these sources can't cover everything you might want to do. If you want something done, the saying goes, sometimes you just have to do it yourself.

One more area where experience in tracing programs is useful is in writing your own programs. Most of us bought our computers with at least a vague intention of learning to program them someday. A fact that seems to be routinely ignored by books and courses on programming is that programmers spend far more time debugging their programs than designing them to start with. Although the best idea is always to reduce debugging time to a minimum by careful design, the best laid schemes of mice and higher-order mammals do go astray with alarming regularity. Since the skills involved in analyzing and tracing a protection scheme are the same skills required to debug your own programs, this is as good a way to learn as any.

Convinced? OK, let's go for it. To remain applicable to program protection schemes as well as programming in general, I'll restrict my examples to machine language. Thus your number one priority is a working knowledge of 6510 machine language (ML). Many people, even some who are proficient in BASIC, feel that machine language is too hard for them to ever learn. I feel that with a little help, if you really want to learn it and you put some time into it, there are few other limits to what you can accomplish. You should learn to be comfortable with reading and writing ML (or more accurately, its assembler mnemonic counterpart). This doesn't mean you have to have the hex opcodes memorized (although that never hurts either). In this respect Commodore owners (and Atari and Apple owners as well) have been let off rather easily; the 65xx family (6502, 6510, etc.) instruction set is by far the simplest of any processor still in widespread use. It's a nice small pond to be a big fish in, if you catch my drift.

In addition to 6510 ML, it is important to learn a bit about how the microprocessor interrelates to the rest of the C-64 hardware and firmware. This includes such things as the KERNAL and BASIC ROMS (firmware); the memory manager (PLA); a little

about the VIC and CIA chips; and similar knowledge about any peripherals we may be dealing with (usually the disk drive). If this partial list already sounds overwhelming, don't be discouraged. You can pick up the knowledge you need as you go. In fact, this is how many successful programmers got where they are today. As you encounter new schemes and new hardware you naturally learn more and more of those very things you need to know.

Finally, you should equip yourself with the right tools. A basic requirement is the C-64 Programmer's Reference Guide. Other suggestions include additional reference books on ML and the C-64 such as The Anatomy of the C-64 by Abacus; a good ML monitor or selection of different monitors on disk and cartridge is important; some utility programs such as a disk sector editor and perhaps an assembler package; and a printer or second computer system if possible.

Don't forget about a good working environment too, with space to work where you can pace and mutter to your heart's content without disturbing normal people. A friend or two is also very helpful, even if they aren't wizards yet either. There is one more thing many people overlook: good habits. Use write protect tabs on originals and important copies. Label disks meaningfully. Be careful with your disks. Always have at least one backup, but keep down the number of obsolete copies. Comment your listings for later reference. Take notes.

The last item deserves some extra attention. You should not depend on your memory alone. You can waste a lot of time trying to remember things like what you have and haven't tried yet, which experiment had which result and where the subroutine you're in got called from. Write it all down in some fashion. Use your own shorthand. Draw diagrams (they're worth a thousand words, you know). Work methodically. Try to think of yourself as a scientist - a COMPUTER scientist. Buy yourself a white lab coat (just kidding).

Assuming you have the requisite desire, time and equipment, we are ready to begin. The place to begin is at the beginning. The program has a main ENTRY POINT that is our entrance into its inner workings. By following the path the program itself takes, and taking into account the computer's initial condition, we can see how it does what it does. In other words, GIVEN ENOUGH TIME AND INFORMATION, we can trace the program's path with 100% accuracy.

Let's start with the computer's initial state. Part of our job is to determine what is important and what is not important to the proper functioning of the program. Some programs are picky about what state the computer is in initially; other programs don't care. To reduce the number of unknowns we should always start our test procedure with a clean slate. Although sometimes a simple RESET is all we need, to really be safe from 'hang-overs' we should start with a power-up.

Next, you should fill as much of memory as possible with a dummy byte before loading the program to be examined. DON'T use $00 bytes; they are just too common, especially with programs that clear memory. My personal favorite for a dummy byte is $99. Although it looks like it might crop up often because of the 9's, remember this value is in hex. Hex $99 converts to an obscure 153 decimal, which seems safe enough. Also, if you run into some 99's while disassembling code, they are listed as STA $9999,Y. You can scroll through these faster than $00 (BRK) bytes, which take up one line per byte. Finally, when you 'Interpret' memory with a monitor, 99's show up as an easy-to-see set of vertical lines, and they list as PRINT statements in BASIC!

I make it a point to fill all non-essential memory during my first tests on a program, at least until I know what areas are used. On the program disk is a program called 'FILL'ER UP'. This fills the following areas of memory with $99 bytes, and optionally loads a program file when finished:

1). The entire stack ($0100-01FF).

2). The operating system input buffer ($0200-0258).

3). Some RS-232 and tape areas ($92-97, $9B-B1, $B4-C4, and $0293-$02A5).

4). The vicinity of the cassette buffer ($0334-03FF).

5). Screen memory ($0400-07E7). You'll see reverse Y's appear when this happens.

6). Other miscellaneous low-mem areas ($02A7-02FF and $07E8-07FF).

7). All of the BASIC area, except where the FILL'ER UP program itself resides ($0800-02C1)

8). The RAM under the BASIC ROM ($A000-BFFF).

9). The free area from $C000-CFFF.

10). All of color RAM ($D800-DBFF). Screen characters will turn brown when this happens.

11). The RAM hidden under the I/O devices and character ROM ($D000-DFFF).

12). The RAM under the KERNAL ($E000-FFFF).

'FILL'ER UP' resides at the beginning of the BASIC area. You load it with LOAD "FILL'ER UP",8 and execute it with RUN. Basically, it fills all of RAM except some essential low memory and the I/O devices (the VIC, SID and CIA chips). Following this it jumps into the BASIC warm-start process via the vector at $A002. This will return you to BASIC. If you want the routine to load the boot for the program you are investigating, all you have to do is change the JMP $(A002) at $086B with 3 NOP's and put the boot program name into memory starting at $0890. As long as you leave the $A0's at the end of the program name, you won't even have to specify the length of the boot program's name. Finally, you need to put a JMP statement at $0889 to start up your program.

One more thing is necessary before we look at the code. We need to have some idea of what to expect. If it's on disk or tape, does the boot load into the BASIC area (LOAD "NAME",device) or does it load somewhere else (LOAD "NAME",device,1). It may run automatically when loaded (autoboot) or you may have to do a BASIC RUN or SYS command. If the program is on cartridge, you can start it with a power-up, a RESET, or possibly with the RESTORE key or a direct SYS. (Note that many cartridges routinely clear to 00's the low mem areas filled by 'FILL'ER UP'.)

If it is disk-based, observe what the disk drive does during the load process. If it goes back and forth from track to track or to the directory often, it may be loading several short program files. If the red access light flickers, the program may be loading random files via the 'U1' command. If a tape-based program starts and ss many times, it may be loading a data file. Basically, you should summarize the start-up process and note any peculiarities for later reference.

I sometimes use a watch with timing functions to time disk loads. The following rules of thumb can help you estimate how many blocks on the disk are being loaded and how much memory is being used:

1). The normal load speed is close to one block per second on a straight load; slower if moving from track to track a lot.

2). Each block on the disk is one page of memory (usually only 254 bytes). That's 1/4 K. Four blocks make up 1K and sixteen blocks make up 4K, which is one 'letter' of memory (from $C000 to $D000 is one 'letter', for example).

Note carefully what happens when the program starts. Does the screen change colors, flash garbage briefly or shrink to 38 columns? Does the sound chip make a brief noise, especially in programs that DON'T use sound? These things may be a sign of some protection value being stored out into the VIC (video) or SID (sound) chip in a sloppy fashion. If you see a brief screen of garbage, the program may relocate VIC screen memory or store some data on the screen. How long does the actual start-up

take? If the program takes an appreciable time after it's loaded in (disk) or after the computer is RESET (cartridge) it may be decrypting itself. A cartridge may also be downloading itself to RAM.

Having summarized the start-up procedure and noted any peculiarities, many people RESET the computer at this point and examine memory with a monitor for an entry point into the code. I'll assume that you've tried this unsuccessfully, or that you just want to learn how the program works. Go back to the boot. If it's an autoboot disk program, figure out which type (see the chapter on autoboots). If the boot is in BASIC, trace that until you come to a SYS into the ML code. For cartridges, the cold-start address at $8000 or $A000, or the RESET vector at $FFFC (MAX cartridges) is the place to start (see the chapter on cartridges).

Once we know where the program starts, we need to follow it through, making notes of any JMPs or JSRs. It is also especially important to keep a running list of the memory locations changed by the program. For my example, I'm going to use the 'FILL'ER UP' program. It makes use of several different types of addressing, has a table-driven structure, and optionally does a program load. Here is the program listing:

BASIC CODE: 10 SYS2061 (=$080D)

ML CODE:

```
080D 78          SEI           Disable IRQ interrupts
080E A9 34       LDA #$34
0810 85 01       STA $01       Switch out ROMs and I/O
0812 A2 FF       LDX #$FF
0814 9A          TXS           Start stack at $FF
0815 E8          INX
0816 BD A0 08    LDA $08A0,X
0819 85 FC       STA $FC       Hi-byte, start of fill
081B E8          INX
081C BD A0 08    LDA $08A0,X
081F 85 FB       STA $FB       Lo-byte, start of fill
0821 D0 04       BNE $0827     Branch if not zero
0823 C5 FC       CMP $FC       Check hi-byte too
0825 F0 26       BEQ $084D     Terminate if both zero
0827 E8          INX
082E BD A0 08    LDA $08A0,X
0831 85 FD       STA $FD       Hi-byte, end of fill area
0833 A9 99       LDA #$99      Filler byte
0835 A0 00       LDY #$00      Start Y index at 0 offset
0837 91 FB       STA ($FB),Y   Store A indirect, Y-indexed
0839 A4 FB       LDY $FB
083B C4 FD       CPY $FD
083D D0 06       BNE $0845     Not done with this area
083F A4 FC       LDY $FC
0841 C4 FE       CPY $FE
0843 F0 D0       BEQ $0815     Done with this area
0845 E6 FB       INC $FB       Next byte to fill
0847 D0 EC       BNE $0835
0849 E6 FC       INC $FC       Next page for fill
084B D0 E8       BNE $0835     Fill next area
084D A9 37       LDA #$37
084F 85 01       STA $01       Switch ROMs & I/O back in
0851 A9 00       LDA #$00
0853 85 FB       STA $FB       Lo-byte, color RAM addr.
0855 A9 D8       LDA #$D8
0857 85 FC       STA $FC       Hi-byte, color RAM addr.
0859 A9 99       LDA #$99      Fill value
085B A0 00       LDY #$00      Reset index
085D 91 FB       STA ($FB),Y   Store A indirect, Y-indexed
085F C8          INY           Next byte
0860 D0 FB       BNE $085D
0862 E6 FC       INC $FC       Next page
0864 A4 FC       LDY $FC
0866 C0 DC       CPY #$DC      Compare to end page
0868 D0 F1       BNE $085B     Branch to continue fill
086A 58          CLI           Done; enable IRQ
086B 6C 02 A0    JMP ($A002)   Warm-start BASIC via vector
                               (remove to autoload file)
```

```
086E A9 0F      LDA #$0F      CODE TO AUTOLOAD A FILE
0870 A2 08      LDX #$08
0872 A0 0F      LDY #$0F
0874 20 BA FF   JSR $FFBA     Open 15,8,15
0877 A9 10      LDA #$10
0879 A2 90      LDX #$90
087B A0 08      LDY #$08
087D 20 BD FF   JSR $FFBD     File name at $0890-089F
0880 A9 00      LDA #$00
0882 A2 FF      LDX #$FF
0884 A0 FF      LDY #$FF
0886 20 D5 FF   JSR $FFD5     Load file
0889 6C 02 A0   JMP ($A002)   Warm-start BASIC (or change
                                to program's start addr.)
```

MEMORY TABLES

```
0890  A0 A0 A0 A0 A0 A0 A0 A0    File name
0898  A0 A0 A0 A0 A0 A0 A0 A0

08A0  00 92 00 97 00 9B 00 B1    Fill area pointers
08A8  00 B4 00 C4 01 00 02 58      (Hi-byte, lo-byte)
08B0  02 93 02 A5 02 A7 02 FF
08B8  03 34 07 FF 09 00 FF FF
08C0  00 00                      Termination indicator
```

In this program, we start with a BASIC SYS to 2061 ($080D). The first thing the ML code at $080D does is disable IRQ interrupts. This is usually a sign that the program wants to have complete control of the computer to do something special. In this case, it is going to reconfigure memory so that all the ROMs and I/O devices are switched out. An IRQ would crash the program if it occurred, since the IRQ routine is in KERNAL ROM.

When tracing programs it is very important to keep track of the current value of the registers (AC, XR, YR etc.) at each point in the program. You should also pay close attention to any stack operations (JSR, RTS, PHA, PLA, PHP, PLP). The 'FILL'ER UP' program doesn't use the stack at all; we'll see an example of manipulating the stack later. Along with the registers and stack, you should keep a running list of the current values of any memory locations used by the program. I make a chart with the registers and memory locations labeled across the top. When the program stores into a new location, I add a column for it. Down the paper I enter the new value in the appropriate column whenever it changes. Related changes should be entered on the same line. You can make notes along the side at important points. Here is what the first few lines of a chart for this program looks like:

SP	AC	XR	YR	$01	$FB	$FC	$FD	$FE
xx	xx	xx	xx	37	00	00	00	00
	34			34				
FF		FF						
	00	00				00		
	92	01			92			
	00	02						00
	97	03					97	
	99		00					
			92					
			97					

While it is not practical to write down ALL memory changes, especially when the program loops, you can use this technique as needed to record the important changes. Thus when you get to a statement such as STA ($FB),Y at $0837, you can determine what the program will do. This statement uses what is called INDIRECT, INDEXED addressing. The location given ($FB) is NOT where the accumulator will be stored; instead, the two-byte CONTENTS of this location are used as an INDIRECT BASE ADDRESS, to which the current contents of the Y-register (the INDEX) are added. The first time this statement is executed the result is to take $0092 and add $00 to get $0092. This is where the contents of the accumulator ($99) will be stored. Indirect, indexed addressing can only use the Y-register for an INDEX; the BASE ADDRESS must be in zero page $0000-00FF ($00FB in this case).

Another type of addressing used in this program is absolute, indexed by X. This is used at $0816, for instance. The term absolute refers to a directly specified two-byte address ($08A0). The X-register is used this time as an index (Y can also be used to index absolute locations). Thus the contents of X ($00) are added to the address $08A0 itself (not the CONTENTS of $08A0) to arrive at the address to load the accumulator from, $08A0. X will be used to step through the pointer table at $08A0. As the program executes further, X will be increased. As long as we have been keeping track of the value of X, we can always tell what pointer will be selected. This program uses these pointers to mark the beginning and ending of areas to be filled with $99. Some other programs may use tables of pointers to jump into different subroutines at various points.

