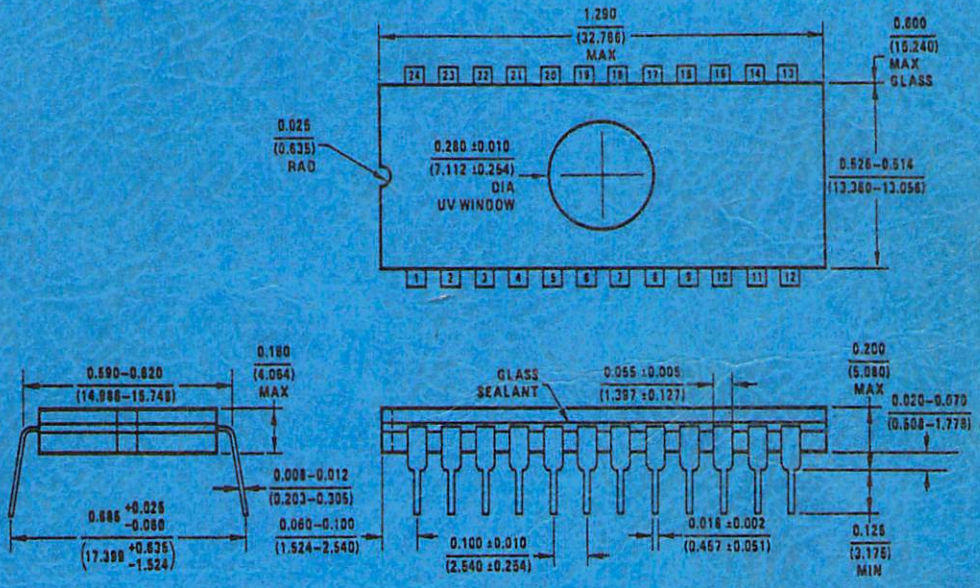
EPROM PROGRAMMERS HANDBOOK

FOR THE C64 AND C128



EPROM PROGRAMMERS HANDBOOK

Written by Bill Mellon

Edited by T.N. Simstad

FOREWORD

Welcome to the EPROM PROGRAMMERS HANDBOOK! This unique book is for anyone who wants to learn how to program EPROMs for the Commodore 64. In this book we assume that you have some general experience with your C64 already. Some experience with machine language and hexadecimal will help you pick up the concepts faster. You'll need an EPROM programmer too, of course. The EPROM programmer we use here at CSM is the PROMENADE C1 from Jason-Ranheim. We feel you just can't buy a better programmer for the C64. Although the examples in the book are based on the PROMENADE, you should be able to adapt the commands to your programmer if you have a different model.

This book is the result of many, many hours of research and testing. We hope that you find it a treasure mine of information about EPROMs, cartridges and the C64. The accompanying diskette contains many useful utilities to help you use your EPROM programmer to the fullest. We are anxious to hear your comments about the book, so drop us a line.

Finally, CSM SOFTWARE INC. would like to acknowledge the help and information that many manufacturers of products used in this book were so gracious to provide. Last but not least, all of us at CSM would like to thank our many fine customers - past, present and future - without whom this book and many other things would not be possible.

COPYRIGHT NOTICE

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

INTRODUCTION TO EPROMS

What is a EPROM? How do you program them? Why would you want to program an EPROM? Once it is programmed what do you do with it? During the course of this manual we will answer these and other questions about EPROMs.

Of all the various facets of home computing, EPROMs are one of the least understood and the most useful. Very little is known by the average user about EPROMs or how to use them. Many people are afraid of opening up their computers, disk drives and cartridges. They don't really know what they are looking at, how the chips work or what they do. Yet, when a user begins to understand the power and versatility of EPROMs, one becomes hooked. A type of addiction soon takes over the user; before long you'll find yourself wanting to put all of your favorite programs on EPROM.

Programs, once put on an EPROM, require virtually no time at all to load. Imagine turning on your computer and instantly having your favorite word processor, data base, spreadsheet or utility up and running. Just by plugging in a cartridge and turning on your computer you can be ready to get to work! No long LOAD times, what used to take minutes to load may require only a fraction of a second from a cartridge. Not only that, it is possible to put many different programs on a single cartridge and pick your favorite from a menu. Let's take a quick look at some of the different possibilities that the use of EPROMs bring to mind:

1. Putting a single program on a cartridge

2. Putting a number of different programs on a single cartridge with a menu that allows you to select the desired one.

3. Using BANK SWITCHED cartridges that allow you to put 256K of memory on a single board. Think of all the various programs that you will fit in 256K!! You can add a menu that allows you to pick and choose programs at will.

4. Modifying the KERNAL or BASIC ROM of the computer so that you have your own totally customized computer. It is possible to change screen colors, title messages on power-up, default device numbers and much more.

5. Modify the DOS (Disk Operating System) for a custom disk drive. Extra tracks and sectors, high speed data transfer routines.

6. Use the PROMENADE as device #16. This allows you LOAD and SAVE files from EPROMs, as you would on your disk drive. Imagine loading your favorite program from an EPROM.