A third type of addressing used in this example is just called indirect. This is illustrated by the JMP ($A000) at $086B. Note that this does not involve any indexing. The two-byte contents of $A002 point to the BASIC warm-start routine to jump to after the program is done executing. Unlike this example, many times a program will use different types of addressing in order to make tracing more difficult. Be sure you understand the different types of addressing.

Now let's look at an example which uses the stack in an unorthodox way. When a program executes a JSR (Jmp to SubRoutine), the two-byte address of the last byte of the JSR statement is put on the stack. The hi-byte is pushed first, then the lo-byte. Since the stack grows backwards from $01FF to $0100 these bytes will appear in the familiar lo-byte, hi-byte order if you examine stack memory from a monitor. When the next RTS (ReTurn from Subroutine) is executed, the address is pulled off the stack, incremented, and used to jump back to accomplish a return. It is very easy, however, to manipulate the stack so that an RTS jumps you anywhere you like. Take a look at the following code, which might be part of a boot:

```
1000 A9 4F      LDA #$4F
1002 48         PHA  PUSH THE 'A' ON TO THE STACK
1003 A9 FF      LDA #$FF
1005 48         PHA  PUSH THE 'A' ON TO THE STACK
1006 20 90 10   JSR $1090
1009 4C 00 9F   JMP $9F00
 .
 .
 .
 .
1090 68         PLA  PULL THE 'A' FROM THE STACK
 .
 .
10FE 68         PLA  PULL THE 'A' FROM THE STACK
10FF 60         RTS
```

At first it looks like the program will do a JSR $1090 and then a JMP $9F00 after it returns. This may lead you to think that the entry point into the main code is at $9F00. On closer examination, however, we see that something else entirely will actually happen. The first section of code pushes two bytes onto the stack with PHA (Push Accumulator), namely $4F and $FF. When it calls the subroutine at $1090, the address $1008 (return address minus one) is pushed onto the stack automatically. The first thing the subroutine does is pull a byte off the stack into A with PLA (PuLL Accumulator). The byte pulled off will be the one pushed on the stack most recently, namely the lo-byte of $1008, which is $08. Then the subroutine does some other things, perhaps involving the value pulled from the stack in order to make sure the subroutine was called from the right location. Eventually, just before executing an RTS, the subroutine pulls another byte from the stack. Now it has wiped out its return address. Instead, the RTS will take the next two bytes on the stack, which are the values $FF and $4F respectively, as the address $4FFF. This is automatically incremented to yield $5000. The program will jump to this location instead of $1009 as we might think. The code at $5000 can do something with the accumulator right away if it wants to ensure that this abnormal path was followed to reach it.

This stack technique can be used over and over to make tracing more difficult. As long as you keep careful track of the contents of the stack, however, you should be able to follow the program flow through any number of RTS's.

My final example ties together several different concepts from earlier chapters. It combines several different forms of protection into one short routine. This multiplies the difficulty and enhances the benefit of tracing it. Since it deals with the disk drive, it offers a good chance to become familiar with the KERNAL communications routines. You'll also find an undocumented opcode, a bit of decryption and a little stack manipulation. Eventually there will be a custom DOS routine involved too.

The routine will not actually run correctly since it depends on some protection on the disk, which we don't want to put there. That's OK; it's similar to a situation you might find yourself in while tracing a program. By the way, it is designed to work with a 1541 drive only. Since it communicates over the serial bus, it may make other serial peripherals like printers do funny things. The program is called 'TRACE ME' on the program disk. If you load it in and disassemble it, here is what you'll see (without the comments of course):

```
1000 A0 1F     LDY #$1F        Decrypt data table
1002 AF        ???             Undocumented opcode
1003 80        ???
1004 10 59     BPL $105F
1006 80        ???
1007 10 99     BPL $0FA2
1009 80        ???
100A 10 88     BPL $0F94
100C D0 F4     BNE $1002
100E A9 0F     LDA #$0F
1010 A2 08     LDX #$08
1012 A8        TAY
1013 20 BA FF  JSR $FFBA       SETLFS 15,8,15
1016 A9 02     LDA #$02
1018 A2 81     LDX #$81
101A A0 10     LDY #$10
101C 20 BD FF  JSR $FFBD       SETNAM at $1081-82
101F 20 C0 FF  JSR $FFC0       OPEN file #15
1022 A9 02     LDA #$02
1024 A2 08     LDX #$08
1026 A8        TAY
1027 20 BA FF  JSR $FFBA       SETLFS 2,8,2
102A A9 01     LDA #$01
102C A2 83     LDX #$83
102E A0 10     LDY #$10
1030 20 BD FF  JSR $FFBD       SETNAM at $1083
1033 20 C0 FF  JSR $FFC0       OPEN file #2
1036 A2 0F     LDX #$0F
1038 20 C9 FF  JSR $FFC9       CHKOUT select #15
```

```
103B A0 0B     LDY #$0B
103D B9 83 10  LDA $1083,Y    Message at $1083-8D
1040 20 D2 FF  JSR $FFD2      CHROUT (PRINT#15)
1043 88        DEY
1044 D0 F7     BNE $103D
1046 20 CC FF  JSR $FFCC      CLRCHN deselect 15
1049 A2 0F     LDX #$0F
104B 20 C9 FF  JSR $FFC9      CHKOUT select #15
104E A0 06     LDY #$06
1050 B9 8E 10  LDA $108E,Y
1053 20 D2 FF  JSR $FFD2      CHROUT (PRINT#15)
1056 88        DEY
1057 D0 F7     BNE $1050
1059 20 CC FF  JSR $FFCC      CLRCHN deselect #15
105C A2 0F     LDX #$0F
105E 20 C6 FF  JSR $FFC6      CHKIN select #15
1061 20 CF FF  JSR $FFCF      CHRIN (GET#15)
1064 48        PHA            Push byte on stack
1065 20 CF FF  JSR $FFCF      CHRIN (GET#15)
1068 48        PHA            Push byte on stack
1069 A9 02     LDA #$02
106B 20 C3 FF  JSR $FFC3      CLOSE 2
106E A9 0F     LDA #$0F
1070 20 C3 FF  JSR $FFC3      CLOSE 15
1073 4C CC FF  JMP $FFCC      JUMP to CLRCHN

DATA TABLE FOR DISK ROUTINES
---------------------------------------
1080 A5 EC 95 86 95 85 90 96 .UFUEPV
1088 85 95 85 97 E0 88 E7 A7 EUEW.H.
1090 A5 B1 F7 88 E8 08 E1 83 .H.H.C
1098 9A C0 3B 97 21 7D 17 63 Z;W!.W.
```

This piece of code might be part of an autoboot or it might be more deeply buried. Let us suppose you have done a little bit of detective work to get to this point. All of a sudden, you run into the code at $1000 above. A few lines of ??? tends to shake your confidence in your work so far. You may have slipped up earlier and gotten off the track, in which case you could be wandering around in irrelevant code till you drop. This is a good reason to keep track of how you arrived at a particular point - you may want to backtrack and double check your work. On second glance, though, things look better. Most of the code seems to be there, and all those KERNAL calls have got to be important (remember the essentials of a protection scheme from PROGRAM PROTECTION VOL. I). You might be tempted at first to ignore the 'bad' code and go on to the code you can read. This will prove interesting but ultimately you will have to come back to $1000 to really figure out what is going on.

Line $1000 looks to be good itself. The problem lies in the opcode $AF at $1002. Since it is listed as ???, it is not a documented 6510 opcode. Luckily, you have this book with its chapter on undocumented opcodes. Turning there you find that

$AF is a simple LDAX Absolute (from a general memory location). This loads both the A and X registers simultaneously from the same absolute memory address. Absolute addressing implies two operand bytes, so the next instruction starts at $1005. If you try disasembling $1005, lo and behold all the rest of the ???'s have dissolved into normal code. Here is a composite disasembly of this first section of code:

```
1000 A0 1F    LDY #$1F
1002 AF 80 10 LDAX $1080     UNDOC. OPCODE
1005 59 80 10 EOR $1080,Y
1008 99 80 10 STA $1080,Y
100B 88       DEY
100C D0 F4    BNE $1002
```

Now is a good time to review the EOR instruction from the encryption and decryption chapters, since that is the purpose of it here. This section of code obviously decrypts something at $1080-9F. We could stop and do the decrypting ourselves, but this routine is simple enough to modify so that the computer does the dirty work for us. Put a BRK ($00) at $100E with your monitor and then execute the routine with G 1000. When you return to the monitor, you'll have the decrypted version (don't forget to take the BRK back out). Still, it wouldn't hurt to trace a couple of times through the loop for practice.

If you want to leave the code in its decrypted form, this can be easily done. The value used in the EOR operation is located at $1080. If you change that to $00, it will EOR the bytes with $00 whenever you execute it. Recall from the EOR discussion that EOR'ing with a $00 value doesn't alter the other input, in this case the already decrypted bytes. In fact, you can replace the undocumented opcode too. All it does is load the A and X registers. Looking down a few lines in the code, X is directly set to $08 at $1010 with LDX #$08. This means that any value put there by the undocumented code will just get wiped out anyway. Therefore, the only real purpose of it is to load the accumulator. So we can replace the $AF opcode with an $AD which gives us the regular instruction LDA $1080. This will make the listing easier to read since the ???'s will be gone.

Now for the heart of the routine. You should be somewhat familiar with the KERNAL SETLFS (Set Logical File Specifications, $FFBA) and SETNAM (Set Name, $FFBD) routines. They are often seen just before a call to LOAD ($FFD5), naturally enough. Here they are used a little differently. The OPEN ($FFC0) KERNAL routine is just like the BASIC OPEN statement, in this case OPEN 15,8,15,'MESSAGE'. SETLFS and SETNAM simply set up the 15,8,15 and 'MESSAGE' part, while OPEN does the actual opening. Note that the 'MESSAGE' part is in memory at a location pointed to by the X and Y registers (lo- and hi-bytes, respectively) at line $1018. The accumulator holds the length of the message, which is NOT a file name in

this case. It is simply a command which will be sent out to the drive when the OPEN is executed.

What is that message? Well, if you've decrypted the code like you should, you know the answer to that question. For those cases that are a little harder to get the decrypted version out in the open, I'll pass along another tip. Notice the device number ($08) in the call to SETLFS ($FFBA). The communications routines are very general purpose routines capable of communicating to many drastically different devices in the same way. In this case, if we merely change the device number to #$03, the 'MESSAGE' will be printed to the screen instead of the drive! Sometimes this goes past too fast, so if you change the device to #$04 instead, you'll get the message on the printer! Pretty slick, huh? Of course, some of the messages sent may not be in ASCII, in which case the printer or screen may do some strange things.

Just as in BASIC, when you OPEN a file it stays open until you close it. However, only one file can actually be sending or receiving information at a time. This is just like the situation in BASIC with the CMD statement, if that helps you any. When we follow the first OPEN process with another one as we do in this routine, the second file takes precedence and the first one is 'de-selected'. The second OPEN statement corresponds to OPEN 2,8,2,'MESSAGE'. Again I leave it to you to find out what this message is.

Following this second OPEN, we encounter an important KERNAL call at $1038, to CHKOUT (CHannel OUT, $FFC9). This routine re-selects a previously OPEN'ed file, for output. In this case, it is file 15. Now we can send information to the drive over this channel. This is accomplished with the loop at $103B-45. This loop retrieves a byte from the decrypted data and sends it to the currently selected file (15) using the CHROUT (CHaRacter OUTput, $FFD2) routine. IMPORTANT: the message sent will not be acted upon until you CLOSE the file OR call the CLRCHN (CLeaR CHaNnel, $FFCC) routine. This can cause frustrating problems if you forget it. If you plan on using the file again, use CLRCHN and you won't have to go through the whole OPEN process again. We are going to use file 15 some more, so we CLRCHN at $1046 and the re-select file 15 with CHKOUT ($FFC9) immediately.

Another message is sent via a CHROUT ($FFD2) loop, and then we CLRCHN again. This time, we're going to use our file for input, so we call upon the CHKIN (CHannel IN) statement to select it for INPUT. Finally, we get two bytes from the drive by calling upon CHRIN (CHaRacter INput, $FFCF) twice. This is like a BASIC GET#15 statement. The byte is returned in the accumulator each time, so the $64,000 question is, what does the routine do with the value in the accumulator? Answer: it pushes it onto the stack where it will stay safely parked during the next operations.

The routine ends by calling CLOSE ($FFC3) to CLOSE files 2 and 15 for good. Finally, it JUMPS to CLRCHN at $FFCC. Notice the use of JMP rather than JSR. What are we to make of this? Remember that all the KERNAL routines are designed to be used like subroutines, so they end with an RTS instruction eventually. The RTS is supposed to return them to the location past where they were called with JSR. Now since JSR was NOT used, the RTS will instead pull off the first two bytes as in the stack example we saw earlier. Thus the net effect of the whole routine is to do something with the disk drive (we don't really care what) that results in it sending us two bytes, which form the address to return to. This may be the main entrance point of the program, or another protection check routine.

It looks like we can bypass this routine entirely and go straight to the next one from now on - if we can figure out how to get the information off the stack to decide what the next address is. That I leave to you to figure out. In a real program you should be sure and check that something set up in this code isn't checked later on, such as the state of the disk drive. The program actually is designed to work with a custom DOS routine (such as in the CUSTOM DOS chapter), which could do just about anything. Basically, all we care about normally is the computer side of things.

As I said at the start of this chapter, the basic idea in program protection is to link the functioning of the program to some physical object, in this case the original disk. Usually this link is very simple; it consists of a 'key' value or values returned from the disk drive. It really doesn't matter what type of protection is on the original disk or how the drive checks for it. What is important is the key value, which will be checked or used by the program. Once we find the key, we can unlock the program.

This concludes our introduction to tracing programs. Tracing is a time consuming way to break programs, but it is useful as a last resort or when you want to find out just what a program does to protect itself. Experience is important when you are tracing programs, and the only way to gain experience is through practice. Think of the time spent as an investment in the future.

In this chapter we 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 for the legitimate user to break your software in order to obtain an archival copy. This may very well be the best deterrent to software piracy. Second, take the time to evaluate the protection schemes used by other programmers. Many times an important concept may be gained by examining other protection schemes. Third, evaluate just how much protection you feel your product requires. If your are writing a specialized program that requires sophisticated documentation very little protection may be required. On the other hand, if your program is the hottest new game or utility it may be of upmost importance to spend some additional time and money to properly protect your investment.

We have heard of more than one programmer who, through their own careless actions, have let unprotected and un-copyrighted versions of their programs slip from their possession. Once this version of the program hits the user groups it somehow becomes 'public domain'. This relates back to the earlier chapters on trade secrets. If you feel your program is worth writing, it should be worth protecting.

How much protection is enough?? How much is too little?? These are tough questions to answer. If you spend an adequate amount of time protecting your product, it may take several weeks or even months. The time that you spend protecting your software may be of no value if the product does not sell or if someone else beats you to the market place with a similar product. Another very important concept to consider is: If you design a really great protection scheme what happens if it can not be duplicated by the mass duplication machines? Most commercial disk duplication equipment is locked into a few standard protection schemes. So, before you spend weeks or months trying to integrate a protection scheme into your program be sure that it can be mass produced. Very much of today's disk duplication equipment is too limited in its ability to duplicate heavily protected software. We have been quoted a price of over $4,500.00 just to produce a single master disk from a commercial disk duplicator. Mind you, this was just for the first disk!!! Each additional duplicated disk would be an extra cost. This was in addition to the cost and time that we had spent in developing the scheme originally.

Each software author must make a tough decision on which method is the best for himself. Do you decide on an easy to duplicate scheme (which is also easy for the pirate to copy)?? Do you spend $5000.00 or more on developing the scheme yourself?? Do you pay someone to develop your protection for you??? O.K. enough questions, let's have some answers.

If you choose an easy-to-duplicate and easy-to-produce protection scheme, you can be sure that it may be easily copied by anyone that has a nybble copy program. If your program is not going to be a very big seller, this may be the way to go. Don't invest much money in protection and don't risk much money in getting it to market. Keep in mind that if the program does become a best seller you may lose many sales to the software pirates. If the program does not sell the pirates aren't going to want it either.

If you choose the second route and adequately protect your own software, be sure to evaluate the time that you are going to spend in protecting it. If it takes you a month to write, debug and implement your protection scheme, you may be wasting valuable time and money. The time that it takes you to effectively protect your software may be better spent by hiring someone who is proficient in program protection and spending your time doing what you do best (namely writing software). Remember, it takes much more time to properly integrate a protection scheme into an program than most people ever realize. Be sure to allow yourself adequate time in the development phase of the program to protect your program. Many times the graphics, sound or file handling of the program is turned over to specialists; it may pay to do the same with the protection scheme. If you choose to do your own program protection be sure to plan for the protection just as you plan for the main program. Don't make your protection scheme an after thought. Many programmers spend months writing a super program, only to spend days on the protection scheme. Don't be caught short, plan your protection strategy from the beginning.

Now that you have spent a month or more protecting your program be sure to re-evaluate your protection scheme. What may have sounded like a good idea just a month ago, may now be obsolete due to a new copy program. If you are going to protect your software do a good job (any job worth doing....). Take the time to learn the tricks and take more time to stay current. Program protection is an ever-changing field. What was STATE OF THE ART just a few months ago is now obsolete.

Keep in mind that any scheme may be defeated. Sooner or later someone will break it or a copy program will be written that will copy your protection scheme. Remember: THERE IS A LARGE GROUP OF PEOPLE THAT CONSIDER PROGRAM PROTECTION TO BE THE ULTIMATE ADVENTURE GAME. These are the folks who don't buy your software for the program, they buy it because of the protection scheme. These are the professionals at program protection. They break the protection scheme for the fun of it. It can be quite

a thrill when one first breaks an extremely difficult program protection scheme. For many people it is more satisfying to break the protection scheme than it is to play the game!! These are the folks that can make your job as a program protector frustrating. They may be more knowledgeable than you in the area of program protection. So after you spend a month or so in protecting your program and you come up with the ultimate scheme don't be surprised if it gets broken very soon. Many program protection schemes have been adequately thought-out in most respects, but not all! Virtually every scheme leaves a big gaping hole in its protection. Many schemes are terrific in all aspects but one. It is this one hole that someone interested in breaking the program will find. So much for your ultimate protection scheme.

Too many programmers rely on the disk duplication company to provide the protection scheme. Some disk duplicators are so eager to get your duplication business that they will 'give' you a protection scheme. You may be sure that the scheme that you receive will be very similar to other schemes produced by the same duplicator. Most companies have their own form of protection to offer the software author. These schemes are often based upon the capabilities of the duplication equipment rather than on the needs of the software author. Be advised that if you rely on the disk duplicator to provide you with the protection scheme you may not be getting your money's worth (even if they give you the scheme for free). Most disk duplicators are specialists at their business - disk duplication. They may not be specialists in program program protection (most are not).

The third choice for the software author is to rely totally on some other party for the program's protection. This would be someone who is proficient in program protection. Program protection is their business, that is their specialty. This may be a reasonable alternative for the software author. The software author will do what he does best (write programs) and the program protector will do what he does best. This sounds like a match made in heaven doesn't it?? Well, it may not be. As with any relationship between two parties there may be problems. First, there is the problem of trust. Can you, as a software author, trust your program to a stranger for its protection? Second, will the program protection conflict with the actual program? This is extremely important and sometimes difficult to verify. There may be some small interaction between the program and its protection scheme that may not be obvious at first. This may cause all sorts of problems a later date. Third, is the person that is going to protect your program capable of doing the job properly? Be sure that the person that you select does the job and does it right. Get a protection scheme that is unique and state of the art, not something that has been used many times before and is already obsolete. Ask for a sample protected disk and examine it yourself, if you feel qualified, or else have a knowledgeable friend look at it. Take it to the real experts: buy several

good copy programs and try them out on the sample. The money spent may well turn out to be a wise investment (this advice applies to your own protection schemes as well). One more thing: as in all business deals, be sure that you can trust the party that you are going to be dealing with.

Finally there is the concept of how much is enough. Remember to evaluate your program and its protection needs. Then make a sound and practical decision as to how much protection is actually required, both in time and money invested in the protection scheme. Once you have determined your needs, proceed with those thoughts in mind during the development of the program. Build the program's protection into the program as you go, don't add it on later. If your program is not going to be very popular and it only appeals to a small group of people you may not want to protect it at all. Most people are honest and will be more than happy to pay for a legitimate copy of a program that they really want. Many people that we know will buy the original program even if that person obtains a broken copy elsewhere. They buy the original for the documentation and they buy the original for the personal satisfaction of knowing that they are doing what is right. They may have obtained a broken copy (illegally) just to see if they like the program or not. Then, if they like the program they will gladly pay for an original. Most broken software that is passed around (illegally, of course) would not have been bought by the person receiving it if that was the only way to obtain a copy. They may get a copy of your program on a disk with twenty other programs, none of which they really have any use for. It seems that some people are just collectors, they just want a copy of everything. These are the folks who wouldn't know how to use your software even if they had purchased it with all the documentation!!! Some of the illegal copies being made today are taking away from the sales that a software author should have, but most are not!! So spend some time and evaluate your needs and your budget. Don't spend every cent that you can on protection, especially if the protection scheme detracts from the useablitiy of the program.

Now to the nitty-gritty of protecting your own program. If you decide to protect you program use every technique that you can. Just don't overlook the most effective tool available to you as a program protector: PSYCHOLOGY. Use the psychological aspect of program protection to your advantage. Many programs may be easily broken if it were not for a few well placed bits of code. This code is truly bogus; it does not function at all. It just looks like it should do something, possibly check the disk for an error, but the program never uses this code. Its whole purpose is pose a psychological barrier to the pirate. The code is left where it may be easily found by anyone examining the program. The experienced pirate will immediately try to defeat your code and then try the broken copy. It won't run, because the protection scheme that the pirate found and disabled is not even used by the program. Another psychological tool that may be used is simply letting the pirate know that what he is doing

is not right. Appeal to the nobler aspects of the pirate - it may work! (then again it may not). DON'T, under any circumstances, DON'T put in any snide comments or wisecracks aimed at the pirate. They don't work and they only serve to encourage the pirate. Personal insults have not served any useful purpose during all of civilized history, so don't think that they will do any good in your program. They won't.

If you are going to protect the program yourself, you need to start somewhere. First decide on a method of protection; half tracks, nybble counting etc. Then be sure that the method selected may be mass duplicated. Base your whole program around the thought of implementing the protection scheme. Once you have made your decision use every tool available that may hinder or prevent the pirate from getting your software. Use some of the older and possibly forgotten techniques to protect your software. Many times the pirate is expecting a more sophisticated program protection scheme that actually exists and he will overlook the obvious.

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 unlistable or enter an endless loop (see the PROGRAM PROTECTION MANUAL VOLUME I). Change the name of the disk, the I.D. and the 2A to unprintable 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 check the protection, 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 check the protection again, store a few more values in memory and load the main program. The main program will contain a number of places that 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 changed 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 almost any pirate will go crazy trying to find them all. Buy a good BASIC compiler and have some of your code in compiled BASIC. It is not usually worth the time to decipher the compiled code.

Use a protection method on the disk that will not cause the drive to beat the R/W head against the end stop. DON'T USE BAD BLOCKS. We have given you many different routines in this manual that will allow you read exotic protection schemes. Use one or more of them. The protection methods presented herein

are viable and effective. Just be sure that your disk duplicator can handle the scheme that you choose.

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 have to 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.

Try to find a disk duplicator who has a programmable disk drive. Save the information to the disk in such a manner that it cannot be copied by the 1541 disk drive. The 1541 disk drive has a few hardware limitations, such as a wide R/W head that will not allow adjacent tracks and half tracks to be properly written. Density changes may be a subtle way of protecting a disk. Nybble counting can be very effective especially if the speed of the drive writing the data is slowed down slightly. Synchronized tracks may easily be done by virtualy any commercial duplicator (NOTE: they also appear to easy to reproduce on the 1541 even though no copy program is currently doing so). Use any of the methods mentioned in this book or better yet use ALL of the methods mentioned here. Use some of the tips contained in the PROGRAM PROTECTION MANUAL VOLUME I to protect your software. Use some ideas 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 (don't pirate someone else's routine). Finally change the 'A' to an 'E' in the BAM (track 18, sector 0). This can be the good deterant to the novice pirate.

One of the biggest problem that software authors have today is called 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, then 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.

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.

READ ONLY MEMORY

The following information is presented for those who wish to learn more about the design, manufacture, and theory of MEMORY CHIPS. This material is presented as general knowledge. Programming EPROMS does not require that you know the information presented here. This information is presented for the advanced EPROM progrmmer or those who wish to expand their knowledge of memory chips.

All you will need to know about programming EPROMS is presented in the chapter on PROGRAMMING EPROMS and in the PROMENADE manual. You will also find some material in the latter section of the chapter on ADVANCED EPROM PROGRAMMING that may prove useful as a supplement to the PROMENADE manual.

Don't let these chapters scare you off!! EPROM programming is fun and easy. You don't even need to read this chapter to program EPROMS successfully. We are presenting this material for those who wish to learn more about the evolution and design of MEMORY CHIPS.

READ ONLY MEMORY

A READ ONLY MEMORY (ROM) is an LSI (large-scale integration) circuit consisting of an array of semiconductor devices such as diodes, bipolar transistors, or FETs which are connected to perform a particular set of switching functions. ROMs are available in five basic types: mask programmed ROMs; programmable ROMs, or PROMs; and erasable programmable ROMs, EPROMs and EEPROMs; electrically alterable ROMs, or EAROMs; and nonvolatile random-access memory, or NVRAM.

A ROM is a form of memory which contains a fixed set of data that can be read in a similar way as RAM. ROM is primarily used for storing information which isn't subject to change. This type of memory contains no mechanism to enable the user to alter the data stored at a particular address.

Unlike RAM, the removal of power from the ROM does not alter its contents. It was for this reason ROM became very popular with the introduction of dedicated microprocessor systems. The system's program was stored in ROM, and the data was stored in RAM.

The first ROMs contained an array of cells in which a series of 1's and 0's was created using a metallized mask. Mask programmed ROMs are permanently programmed at the time of manufacture by adding or leaving out diodes or transistors. The user had to supply the manufacturer with a truth table on punched cards or tape, and a computer generated a mask for the ROM which would give the desired truth table.

Some of the more typical uses for ROMs are: as a code converter, like in printer interfaces; as a look up table; as a character generator; as a keyboard encoder; or as a logic gate replacement, like the 64's PLA memory management chip.

The major drawback to using ROMs are the set-up charges. The tooling costs for ROMs is quite high unless users plan on using large volumes of the same ROM. It can cost several thousand dollars to develop a mask for a single ROM.

To counteract the high set-up costs of ROMs, manufacturers developed PROMs, or user-programmable ROMS. PROMs come with a diode in every position. Each diode is a fusible link which can be ''blown'' with a PROM burner leaving a 1 in the corresponding bit position. This type of PROM was preferred over nichrome-fuse PROMs which tended to regrow, changing the programming with age. PROM programming is permanent, reading just like a ROM once programmed. Programming errors could not be corrected, so mis-burned chips had to be discarded. One advantage to PROMs was that they provided a low-cost way to develop computer firmware for low volume productions.

The increased need for low-cost, user-programmable ROMs led to the development of an erasable, MOS-technology PROM, or EPROM. This type of ROM uses charge-storage programming to store the truth table of 0's and 1's.

This chip was packaged in a ceramic DIP package with a

quartz window used for exposing the chip to high intensity, short-wave ultraviolet light. The high energy UV photons collided with the EPROM's electrons causing the stored charges to leak off, erasing the EPROM.

The EPROM wasn't intended for read/write operations, but it became very useful for prototyping and other applications where data needed to be altered several times. Initially, EPROMs were popular in development labs, but as production costs decreased, EPROMs began to be used in medium and low volume production applications. The major disadvantages to EPROMs is the need for an external programmer, and having to erase and reprogram the entire chip to alter a single byte.

The latest advancement in the ROM field is the user-programmable, electrically erasable ROM, or EEPROM. The major advantage to this type of device is the ability to alter the non-volatile memory while it is still in the host circuit. Removal and exposure to ultraviolet light is no longer necessary. Some types of EEPROMS can even be programmed using the standard TTL (+5 volts) voltage levels. This type of ROM even allows the erasure of single bytes without the need to erase the entire chip.

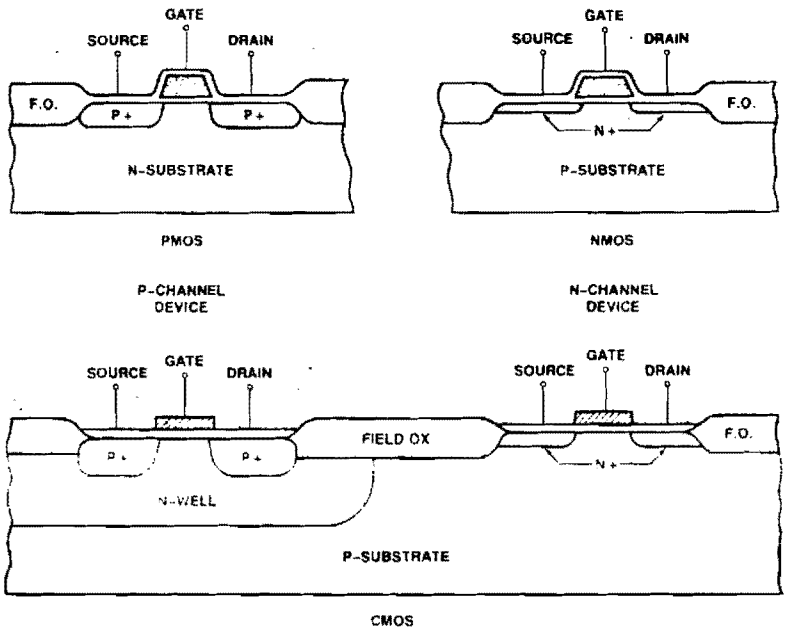
EEPROMs are opening up new applications in many areas. EEPROMs are used in digital instruments for calibration and diagnostics. If the instrument drifts out of calibration, the contents of the EEPROM can be changed to compensate for the drift. EEPROMs are also used as non-volatile look-up tables in remote scanning terminals, programmable controllers and data loggers.