7. Program EPROMs that can be used in other computers. The PROMENADE may be used to program a wide variety of EPROMS.

8. Create your own, totally new, operating system for the C-64 (or other computer) and end up with a specialized, dedicated computer. This is extremely useful in scientific applications, business and industrial control applications.

Now that we have looked at some of the applications let's examine a few of the fundamentals of EPROMs and EPROM programming.

Let's start off by looking at what an EPROM is and how it works. An **EPROM** is a type of memory chip, i.e., an electronic storage device, similiar in nature to a disk. Data, programs or other types of information can be put onto an EPROM for later recall, just as the same data may be saved to a disk and later recalled. Data may be erased from an EPROM and the EPROM may be reprogrammed, similiar to erasing and copying a disk.

EPROMs are related to other types of memory chips such as ROMs and PROMs. **ROM** stands for **R**ead **O**nly **M**emory. 'Read only' means that we can only read information from this memory chip, not write to it. That is, we can examine and use the information contained in a ROM, but we can't change it. ROM chips come pre-programmed directly from the manufacturer and cannot be changed or modified in any way. Once a ROM is programmed it will retain its data indefinitely, without requiring any power to do so. In other words, we may turn the computer's power off and when it is turned back on the ROM will have retained all the data.

PROM stands for **P**rogrammable **R**ead **O**nly **M**emory. The word 'Programmable' refers to the fact that the user may program this chip. This programming is a one-shot affair; once it is programmed the code may not be altered or erased. In this manner the PROM is very closely related to the ROM. Like the ROM, the PROM does not require any power in order to retain its data.

Next we have the **EPROM**, **E**raseable **P**rogrammable **R**ead **O**nly **M**emory. The EPROM may be programmed and erased thousands of times. The EPROM will perform identically to a ROM when it is installed in a circuit. As before, the EPROM does not require any power in order to retain its data. The major difference comes when it is time to change or modify the data contained on the chip. With a ROM or a PROM it is not possible to reprogram them, whereas an EPROM may be erased and re-programmed. One can easily see that it is very advantageous to have a chip that may be reused time and time again.

Closely related to ROM, PROM, and EPROM is the RAM chip. **RAM**, which is **R**andom **A**ccess **M**emory, means that we can read OR write to any byte on the chip any time we want. Another difference between RAM and ROM is that the contents of RAM disappear when the power is removed. ROMs, on the other hand, hold onto their data without needing any power. The operating system of the C-64 is stored in a type of ROM so it is instantly available when the machine is powered up. The term ROM is sometimes used loosely to include PROMs and EPROMs too.

Remember, true ROMs have to have their information put on at the factory, so they aren't much use to us. Much more useful is a **PROM** - a **P**rogrammable ROM. 'Programmable' means that we can put whatever information we desire on the chip ourselves, by 'programming' it with a special device such as the PROMENADE 'programmer'. However, we can only program a true PROM once since there is no way to erase it. This brings us to EPROMs. An **EPROM** is an **E**rasable PROM. Not only can we program the chip ourselves, but we can also erase and reprogram it many times. Erasing the chip is not the same as writing to it, for two reasons. First, the entire chip is erased at the same time, whereas the chip is written to one byte at a time. Second, when an EPROM is erased, each bit of the entire chip is filled with 1's. Writing to (programming) an EPROM can only change a 1 bit to a 0 bit. When an EPROM is programmed the only bits that get changed are those that contain 0's. The 1 bits are left as is. EPROMs are erased using special ultraviolet (UV) light. Complete erasure of an EPROM generally takes from 5 to 15 minutes.

The EPROM has a very special relative called the **EEPROM**. The **EEPROM** is an Electrically Eraseable PROM. This means the EEPROM may be erased by an electric signal rather than UV light. Also, single bytes can be erased without erasing the rest of the chip. While the PROMENADE is capable of programming and erasing many EEPROMs, they are relatively expensive and they don't have any advantages over EPROMs as far as this book is concerned. We'll stick to EPROM programming in this book.

There are many types of EPROMS which differ in memory size, organization, speed, number of pins and other features. This may sound like a bewildering set of factors, but almost all projects for the C-64 can be accomplished with a couple of common chip types. The memory size (storage capacity) of the chip is the most important factor when you're choosing an EPROM. The size of a chip is indicated by the chip number. For instance, we'll be using the popular 2700-series EPROMs, which include the 2732, 2764, 27128, 27256 and 27512. The digits after the 27 represent the size of the chip in **bits**. A 2732 has 32K bits, a 2764 has 64K bits, etc. (1K=1024). Since there are 8 bits per byte, you must divide the number of **bits** by 8 to convert it to **bytes**. This means a 2732 has 4K bytes, a 2764 has 8K bytes, etc. on up to the 27512, which has 64K bytes! The 2764 chip (8K) will be the most common EPROM in this book by far. It can be used in an 8K or 16K cartridge or with a special adaptor to replace the KERNAL, BASIC, or 1541 DOS chips.