The electrically alterable ROM, or EAROM, has several features of the EEPROMs. Individual addresses within the EAROMs can be reprogrammed electrically, as with the EEPROMs. EAROMs are quite slow, up to 600 ns access time, and the fabrication process is complicated and expensive. As a result, the cost of EAROMs has remained high, and they are only available in low densities.

Another type of nonvolatile memory, the NVRAM, is sometimes confused with EEPROMs. The NVRAM contains RAM, with a duplicate bank of EEPROM. The memory operates as conventional RAM until the power is removed. On power-down, the chip uses a power-down routine to transfer the contents of RAM to EEPROM. The process is reversed when power is returned. It takes from 1 to 5 microseconds to transfer EEPROM to RAM, and about 20 milliseconds to store data into the EEPROM. The major disadvantage to NVRAMs is the requirement of nine transistors to store a single bit, versus two for EEPROMs. This is one reason that NVRAMs are the least dense of all memory technologies. By the end of 1985, EEPROMs are likely to be available in densities up to 256k, whereas NVRAMs probably won't exceed densities of 16k.

There are three families of MOS technologies - CMOS, PMOS and NMOS. The family types refer to the type of MOS transistors which are used in the circuit. Figure 1 shows the structure of the three MOS families.

FIGURE 1



PMOS ICs use p-channel transistors which are created by diffusing Boron into an n-type silicon substrate to form the source and the drain of the transistor. They are also called p-channel because the channel is composed of positively charged carriers.

NMOS ICs are similar to PMOS, but they use phosphorus or arsenic to make n-channel transistors in p-type substrates. N-channel ICs have channels which are composed of negatively charged carriers.

CMOS or Complementary MOS ICs utilize both p-channel and n-channel devices on a common substrate. Either p- or n-type substrates can be used, but areas of the opposite substrate (wells) must be present to create the complementary transistor type.

Most early memory devices were made with PMOS circuitry. The demand for higher speeds and greater densities lead to the development of NMOS ICs. NMOS ICs are inherently faster due to the greater speed of the n-channel electrons. The majority of memory devices in production are fabricated with some type of NMOS technology. The use of CMOS technology in memory has become widespread because of the very low power

consumption, noise immunity, and temperature insensitivity. However, CMOS's slow speed, and the high cost to produce two transistor types on a single substrate, has limited their use to those where the advantages justify the cost.

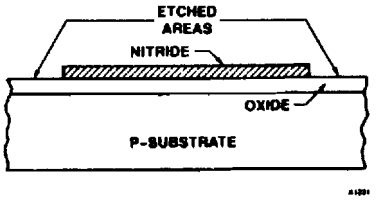
FABRICATION

The fabrication process for most types of memory technologies is quite similar. The next section describes the fabrication process of the new n-channel HMOS circuit. This type of cell is primarily used in 5 volt memory devices.

The MOS IC fabrication process begins with a 100 millimeter diameter, half millimeter thick slice of a single silicon crystal. The silicon slice is then oxidized in a furnace at a temperature of 1000 degrees Celsius to develop a layer of silicon dioxide on the surface. The silicon dioxide is used as an insulator between layers, and it resists diffusion (furnace operations). However, there are certain areas in which diffusion is desired, so the silicon dioxide must be removed with hydroflouric acid. Silicon nitride is used as a photoresist because it resists etching by hydroflouric acid. A photographic mask is placed over the silicon dioxide, and silicon nitride is deposited on the exposed surface in a gas-phase chemical reactor. When the photoresist (silicon nitride) is exposed to ultraviolet light, it polymerizes. Subsequent washings in the appropriate solvents removes the polymerized areas. If the crystal is now exposed to hydroflouric acid, the silicon dioxide will remain in the areas protected by the photoresist, and will be removed in unprotected areas. Since this is a photographic process, the mask openings can be very small and located very accurately so that many components can be placed on a single chip. Usually masks are produced from large scale drawings, which are photographically reduced.

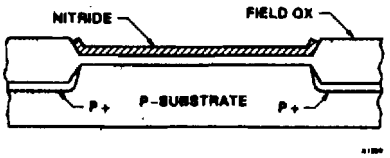
The first pattern defines the regions where the transistors, capacitors, diffused resistors, and first level interconnects will be made. Figure 2 shows a wafer after completing the first etch.

FIGURE 2



Next, the wafer is implanted with accelerated boron atoms. The boron will only reach the silicon substrate where the nitride resist and oxides have been etched away, exposing p-type areas which will electrically separate active areas. After implanting, the wafers are oxidized again. The oxide only grows in etched areas because silicon nitride acts an oxide barrier. When the oxide is grown, some of the substrate is dissolved creating a physical as well as electrical isolation from adjacent devices. The remaining silicon nitrides are removed, and the wafer is etched again to remove the second layer of oxides, leaving a field oxide. Figure 3 shows the result of the nitride removal and the second etch.

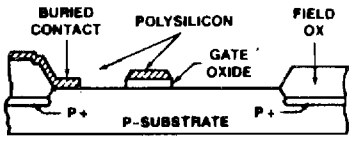
FIGURE 3



Once the active regions have been defined, the transistor types need to be determined. Another pattern is laid over the wafer to define any special characteristics, and the wafer is implanted with dopant atoms. The dose and energy at which the atoms are implanted determines the characteristics of the transistor. The type of dopant determines depletion mode (n-type) or enhancement mode (p-type) operation.

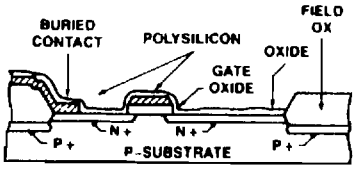
After the transistors are defined, the gate oxides are deposited. Care must be taken to avoid contamination or defects in the oxide to ensure uniform thickness of the oxide. If the thickness of the oxide varies, the characteristics of the component won't be uniform. The gate oxide is then masked and holes are etched to allow for direct gate contacts. Next, the wafers are covered with a poly crystaline silicon gate material in a gas phase chemical reactor. The gate material is then doped with phosphorus to reduce the resistance to 10-20 ohms. The gate layer is then masked to define the transistor gates and interconnects. Figure 4 shows the wafer after the gate material has been deposited.

FIGURE 4



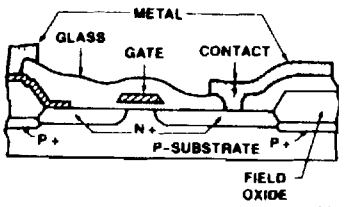
The wafer is then coated with arsenic or phosphorus to form the drain and source junctions. Then the wafer is oxidized with a layer of silicon dioxide to seal the junctions from contamination. Figure 5 shows the wafer after being sealed with silicon dioxide.

FIGURE 5



To minimize capacitance and provide insulation between the layers and metal interconnects, the wafer is coated with a layer of glass. The glass is patterned with contact holes and the wafer is placed in a furnace to smooth the surface. The metal interconnects are usually aluminum or aluminum/silicon. The metal is deposited on the wafer defining the interconnect patterns and bonding pads. The wafers are then covered with a low temperature alloy to insure good contact between the aluminum and polysilicon. Figure 6 shows the wafer after the circuit completed.

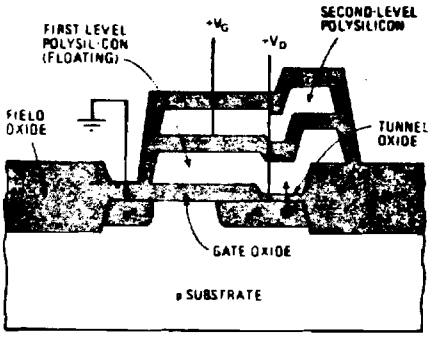
FIGURE 6



The circuit is now complete, but the metal is very soft and the circuit is susceptible to contamination by moisture. To protect the circuit, the wafer is coated with silicon nitride or a silicon and phosphorus oxide composite. The final step in the fabrication process is patterning for the bond pads were the external connections will be made.

EPROMs and EEPROMs are fabricated with the same general process, but EEPROMs have a double poly structure instead of a single poly structure. The double poly structure is patterned after the second nitride layer is deposited. The double poly structure requires the formation of a capacitive floating gate node which stores the cell charge. Figure 7 is an example of a double poly structure device.

FIGURE 7



When fabrication is completed, the wafers are tested. Each circuit is individually tested under operating extremes to determine which circuits will operate reliably in normal use. After testing, the wafer is cut into the individual circuits. Circuits which pass testing are sent to packaging, and the faulty circuits are discarded.

There are two types of packaging: hermetic and

non-hermetic. Hermetic packages consist of two ceramic halves sealed with glass, or ceramic packs with metal lids. Non-hermetic packs are usually molded plastic.

The ceramic package consists of two parts, the base and the metal lid. The base contains the leads and the cavity in which the finished circuit will be placed. The base is placed on a heater and the circuit is bonded to the base with a low temperature alloy. Next, wires are bonded to the circuit and to the leads. Finally, the package is placed in a dry atmosphere and the metal lid is soldered to the base.

In a plastic package, the circuit is bonded to a pad on the lead frame and is connected to the leads with gold wires. The frame then goes to a mold injection machine and the package is formed around the lead frame. After packaging, the circuits are retested and sorted according to speed and power consumption. The finished circuits are labelled with circuit type and date of manufacture. For example:

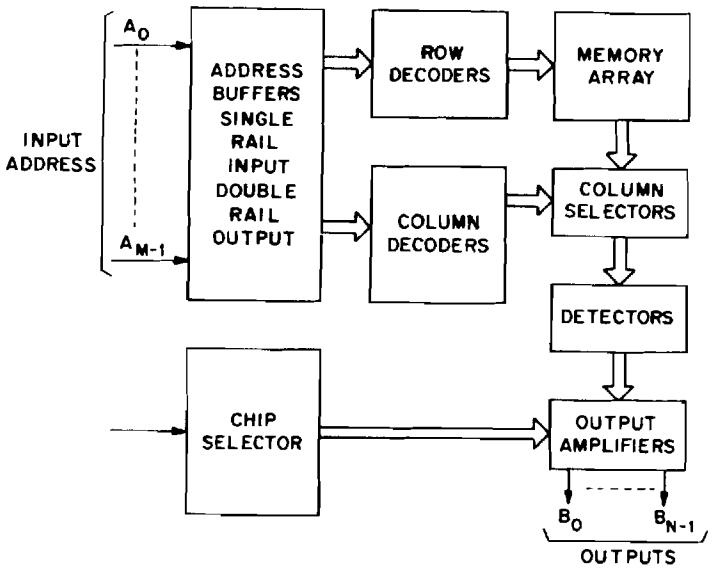
```
D2764D-4  :type and speed
8349EX030 :date
```

The above example would be printed on a 2764 EPROM having an access speed of 400ns which was manufactured the 49th week of 1983.

READ ONLY MEMORY STRUCTURE

A block diagram for a ROM is shown in figure 8. Information is stored in a rectangular array of crosspoint elements which are usually diodes, bipolar transistors, or MOS transistors.

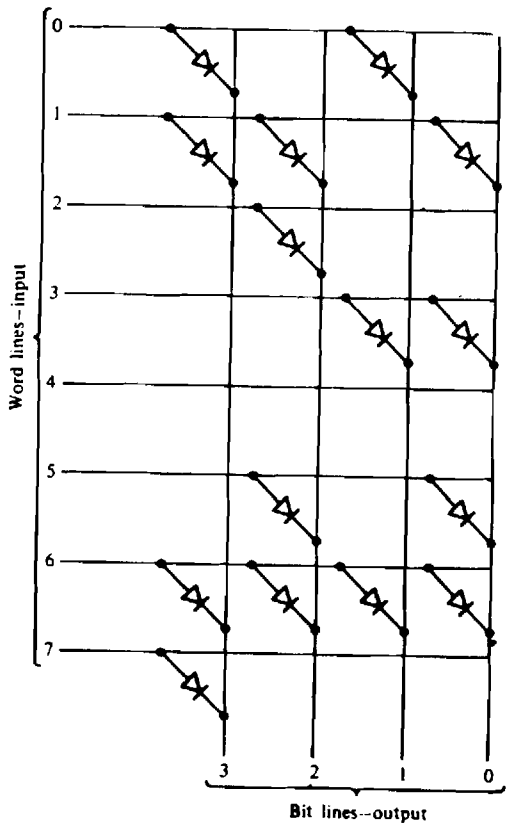
FIGURE 8



The binary address of the desired output word is applied to the address-input buffer which provides an interface between the logic levels of the external system and those of the ROM. It also converts the input from single-rail to dual rail logic. The address signals are transmitted to a word-line decoder which is a series of AND gates connected to select one of the word lines. The addresses are also applied to a bit-line decoder which selects one of the groups of bit lines. The column selectors send the bit line signals to the detectors. There is one detector and one output amplifier for each bit of the output word. The chip-select circuit disables the ROM so that it can be used as part of a memory with a capacity which exceeds that of the individual ROM chips.

Figure 9 shows the internal structure of a diode ROM. ROM consists of rows called word lines and columns which are called bit lines. A ''1'' (logic high) is placed on one, and only one input (word) line at a time. A single input signal produces a word on the output (bit) lines.

FIGURE 9

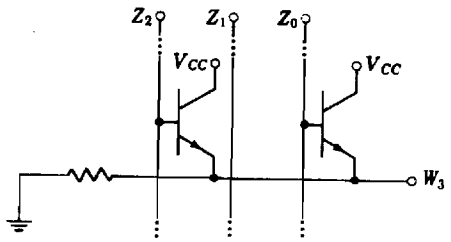


The horizontal and vertical ''wires'' are connected only through diodes. Only one input line is selected (made high) at a time. Suppose input line 5 is high. Output lines 0 and 2 would be made high because of diode coupling. Output lines 1 and 3 will be low since there is no diode. Thus, the output

will be 0101. Likewise, when line 4 is high, the output is 0000; when line 6 is high, the output is 1111; when line 0 is high, the output is 1010. The major disadvantage to diode ROMs is that the output current is drawn directly from the input lines, not from a power supply like bipolar and MOS ROMs.

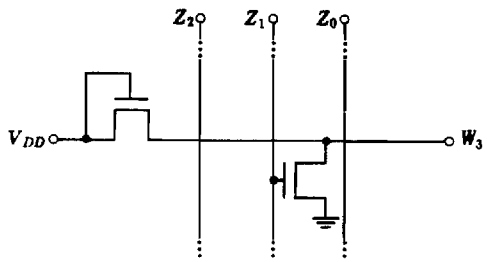
Transistor (bipolar and MOS) ROMs operate in much the same way as diode ROMs. These types or ROMs offer faster access times and lower current draw on the input lines. Bipolar ROMs have access times (time interval between the application of an address at the input and the appearance of data at the output) as low as 15ns and are used in applications where speed is a primary consideration. A bipolar transistor, sometimes using a diode clamp, forms the basic cell. Any cell can be addressed via the x-y decoder. Logic 1 is obtained at the output when the base of the transistor is high. For example, in figure 10, if Z2 was high, a 1 would appear at W3 (the output). Figure 10 shows the structure of a bipolar ROM.

FIGURE 10



The basic organization of a MOS ROM is shown in figure 11.

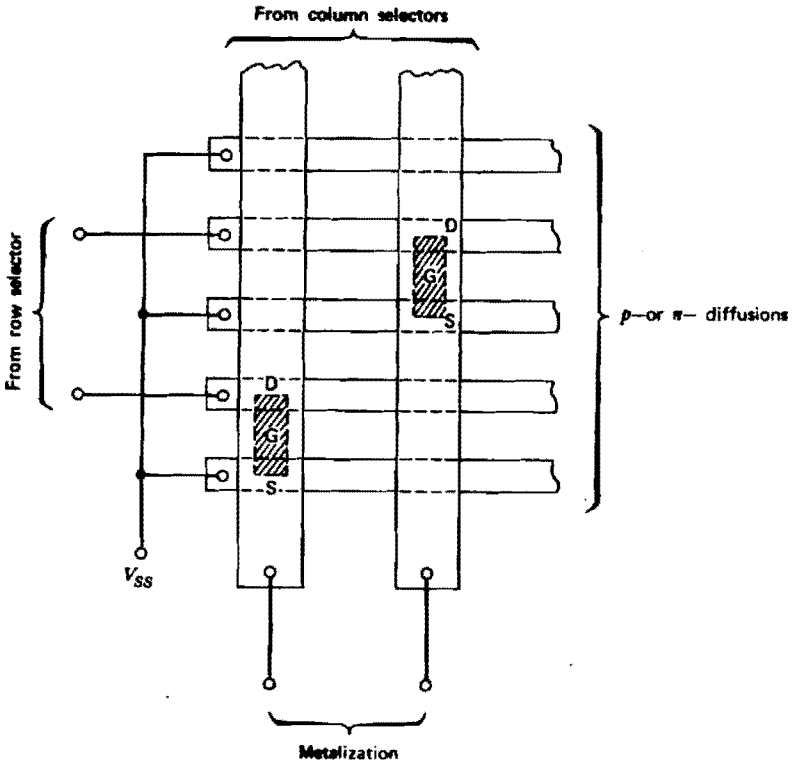
FIGURE 11



A MOS ROM works like a bipolar ROM except the outputs are high, and are brought low when an input is high. In figure 11, w3 is high until z1 goes high. When z1 is high, w3 goes low.

The physical structure of a MOS ROM is shown in figure 12.

FIGURE 12



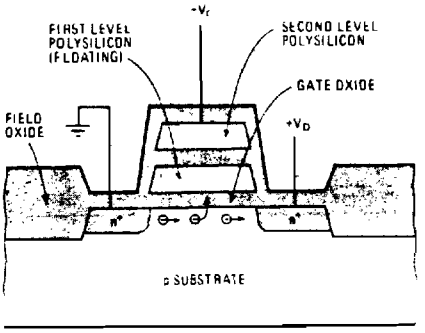
The rows are p- or n- doped silicon which are connected alternately to +5 or the row selector. The metalized columns are connected to a column, or y-selector. A thin oxide is deposited at each intersection forming a transistor. The rows connected to +5 are transistor's source, while the alternate rows connected to the row selector are the drains. MOS ROMs are preferred to bipolar ROMs because larger densities can be made at a lower cost. However, they are much slower, having an access time of 400ns.

In 1971, Intel developed the EPROM. It was created to provide non-volatile memory storage like a PROM, but could be erased with exposure to ultraviolet light. The EPROM's new transistor structure opened a new field to dedicated microprocessor applications.

The EPROM transistor cell takes advantage of the insulating properties of silicon dioxide to store a charge on a gate isolated within silicon dioxide. Figure 13 is a cross

section of a typical EPROM cell.

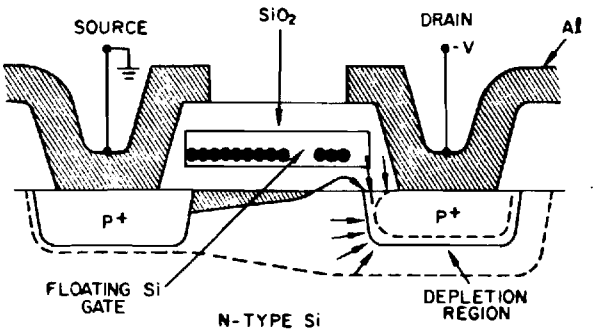
FIGURE 13



When a charge is stored on the ''floating'' gate it changes the characteristics of the transistor. When in the read mode, a transistor with an uncharged floating gate will conduct current from drain to source. In the charged state, the transistor will not conduct current. The charged or uncharged state of the gate gives the logical 1 or 0 output.

The chief benefit of floating gate, or FAMOS, transistor structure is the ability for the charge to be removed or restored at the user's discretion. The floating gate structure consists of a polycrystalline silicon FET embedded in silicon dioxide. Figure 14 is an example of a FAMOS transistor.

FIGURE 14



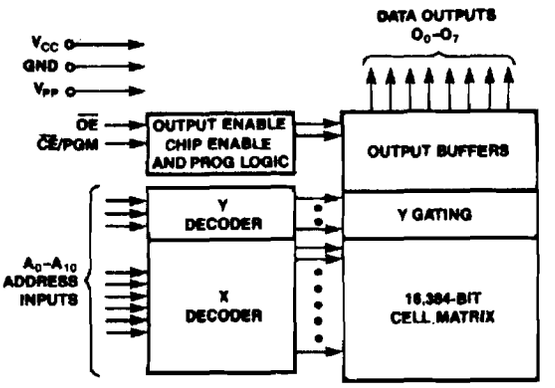
The passing of a very large current from the drain to the source of the FET causes avalanche breakdown to occur in the oxide and a negative charge to accumulate in the gate. If too large a voltage is applied between the drain and source a current will pass through the substrate destroying the cell. When the current is removed, the breakdown of the oxide stops and a negative charge is trapped on the gate. The negative charge on the gate turns the transistor on. An EPROM can be erased by exposing the circuit to high-intensity ultraviolet light. When the cell is exposed to UV light, the breakdown mechanism is set up on the oxide and the trapped charge is released.

The erasure of most EPROMs begins upon exposure to UV light with wavelengths shorter than 4000 Angstroms. Sunlight and flourescent lamps generate light with wavelengths from 3000-4000 Angstroms. Constant exposure to roomlight could erase data in approximately 3 years, while exposure to sunlight could erase data in as little as 1 week. For this reason, opaque covers should be placed over the quartz window to prevent accidental erasure. The recommended erasure procedure is exposure to ultraviolet light which has a wavelength of 2537 Angstroms. The UV dose (intensity x exposure time) should be at least 15W-sec/cm squared. The erasure time with this dosage is approximately 15 to 20 minutes using a UV light with a 12000 micro W/cm squared. For best results, the chip should be placed approximately one inch from the lamp. It is possible to over-erase an EPROM, which destroys the silicon dioxide insulator. Over-erasure can begin to occur if the EPROM is erased in excess of 50 hours.

MODES OF OPERATION

All EPROMs types operate in a similar manner. There are five modes of operation common to all EPROMs. Figure 15 is a block diagram and a table of operating modes for the 2716 EPROM.

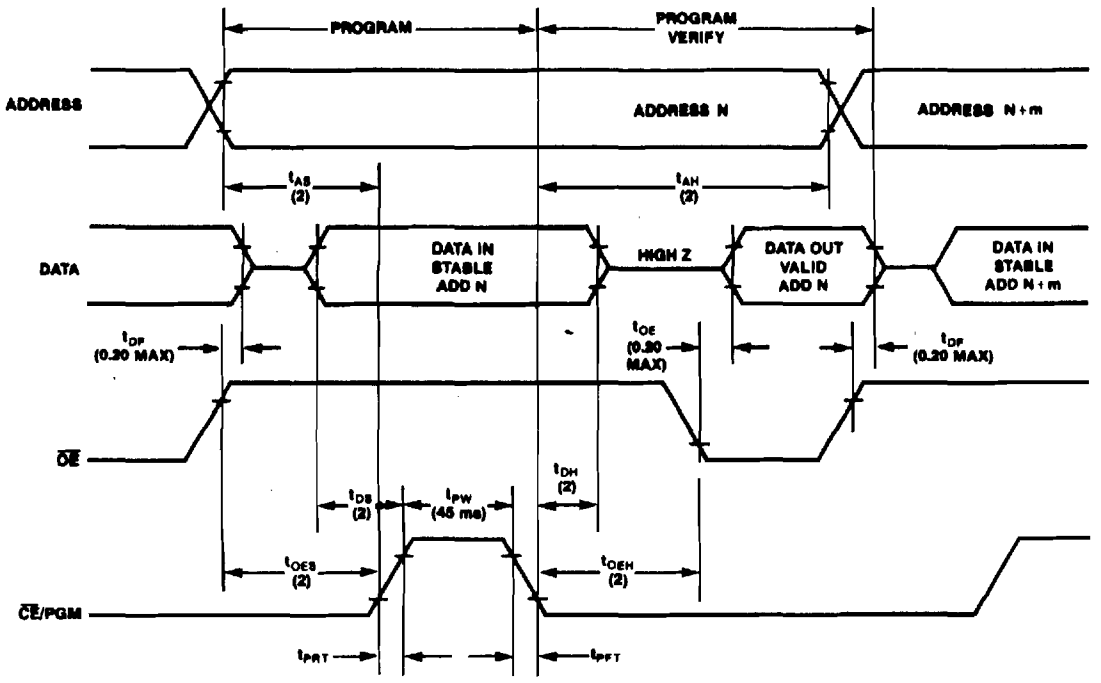
FIGURE 15



Mode \ Pins	$\overline{CE}$/PGM (18)	$\overline{OE}$ (20)	V_{PP} (21)	V_{CC} (24)	Outputs (9–11, 13–17)
Read	V_{IL}	V_{IL}	+5	+5	D_{OUT}
Standby	V_{IH}	Don't Care	+5	+5	High Z
Program	Pulsed V_{IL} to V_{IH}	V_{IH}	+25	+5	D_{IN}
Program Verify	V_{IL}	V_{IL}	+25	+5	D_{OUT}
Program Inhibit	V_{IL}	V_{IH}	+25	+5	High Z

READ: EPROMs have two control lines which must both be low in order to read data at the outputs. The Chip Enable (CE) is the power control which is used to select the EPROM. The Output Enable (OE) is the output control and is used to gate data from the data outputs. If the inputs (addresses) are stable, EPROM access time is equal to the delay from CE to data output (Tce). If the addresses are stable and the CE is low, data is available at the outputs after the falling edge of the OE. Figure 16 is the timing diagram of a typical EPROM.

FIGURE 16



STANDBY: EPROMs have a standby mode used to reduce the current draw from 125 mA to 35 mA. The EPROM is placed in the standby mode when the CE line is high. The condition of the OE line doesn't matter when the CE line is high.

OUTPUT DISABLE: EPROM output is disabled when the OE is high and the CE is low. The chip is still selected, but the output of data is inhibited. In the standby mode, the outputs are in a high impedance state.

PROGRAMMING: After the EPROM has been erased, all the bits are set to 1's ($FF hex). Data is programmed into the chip by programming 0's into the appropriate locations. Individual bits can be only be turned off (1 to 0). Erasure

of the entire chip is necessary to return bits to the high state (1). Data is programmed into the chip when the Vpp (programming voltage) is at 21 volts and the CE is low (see figure 15b). In most cases, exceeding 22 volts on the Vpp pin will permanently damage an EPROM. It is also suggested that a 0.1 microfarad capacitor be connected from Vpp to ground when programming to suppress spurious noise which may alter the programmed data. Mitsubishi chips are especially difficult to program without this capacitor.

After the data and address lines are stable, a 50 millisecond pulse is applied to the CE line. This procedure must be performed at every address which is to be programmed. Any address can be programmed in any order. The programming pulse must not exceed 55 milliseconds or the chip will be ''over-programmed''. It is also possible to program several EPROM's simultaneously as long as proper TTL levels are maintained.

PROGRAM INHIBIT: It is also possible to program multiple EPROMs with different data by controlling the CE line. Programming is inhibited when the CE line is high. Programming can be continued when the CE line is pulsed low and 21 volts is present at the Vpp pin.

PROGRAM VERIFY: A program verify should be performed after each location is programmed. The verify mode is entered when the CE and OE are low, and 21 volts is applied to Vpp. Data will appear at the outputs which can be compared to the desired data. If the data fails, the location can be placed in the program mode again.

EEPROMS

The EEPROM evolved from EPROM technology. The EEPROM cell takes advantage of the FAMOS circuitry of the EPROM with the addition of a tunnel oxide region above the drain of the floating gate transistor. This additional layer of oxide allows an electrical charge to move bidirectionally either onto or off of the floating gate. Figure 17 shows an EEPROM cell in comparison with an EPROM cell.

FIGURE 17

When 21 volts is applied to the transistor's gate and a ground to the drain, a positive charge is passed through the tunnel oxide onto the floating gate. Reversing the voltages removes the charge from the gate.

EEPROMs offer many benefits over the conventional ROM or EPROM. The tunnel oxide mechanism has very low power requirements, so programming can be accomplished with a small power supply. The low current requirements make it possible to program an EEPROM with the existing supply of most computer systems. The read, write, and erasure modes can be accomplished in-circuit with only minor modifications to an existing EPROM system.

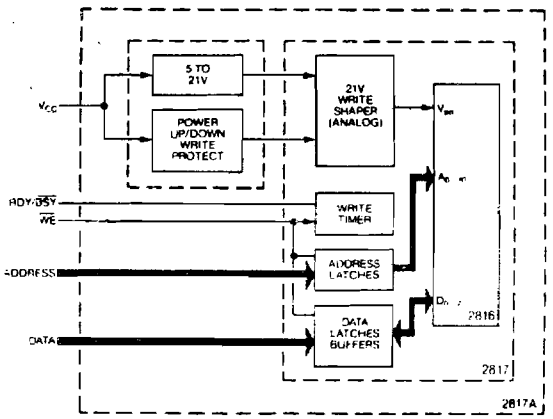
One of the major advantages of the EEPROMs is the ability to erase and reprogram a single byte. Unlike EPROMs, the entire chip does not have to be erased to alter the data. Some EEPROMs allow byte erasure and reprogramming in as little as 10 milliseconds. Also, certain types have entire chip erase mode which will erase the chip in 10 milliseconds in-circuit, versus the out-of-circuit 15-20 minute erasure required for EPROMs.

The first commercially available EEPROM was the Intel 2816. The 2816 was introduced in 1981. It represented the first step in EEPROM reliability, and demonstrated the viability of the floating gate technology. The 2816 incorporated the basic requirements for an EEPROM: 10 millisecond write/erase cycle; single byte erasability; 10,000 erase/write cycles per byte; and a 250 nanosecond access time.

The first generation EEPROMs required a number of external components for writing data to the chip. These circuits included address and data latches, a write timer to tell the other circuits when the EEPROM was writing, write-protect circuits to prevent accidental writing on power-up and power-down, and shaping circuitry to round off the leading edge of the write pulse. The shaping circuitry prevented sharp spikes from damaging the tunnel oxide layer. The absence of this circuitry seriously impaired the reliability of the EEPROM.

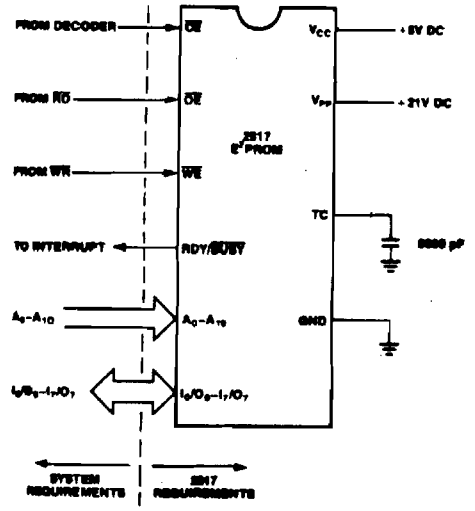
The 2817, a second generation EEPROM, incorporated much of the required external circuitry. Intel's goal was to make the chip operate as closely as possible to RAM. The newer 2817 contained address and data latches, write timer, write pulse, and power-up and power-down circuitry. In addition to eliminating the external circuitry, the internal latches allowed the processor to continue it's duties without having to wait for the completion of the 10 millisecond write/erase cycle. The latches allow the processor to spend the same amount of time writing to EEPROM as it would spend writing to RAM. The computer sends data to the EEPROM and the chip latches the data until the write cycle is completed. The 2817 incorporated a READY/BUSY line which prevented the processor from sending additional data until the chip was ready. Figure 18 shows the internal circuitry of the 2817.

FIGURE 18



Writing to the 2817 was as easy as writing to RAM because the 2817 incorporated an automatic erase-before-write. The address and data lines were latched with the WE line and the on-chip timer generated a programming pulse using an external 560 picofarad timing capacitor. The READY/BUSY line went low when a write was in progress, and returned high to enable another write or read. The only external circuitry required by the 2817 was a static 21 volt supply and a write-protect circuit. The write protect circuitry is required to prevent 21 volts (Vpp) from reaching the chip before 5 volts (Vcc). The time for the complete erase/write operation was rated at 20 milliseconds. The actual time was less than this due to a control loop which measured the charge on the floating gate. Once it detected the charge was sufficient to retain the data, it would discontinue the write process and return the READY/BUSY line high. Reading the 2817 was accomplished in the same manner as reading a 2716 EPROM. Figure 19 shows the system interface requirements and operation modes for the 2817 EEPROM.

FIGURE 19



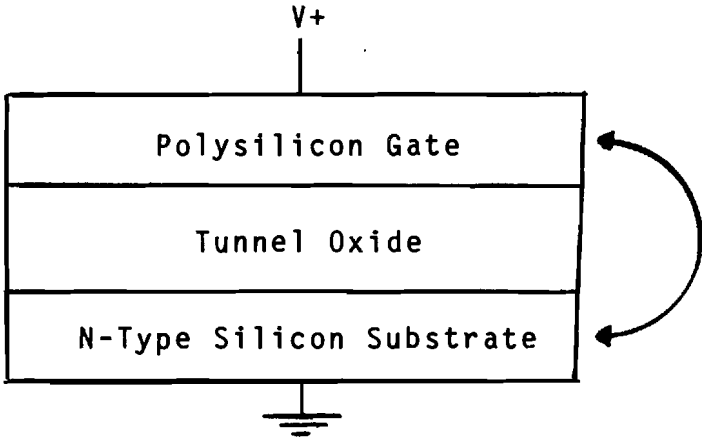
The latest version of the 2817 is the 2817A. This version can be programmed with a 5 volt-only supply. This eliminates the possibility of destroying the chip when 21 volts is applied without 5 volts being present, and makes it easier to incorporate the 2817A into existing systems. The 2817 and 2817A are the easiest EEPROMs to incorporate into an existing computer system. However, they are among the most difficult to program with an EPROM or EEPROM programmer because of the READY/BUSY line. The programmer must have some method of reading this line to determine the mode of operation of the 2817. In an existing system, this task is easily accomplished by connecting the line to the processor's interrupt line. An EEPROM programmer would require modification to be able to sense the READY/BUSY line. For this reason, few if any external programmers can handle the 2817 or 2817A.

EEPROM INTERNAL STRUCTURE

The design requirements for a non-volatile electrically erasable memory presented formidable problems for integrated circuit engineers. The device had to have long-term data retention, fast access time, and high density. It had to be designed for incorporation into existing systems and also be able to be included into the latest 16 & 32-bit technology. These requirements prompted the development of a new non-volatile technology, HMOS-E, and a new type of cell structure, floating-gate tunnel oxide, or FLOTOX.

It was originally believed that the EEPROM cell would be impossible to produce. After extensive research, engineers believed they could pass electrons through an oxide using Fowler-Nordheim tunnelling. When 10,000,000 volts/cm is applied across an insulator, electrons jump from the negative electrode to the positive electrode. In other words, the electrons would jump from gate to substrate. Figure 20 illustrates this process.

FIGURE 20



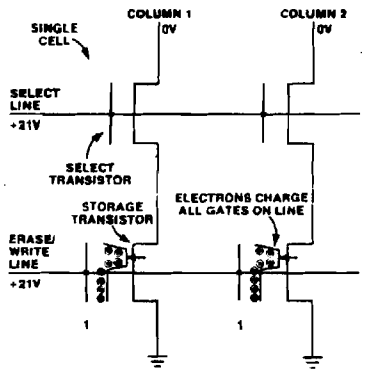
The Fowler-Nordheim tunneling presented two major problems. First, MOSFETs were normally operated at only 1,000,000 volts/cm, one-tenth the voltage required for tunneling. Second, to induce tunneling with voltages of about 21 volts, the oxide had to be less than 200 Angstroms thick. At that time, oxide thicknesses of less than 500 Angstoms had never been produced. It was believed that oxide inconsistencies would be too high at thicknesses below 500 Angstroms.

However, tunneling offered too many advantages to be ignored. Tunneling is a low-energy process requiring extremely low currents. It also offerred the capability to charge as well as discharge the FET gate. The tunnel could also be made very small, making it ideal for producing high-density circuits. These factors initiated research into producing oxides of less than 200 Angstroms. Eventually this led to the development of a cell structure called FLOTOX (see

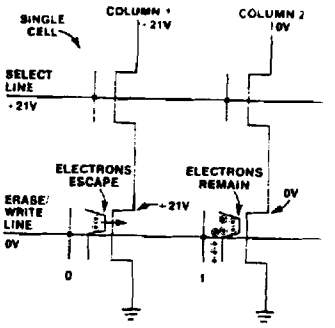
figure 17)

The FLOTOX cell was very similar to the FAMOS cell with the exception of an additional tunnel-oxide layer over the FET drain. When Vpp (21 volts) is applied to the gate, and the drain is grounded (0 volts), the floating gate is capacitively coupled to a positive voltage. Electrons are drawn through the tunnel to charge the floating gate. Applying 21 volts to the drain and 0 volts to the gate, discharges the gate into the tunnel. The physical structure of an EEPROM FLOTOX cell is shown in figure 21.

FIGURE 21



Schematic of Memory Cell Operation During Erase



Schematic of Memory Cell Operation During Write

Each FLOTOX circuit consists of two MOSFETs as shown in figure 21. The cell consists of a storage transistor (the actual FLOTOX device), and a select transistor. When the storage transistor is discharged, the select transistor prevents current from leaking into nonselected memory cells. The select transistor also prevents the storage transistor from discharging when an adjacent column is high. In figure 21, the select line and program lines correspond to the gate, and the column is the drain of the MOSFET. To erase the chip (store a positive charge on the gate), 21 volts is applied to the gate, while the drain is at 0 volts. This process can be used to erase single cells or the entire matrix.

FLOTOX technology has been susceptible to two types of failure: Tunnel oxide breakdown and oxide breakdown of the row select circuit. Approximately 88 percent of device failure is due to tunnel oxide breakdown, while only 10 percent of device failures was caused by defective row select circuitry. Either of these conditions can trap charges during

the erase/write cycle. If this occurs, the oxide retains enough residual charge to make it difficult to discharge a particular cell.

The reliability of EEPROMs is quite excellent. They are designed to withstand up to 10,000 write cycles, and to store data for over 20 years. 10,000 write cycles is equivalent to altering the data 3 times a day for 10 years. Intel has found that approximately 99 percent of the bytes in their EEPROMs are still functionally reliable after as many as 100,000 write cycles.

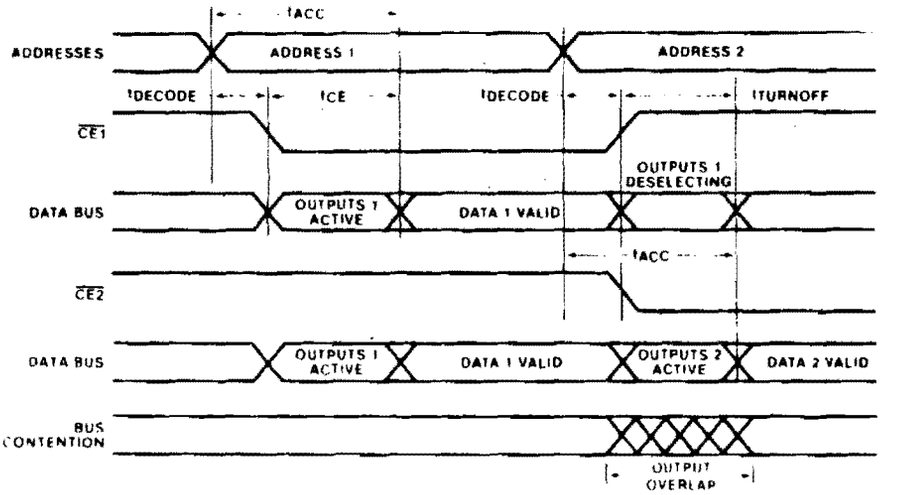
NEW EEPROMS AND PROGRAMMING

Within the last two years, Intel, Seeq, Xicor, and Exel have developed 5 volt programmable EEPROMs. The Intel 2816A and the Seeq 5213 require external latches and timing circuitry. The Exel XL2816A, Xicor X2816A, and Intel 2817A incorporate the necessary timing and latch circuitry. These chips also feature an automatic erase-before-write allowing them to be accessed as RAM for read and write operations.

The most popular 16k EEPROMs are the Exel and Xicor 2816 and the Intel 2817A. These devices are popular because of the limited external support circuitry required for programming. Three popular devices for direct 2716 replacement are the Intel 2816A, the Seeq 5213 or 52B13, and the Xicor 2816. The chips are pin compatible with the 2716, and once programmed can be read just like a 2716. The mode selection and timing diagram for Intel's 2816A is shown in Figure 22.

FIGURE 22

MODE \ PIN	$\overline{CE}$ (18)	$\overline{OE}$ (20)	$\overline{WE}$ (21)	INPUTS/ OUTPUTS
Read	V_{IL}	V_{IL}	V_{IH}	D_{OUT}
Standby	V_{IH}	Don't Care	Don't Care	High Z
Byte Erase	V_{IL}	V_{IH}	V_{IL}	$D_{IN}=V_{IH}$
Byte Write	V_{IL}	V_{IH}	V_{IL}	D_{IN}
Chip Erase	V_{IL}	+9 to +15V	V_{IL}	$D_{IN}=V_{IH}$
No Operation	V_{IL}	V_{IH}	V_{IH}	High Z

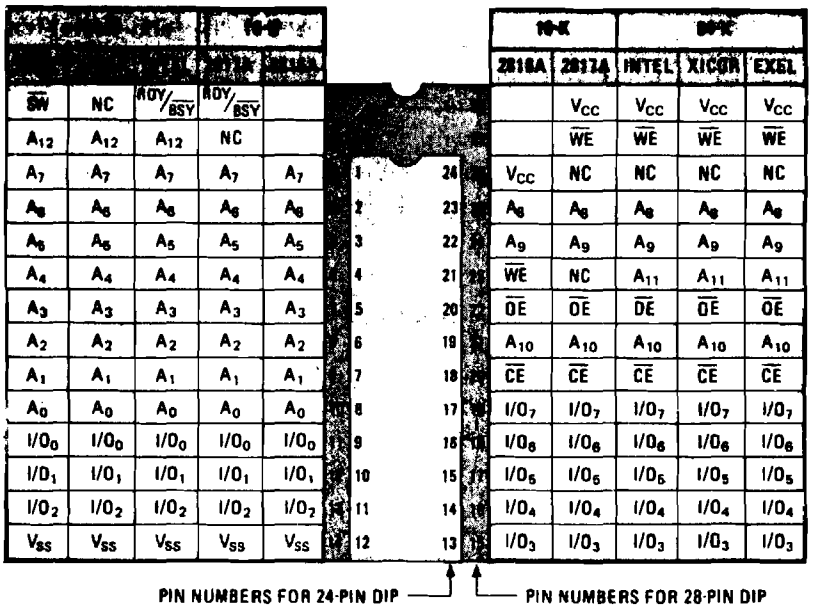


The Xicor 2816 is very similar to the Intel 2816A, but the Xicor version requires fewer external support devices. The Xicor 2816 has on-chip latches, a timing-pulse generator, a high-voltage pulse generator, and write-protect circuitry. The write cycle takes 10 milliseconds to complete, but once triggered, the cycle is automatic. This enables the processor to perform other duties between write cycles.

To write a particular location, that byte must be erased prior to the write. Erasing is accomplished by selecting the address, latching all data pins high, and pulsing the CE and WE. The OE must be held high during a write/erase cycle. The pulse to the WE pin must be at least 9 milliseconds and should not exceed 70 milliseconds. Once the location is erased, a write is accomplished in the same manner, except the state of the data lines is different.

A number of the manufacturers have announced densities of 64k. All the new chips are supposed to conform to the JEDEC (Joint Electron Device Engineering Council) standards for 28 pin packages. Figure 23 lists many of the new EEPROMs and their pinouts.

FIGURE 23



						2816A	2817A	INTEL	XICOR	EXEL
$\overline{SW}$	NC	$RDY/\overline{BSY}$	$RDY/\overline{BSY}$				V_{CC}	V_{CC}	V_{CC}	V_{CC}
A_{12}	A_{12}	A_{12}	NC				$\overline{WE}$	$\overline{WE}$	$\overline{WE}$	$\overline{WE}$
A_7	A_7	A_7	A_7	A_7	1	24	V_{CC}	NC	NC	NC	NC
A_6	A_6	A_6	A_6	A_6	2	23	A_8	A_8	A_8	A_8	A_8
A_5	A_5	A_5	A_5	A_5	3	22	A_9	A_9	A_9	A_9	A_9
A_4	A_4	A_4	A_4	A_4	4	21	$\overline{WE}$	NC	A_{11}	A_{11}	A_{11}
A_3	A_3	A_3	A_3	A_3	5	20	$\overline{OE}$	$\overline{OE}$	$\overline{OE}$	$\overline{OE}$	$\overline{OE}$
A_2	A_2	A_2	A_2	A_2	6	19	A_{10}	A_{10}	A_{10}	A_{10}	A_{10}
A_1	A_1	A_1	A_1	A_1	7	18	$\overline{CE}$	$\overline{CE}$	$\overline{CE}$	$\overline{CE}$	$\overline{CE}$
A_0	A_0	A_0	A_0	A_0	8	17	I/O_7	I/O_7	I/O_7	I/O_7	I/O_7
I/O_0	I/O_0	I/O_0	I/O_0	I/O_0	9	16	I/O_6	I/O_6	I/O_6	I/O_6	I/O_6
I/O_1	I/O_1	I/O_1	I/O_1	I/O_1	10	15	I/O_5	I/O_5	I/O_5	I/O_5	I/O_5
I/O_2	I/O_2	I/O_2	I/O_2	I/O_2	11	14	I/O_4	I/O_4	I/O_4	I/O_4	I/O_4
V_{SS}	V_{SS}	V_{SS}	V_{SS}	V_{SS}	12	13	I/O_3	I/O_3	I/O_3	I/O_3	I/O_3

PIN NUMBERS FOR 24-PIN DIP — PIN NUMBERS FOR 28-PIN DIP

Many of the new 64k EEPROMs incorporate features not offered on the 16k chips. For example, Xicor has developed Data Polling which enables the processor to determine if the write cycle is complete without sacrificing a pin like the 2817A. The Data Polling pin will double as A14 enabling the 28 pin package to be used up to a density of 256k. One of the most popular new features is a page-mode used to store more than one byte of data during a single write cycle. Page-mode techniques could improve the speed of loading data by a factor of 64. The problem with this technique is that it drastically reduces the expected life of the chip.

To overcome the limitations of the page-mode, Exel developed a pin called the Status Word (SW). It allows for features like page-mode to be controlled by user software. The Status Word uses a set of registers that may be read or written to when the SW pin is low. The SW pin allows other special functions like a fast-write mode to be incorporated into the chip without the need for additional pins.

Since EEPROMs maintain data on power-up and power-down, they are susceptible to false write cycles. These can occur during power transients when the WE control is low, and the chip is still receiving enough power to initiate a write operation. Early EEPROMs did not have write-protect circuitry. Manufacturers have attempted to solve this problem with gated control lines, voltage sensors, or on-chip timers. Figure 24 compares features of the more popular EEPROMs. Pinouts of many of the popular EEPROMs are included at the end of this chapter.

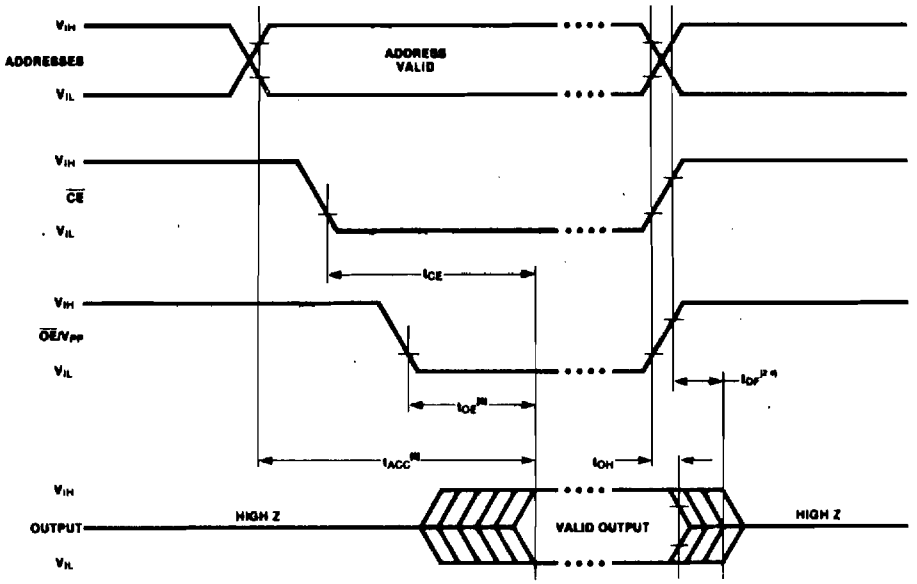
FIGURE 24

Mfr. Part No.	No. Bits	5 Volt Programmable?	Addr/Data latches?	Autowrite timeout?	Page Mode?	Power-up/down protection	Pin #1 Function
Intel 2816	16-K	no	no	no	no	not needed	A_7
Intel 2817	16-K	no	yes	yes	no	not needed	V_{PP}
Intel 2817A	16-K	yes	yes	yes	no	good	RDY/$\overline{BSY}$
Excel 2816A	16-K	yes	yes	yes	no	fair	A_7
Xicor 2816A	16-K	yes	yes	yes	no	marginal	A_7
Seeq 52813	16-K	yes	yes	no	no	marginal	A_7
Excel 48C64	64-K	yes	yes	yes	32-byte	foolproof	status-word control
Xicor 2864A	64-K	yes	yes	yes	16 byte	good	no connection
Intel 64K	64-K	yes	yes	yes	16-byte	good	RDY/$\overline{BSY}$
Inmos 3630	64-K	yes	yes	no	64-byte	marginal	program/erase control

Previous sections of this chapter have covered the basic requirements for programming EPROMs and EEPROMs. Attempting to program these chips without a commercial ''burner'' is virtually impossible and potentially very costly in terms of damaged chips.

The Timing diagram of a typical EPROM is shown in Figure 25.

FIGURE 25



To properly program an EPROM without damaging the matrix, the switching of the mode-select lines and the voltages must be precisely controlled. If a programming pulse is too long, or the voltages are too high, the chip will be permanently damaged. It is for these reasons that programming of EPROMs or EEPROMs should only be attempted with a commercial programmer.

A number of EPROM programmers have been tested with the VIC-20, C-64 and PET computers. The results were far from satisfactory. Some programmers supplied improper voltages, destroying the chips, while others altered the data they were attempting to ''burn'' onto the chip. However, there is one programmer which is without question the most versatile device for working with EPROMs or EEPROMs on the C-64 or VIC-20. The programmer is manufactured by a California-based company called JASON-RANHEIM (JR). The programmer retails for $99.00, but contains features not found on most programmers costing 20 times as much. JR also offers cartridge boards which hold up to 16k, and two boards, the PCC-4 and PCC-8, that can bank switch up to 128k and 256k respectively! These boards come with download/run software that enables the board

to download up to to 38k at a time into the computer. It is even designed to download and execute a program on power-up, enabling the computer to continuously run the same software without the need to re-load the software if the power fails! These boards make it possible to store your favorite word processor, database, or spreadsheet on a single board and load the one you wish to use into the computer in approximately 3 seconds! JR products are available through CSM Software Inc.

The PROMENADE C1 can program single bytes or entire chips. It also has the capability to store files on EPROMs and recall them from program control. The PROMENADE has 4 programming voltages, 2 EPROM supply voltages, 3 intelligent programming algorithms, 15 bit chip addressing, and 3 LED's that tell the user precisely what the programmer is doing. The PROMENADE has the capability to program the following chips:

2758	462732P	68766	
2516	2564	5133	:EEPROM
2716	2764	5143	''
27C16	27C64	2815	''
2532	27128	2816	''
2732	27256	2816A	''
27C32	27512*	X2816	''
2732A	68764	48016	''
		52B13	''

* requires external switching

The best feature of this EPROM programmer is that it is very easy to use. Just plug the programmer into the USER PORT and load the PROMOS software. The PROMOS software works in conjunction with most ML monitors that are not located at $9000 (address $9000). The PROMOS software is also available on a cartridge with HESMON and the DOS wedge built in. The rest of this chapter will be devoted to the use of the JR programmer.

One important point to remember when using the JR programmer is that the PROMOS software normally resides at $9000 (36864 decimal), so the program you wish to put on EPROM should be located in another area of memory when it is being ''burned'' (more on this later).

There are six control commands which are ''wedged'' into the computer's operating system by the PROMOS software. These are PI (shift up-arrow), English Pound, Z, shifted-E, shift-S, and shift-L. The first three are the ones that you will use almost exclusively.

(PI): This key (shift up-arrow) is used to place the PROMENADE into programming mode. The programming parameters should be typed as follows:

(PI)[mem start],[mem end],[EPROM start],[CW],[PMW]

The PROMENADE needs the starting and ending locations of the program to be ''burned'', the location where it is to start in the EPROM, and the control and programming words (more on these later). For example, (PI)8192,16383,0,5,7 would program a 2764 EPROM with the data from 8192 to 16383 ($2000 to $3FFF hex). It would start at the first byte in the EPROM, and program the chip using intelligent method #1. When sending commands to the PROMENADE you must make sure you don't tell it to program more bytes than there are on the chip. For example, a 64k (8Kx8) EPROM contains 8192 bytes of available space. In our example, 16383-8192=8191 not 8192! You must subtract one from the total number of bytes you wish to program because the programmer is starting at location 0 not location 1 in the EPROM. If you tell the programmer to program too many bytes, it won't be detected until the end of the ''burn'' cycle. The yellow error light will flash, and you will have to erase and reprogram the data.

English Pound (EP): This key is the opposite of the PI key. It is used to read data from the EPROM into the computer. The only difference is that you don't need the Program Method Word (PMW). For example (EP)8192,16383,0,5 would transfer the entire contents of a 64k EPROM into the computer starting at 8192.

Z: The 'Z' command is used to 'zero' the PROMENADE socket. On power-up, all the lights on the PROMENADE will be lit. This means that the socket is 'live' and must be turned off before inserting a chip. If the socket is not turned off, the chip could be damaged.

shift-E: This command is designed to erase the 48016 EEPROM in 2/10's of a second. Other types of EEPROMs must be erased by programming a 1's ($FF hex) into the locations that are to be reprogrammed.

shift-s: Takes data from a section of memory and produces a program file which is then ''burned'' to a chip.

shift-l: This command retrieves a file from a chip and loads it into memory. This command is only designed to read files into memory. It is rarely used and will not be covered in the rest of this chapter.

The PROMENADE has 3 lights that tell the user precisely what the programmer is doing. The GREEN light means the PROMENADE is receiving power from the computer. The Red light means that the socket is ''active'', or that certain lines on the socket are now at conditions other than 0 (logic low). Finally, the yellow light indicates that the PROMENADE is programming the chip in the socket. If there is an error while programming, the yellow light will flash and programming will stop.

To illustrate the programming of an EPROM, the following example assumes you wish to create or duplicate an auto-start 8k cartridge ($8000-$9FFF hex, or 32768-40959). The only

problem with the PROMENADE is that it requires the memory locations to be in decimal, and ML monitors require all operations to be in hex. This sometimes get confusing, and makes it very difficult to work with a ML monitor and the PROMENADE without hex-decimal calculator. You must also remember to relocate code which exists from $8000-$9FFF because this is where the PROMOS software normally resides (actually, it relocates itself just before the end of the BASIC area). The best type of ML monitor to use is a monitor that resides at $C000, like HIMON. The Hesmon/PROMOS package can be difficult to use when trying to work with routines that reside at $8000 because that is where it resides. The following steps will outline the method for programming a 2764 EPROM with 8k of code which resides at $8000.

1. Use the monitor to relocate the 8k of code from $8000 to $2000 (8192 decimal). This is usually the best place to relocate code because it gives you plenty of room for large programs. For example:

T 8000 9FFF 2000

This will transfer the program from the place where it normally operates ($8000), to an area that will allow you to ''burn'' it on a chip ($2000). This does not alter the code in any way. When the chip is inserted into the computer it will operate normally, assuming you programmed it correctly. The next step is to save the code that was transferred to $2000.

S''program name'',08,2000,4000

You must save the copy at $2000, not at $8000, so when it is loaded back into the computer to be ''burned'' it will load at $2000 and not at $8000. Once the code is saved, turn the computer off to clear the memory.

2. The next step is to load and run the PROMOS software. You should get a message saying that PROMOS is active. Once that is done, load the copy of the program that was saved at $2000.

LOAD''program name'',8,1

It must be loaded with a '8,1'. If it is not, it will load at $0800 and you will ''burn'' whatever garbage is residing at $2000.

3. Next, 'zero' the PROMENADE socket with the 'Z' command. Insert the 2764 chip into the socket and lock it in. BE SURE TO CORRECTLY ORIENT THE CHIP IN THE SOCKET. THE CHIP CAN BE DAMAGED IF IT IS INSERTED BACKWARDS.

4. Now the chip can be programmed using the (PI) command.

(PI)8192,16383,0,5,3

This programs the chip with the data from $2000-$3FFF. MAKE SURE YOU HAVE ENTERED ALL THE PARAMETERS CORRECTLY. IF THEY ARE WRONG, ESPECIALLY THE CW, THE CHIP COULD BE DAMAGED. With these parameters, it will take approximately 7 minutes to program a 2764. This time can be reduced with the use of other PMW's.

5. When the PROMENADE is finished programming the lights will shut off. If there was an error, the yellow light will flash. An error could result from wrong parameters or the inability to program the correct data. This usually results from incomplete erasure of the chips. If there were no errors, the chip can be inserted into a board and tested in the computer. If you followed the directions above, the program should run. More information on loading and running programs and files is outlined in the PROMENADE's manual.

The Control Word (CW) and the Program Method Word (PMW) are two very important features of the PROMENADE. The CW is used to tell tell the PROMENADE the type of EPROM that is in the socket. The CW tells the programmer the programming voltage (Vpp), the pin to receive Vpp, the pin to receive the programming pulse, and the standby logic level in the read mode. The following table will explain the parameters:

CONTROL WORD

BITS		FUNCTION
0	1	voltage control
0	0	25 volts
1	0	21 volts
0	1	12.5 volts
1	1	5 volts
2	3	pin select for Vpp
0	0	pin 22
1	0	pin 1
0	1	pin 23
1	1	pin 1 set low. select mode for 52B33
4	5	pin select for Programming Pulse
0	0	pulse pin 27
1	0	pulse pin 22
0	1	pulse pin 20
1	1	no action
6	7	standby logic level of pin 20 on a read
0	0	set low: no action on read
1	0	set low: taken high on read
0	1	set high: no action on read
1	1	set high: taken low on read

PROGRAM METHOD WORD

The PMW tells the PROMENADE how to program the EPROM you chose with the CW. Basically, the PMW controls the pulse duration of the programming pulse. There are four groups of PMW's available: 0-3, 4-7, 8-11 and 12-15. These groups correspond to the standard method, Intelligent method 1, Intelligent method 2, and Intelligent method 3. The standard method takes the most amount of time because it assumes the worst case for every byte. Most bytes will not require nearly as long as the ''worst-case'' byte. The Intelligent PMW's reduce overall programming time by testing the EPROM as it is programming. The following table lists the pulse times for the standard PMW:

PMW	Pulse Duration in milliseconds
0	6
1	12
2	24
3	48

The next table lists the test pulse time in milliseconds, and the maximum permitted programming time before failure in milliseconds.

PMW	METHOD	TEST PULSE	MAX PROG TIME
4	1	var	12ms
5	1	var	25ms
6	1	var	50ms
7	1	var	100ms
8	2	.25ms	75ms
9	2	.5ms	75ms
10	2	1.0ms	75ms
11	2	2.0ms	75ms
12	3	.25ms	100ms
13	3	.5ms	100ms
14	3	1.0ms	100ms
15	3	2.0ms	100ms

Bits 0-3 of the PMW control the duration of the programming pulse. PMW of 16 or greater are used to write-protect files, or make files non-relocatable. PMW's of greater than 16 are used in conjunction with the shift-S command. Bit 4 of the PMW is used to produce non-relocatable files. Any file can be made non-relocatable by adding 16 to the PMW you choose (this sets bit 4 high). Bit 5 write-protects files and can be implemented by adding 32 to the PMW (this sets bit 5 high). If you want to write-protect and make a file non-relocatable, add 48 to the PMW (sets bits 4 and 5 high).

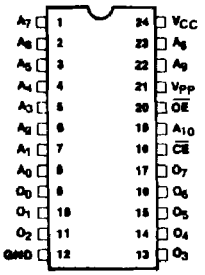
There are only two features of the PROMENADE which could be improved. First, the PROMOS software requires the memory parameters to be entered in decimal. This becomes very confusing when using the PROMOS software in conjunction with an ML monitor (although Hesmon does have decimal-hex conversion). Second, there is no way of knowing what type of error has occurred when the yellow error light flashes. These two minor drawbacks don't take away from the versatility of the PROMENADE. It is the best EPROM programmer available for any micro computer system.

The only other device you will need to program EPROMs is an EPROM eraser. The best, low-cost eraser is the DATARASE manufactured by Walling. This eraser holds two chips and will erase them in about 10 minutes. It retails for $34.95 and is available from CSM Software Inc.

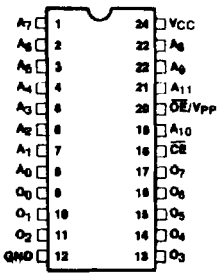
The final section of this chapter is a reference section listing of many EPROM and EEPROM pinouts.

EPROM AND EEPROM PINOUTS

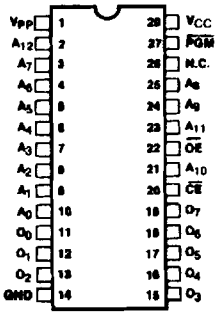
2716



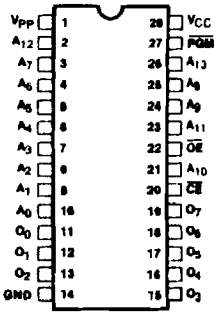
2732A



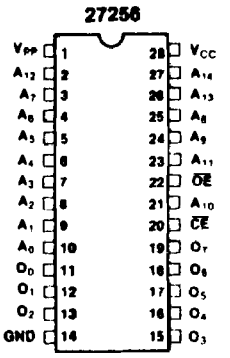
2764

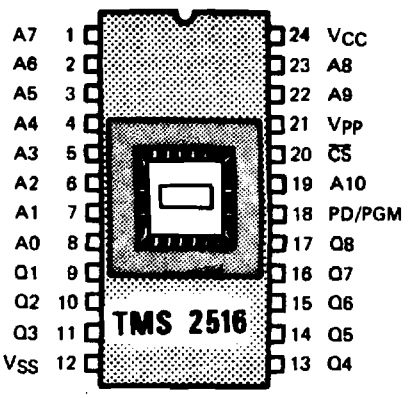


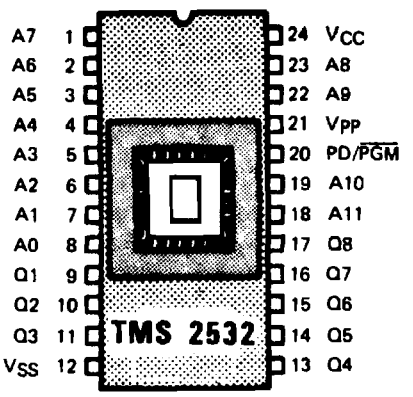
27128



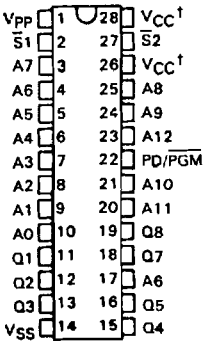
27256

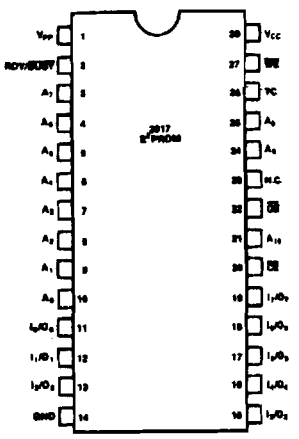


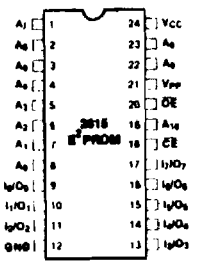


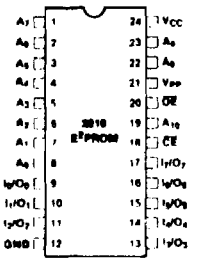


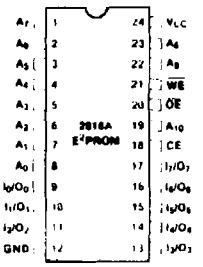
TMS2564 . . . JL OR JDL PACKAGE
SMJ2564 . . . J PACKAGE
(TOP VIEW)

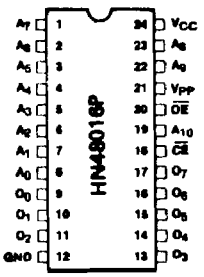


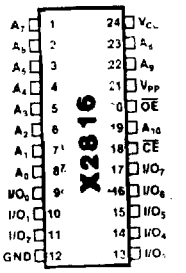


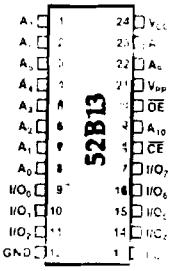












The 1541 disk drive will format the new disk to be read and write compatible with many other Commodore (R) disk drives. The proper syntax to format a disk is:

```
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 will be used in discussing the disk drive.

DEFINITIONS:

ARCHIVAL COPY - A copy of a program to be used only in the event the original program should fail. In the USA, the legal owner of a copyrighted program is the only person that may possess an archival copy. Any unauthorized copying or duplication of a copyrighted program is illegal.

AUTO-BOOT - A ML program that will, upon loading into the computer, load another program and execute that program. The term auto-boot and auto-load are synonymous

BOOT - Refers to getting a computer or computer program started. Often times the early computers needed a little help to get them going. As with all equipment, a little nudge (i.e. from your boot) may be required to get it going.

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

BACKING UP - The process of making a copy of a program. - Many time this refers to making a copy of something that never should have been saved in the first place.

BINARY - The native number system of the computer. The binary number system contains only the numbers 0 and 1.

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. $04, FF, 00, 82).

COMPILER - A program used to convert BASIC (or other languages) into ML or a modified ML called P-CODE. P-Code or ML code will often times execute many times faster than the original code that was used to generate it.

COPYRIGHT - A legal method of protecting computer software from being copied. No one should ever make an unauthorized copy of copyrighted software.

CRASH - Refers to the erratic functioning of a program or to the program ceasing entirely. The program may crash due to the fault of the programmer (a bug) or to the program failing to pass its protection scheme (as in a copy disk).

CURSOR - What a pirate does when the bootleg copy he obtained does not work. Just checking to see if anybody actually reads definitions.

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 are

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. The term DOS has also been applied to the ML routine that a programmer may use in the disk drive to accomplish a specific task.

EEPROM - ELECTRICALLY Eraseable Programmable Read Only Memory. This is a computer chip that may be programmed by the user (see the EPROM section later in this manual). This chip will retain its memory even when the power is turned off. The chip may be erased electrically by an EPROM programmer. After eraser the chip may be re-programmed

ENCRYPTION - A method of coding the data on the disk. Encrypted programs will appear to be 'garbage' when the program is on the disk.

EPROM - Eraseable Programmable Read Only Memory. This is a computer chip that may be programmed by the user (see the EPROM section later in this manual). This chip will retain its memory even when the power is turned off. The chip may be erased by exposing it to ultra-violet light. After erasure the chip may be re-programmed.

GUARD BAND - The area directly adjacent to both sides of the track. The guard band is erased by the read/write head whenever the disk drive writes data to the disk. This is done to prevent interference between adjacent tracks. The guard band will also erase data that has been placed on the half track.

GCR - Group Cyclic Recording. A big fancy name for the way that the 1541 stores data on the disk. In a nutshell, the 1541 will take 4 bits of data and expand it into 5 bits (8 bits into 10). This is done to insure that there is never more than two consecutive 0's on the disk.

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.

HALF TRACKS - Data is stored on the disk in concentric circles called tracks. Each track is approximately 0.020 inches apart, center to center. Half tracks is a method whereby the data is stored in between the tracks. On the 1541 the data may only be reliably stored on the track or on the half tracks. The read/write head is too wide to permit the use of adjacent tracks and half tracks.

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 - Hexadecimal. 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.

INTERRUPT - This is a process whereby the microprocessor may be forced to suspend its normal operation and begin another operation. The microprocessor will first complete the command that it is currently being processed and then it will service the interrupt. Three normal type of interrupts are: NON MASKABLE INTERRUPT, INTERRUPT REQUEST and the BREAK INSTRUCTION. A RESET of the computer will also force the microprocessor to suspend its normal operation.

I/O - Refers to the terms INPUT and OUTPUT. I/O is generally used to refer to the communication between two (or more) computer chips or peripherals. It may also be used when you have bought more computer equipment that you can afford (I owe).

NON-STANDARD SECTORS - Any deviation from the normal sector pattern of the 1541 format. Normally the sectors will be in sequential order (0-20). Tracks 1-17 will have 21 sectors each, tracks 18-24 have 19 sectors each, tracks 25-30 have 18 sectors each and tracks 31-35 have 17 sectors each. Duplicate sectors, displaced sectors, extra sectors or missing sectors all may be considered non-standard sectors.

NYBBLE COUNTING - This term is a carry over from the APPLE computers. Actually, on the 1541 disk drive we count bytes (8 bits) not nybbles (4 bits). Nybble counting refers to the actual number of bytes on a track. The number of bytes contain on a track will vary depending upon drive speed, the brand of disk used, temperature, humidity etc. Even the best of disk drives will show a variation in the number of bytes written to a track.

PLA - PROGRAMMABLE LOGIC ARRAY - The memory management chip in the C-64. The PLA will control which section of memory that the 6510 microprocessor will access. The PLA can configure memory so that the 6510 microprocessor can have access to ROM or RAM at the same memory location (although only RAM or ROM may be accessed at any one time).

RAM - Random Access Memory: This is the part of your computer's 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.

RESET - A hardware method where by the operation of the microprocessor is immediately suspended. This may be accomplished by momentarily grounding the RESET line to the microprocessor.

ROM - Read Only Memory: This is the part of your computer's memory that is a permanent part of the computer. ROM cannot be changed, modified or erased. The ROM in your computer allows you to 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 computer's brains.

R/W HEAD - READ/WRITE HEAD - This is the actual mechanism that reads data from the disk or writes data to the disk.

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 track 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.

SYNCHRONIZED TRACKS - Tracks that were written in alignment to a particular reference point. Usually this is done on a disk drive that starts writing each track immediately after the timing hole on the disk breaks a photo-electric beam. The 1541 does not use the timing hole as a reference point. Usually this technique is performed on a programmable disk duplication machine.

SYNC MARK - A special sequence of bits stored on a track that synchronizes the data that follows it to the READ/WRITE circuitry of the disk drive. On the 1541 disk drive 40 '1' bits are used to provide a sync mark.

SPIRAL TRACKING - SEE TRACK ARCING - A variation of the track arcing technique whereby the data is written out across several tracks and half-tracks. Each track or half track contains only a small amount of data. This insures that there will not be any cross talk between the adjacent tracks and half tracks. The tracks are not truely spirals, they are actually a 'stepped spiral'. Spiral tracking is another variation of synchronized tracks. If the programmer knows how the data on the various tracks has been written out, the programmer may write a variation of the synchronized track routine to read the data back in.

TRACK - A concentric ring (circle) used for storing information on the disk. There are 35 tracks on the 1541 format. Track 1 is the outter-most, track 18 contains the BAM and directory and track 35 is the innermost. Although only 35 tracks are used by the 1541 most drives are capable of using 40.

TRACK ARCING - This refers to a technique that writes out a few sectors of data to a track, then the R/W head of the disk drive is stepped 1/2 track, a few more sectors are written. Because the data on the track and the adjacent half track are not side by side there will not be any problems in reading the data that was written. Track arcing uses routines similar to synchronized tracks and spiral tracking.

UNDOCUMENTED OPCODES - These are opcodes that actually do cause the microprocessor to perform a specific function. While the function does indeed get performed, the manufacturer will not guarantee that every chip will perform the same function for a given opcode.

COPYRIGHT NOTICE

PROGRAM PROTECTION MANUAL FOR THE C-64 VOLUME II
COPYRIGHT 1985 (C) BY CSM SOFTWARE INC
ALL RIGHTS RESERVED

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 CSM SOFTWARE INC.

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 CSM SOFTWARE INC.

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 CSM SOFTWARE INC.

WARRANTY AND LIABILITY

Neither CSM SOFTWARE INC., nor any dealer or distributor makes any warranty, express or implied, with respect to this manual, the disk 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 will CSM SOFTWARE INC. 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. CSM SOFTWARE INC. will not assume any other warranty or liability. Nor do they authorize any other person to assume any other warranty or liability for them, in connection with the sale of their products.

UPDATES AND REVISIONS

CSM SOFTWARE INC. reserves the right to correct and/or improve this manual and the related disk at any time without notice and without responsibility to provide these changes to prior purchasers of the program.

IMPORTANT NOTICE

THIS PRODUCT IS SOLD SOLELY FOR THE ENTERTAINMENT AND EDUCATION OF THE PURCHASER. IT IS ILLEGAL TO SELL OR DISTRIBUTE COPIES OF COPYRIGHTED PROGRAMS. THIS PRODUCT DOES NOT CONDONE SOFTWARE PIRACY NOR DOES IT CONDONE ANY OTHER ILLEGAL ACT.