The size of a chip in bits is not the whole story. The organization of the chip is also important, that is, how the bits are grouped. This is also called the 'width' or word size of the chip. In the 2700 series, the bits are organized into 8-bit groups. When you request a single piece of data from the chip, it consists of 8 bits in one 'chunk'. Other types of ROM or RAM chips may retrieve their data in groups of 4 bits (nybbles), or even as single bits. As we saw, a 2764 has 8K bytes with 8 bits each. This is described as **8Kx8** (notice how 8Kx8 = 64K bits).

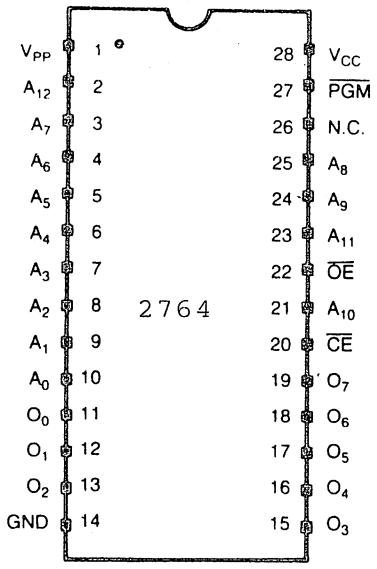
The speed or access time of an EPROM refers to how fast the chip can respond when data is requested from it. The access time is measured in **nanoseconds (ns.)**, which means **billionths** of a second. That's fast! One billion (1,000,000,000) nanoseconds is only one second, while a billion seconds is about 32 years! The C-64's internal clock 'ticks' a mere **million** times a second, or once every **microsecond**. For technical reasons, EPROMs and other memory chips for the C-64 must be more than twice as fast as this clock. Since 1 microsecond equals 1000 ns., we need access times less than 500 ns. The most common speed to use is 450 ns. Faster chips are OK to use but they usually cost more and the extra speed doesn't make any difference. The speed of a chip is usually printed on it right after the chip number. A 450 ns. 2764 may be labelled "2764-450" or just "2764-45". A 300 ns. chip might even be labelled "2764-3".

EPROMs are supplied in ceramic DIPs (dual inline packages) with many metal 'legs' or pins for connecting the EPROM to the outside world. The actual silicon chip at the heart of an EPROM is much smaller than the case, typically 1/4 inch square or smaller. It is visible through the window on top of the EPROM. The EPROMs we'll be concerned with have either 24 or 28 pins. The 2732 has 24 pins and the 2764, 27128 and 27256 have 28 pins. As with other integrated circuits (ICs), the pins are numbered in a standard way. One end of the EPROM has a notch or dot on it. Looking at the EPROM from the top, the pins are numbered **counterclockwise** from the notch, starting with pin #1 (see figure 1-1). It is very important to remember which end of an EPROM is which, since inserting it in a circuit the wrong way can damage the EPROM. Usually the circuit board and/or the socket will also have a mark to aid in inserting the EPROM properly.

Other differences among EPROMs include power consumption, programming and erasure time. Power consumption is not an important factor when choosing EPROMs for the C-64. However, since the C-64's power supply is barely sufficient for normal use, we recommend that you don't have more than 10 additional EPROMs plugged into the computer at any one time. Programming and erasure time will vary considerably from one manufacturer to another (even from chip to chip), so the best guide to erasing EPROM is your own experience. If you find a brand you like, try to stay with it. This shouldn't be a major concern, though, so don't fight rather than switch! (especially if the price is right).

Before we move on to the next chapter there is one more point to consider when looking at EPROMs; the size of the MATRIX. The MATRIX is what you see when you look through the window of an EPROM. For the beginning EPROM user, the bigger, the better. Large MATRIX EPROMs will be more forgiving to programming errors. This is due to the physical size of the internal components. The larger the internal structure, the greater the abuse that the chip can withstand before failing. There is absolutely nothing wrong with small MATRIX chips, we just don't recommend them for beginners. During the first few times that you program EPROMs you're likely to make a few mistakes. Give yourself some latitiude and get the big MATRIX chips if you have a choice.

Figure 1-1: Standard IC Pin Numbering (2764)



HOW EPROMS WORK

In this chapter we will try present a layman's view of how EPROMs work, why it possible to program them, how to erase them and how to avoid some of the more common problems associated with EPROM programming.

When the user first tries to program an EPROM, there is a large number of unanswered questions. What happens when a chip gets programmed? What actually is going on inside the chip? If I experience a problem, how do I erase the chip? What happens when the chip is erased? How do I know if a chip is fully ersased?

An awful lot of questions, right? You can rest assured that we will clarify the common concerns of all those who program EPROMs. Any time you learn a new concept there are sure to be some unanswered questions and unexpected problems. We feel that if you have a better understanding of how an EPROM works and what goes on inside, you'll be more comfortable with them.

One problem that many people have experienced is that of buying used or surplus EPROMs (you wouldn't buy used disks would you?) Don't buy used EPROMs, factory seconds or surplus goods. You just can't be sure of the quality that you are getting. Some of the surplus EPROMs are quite reliable, some are not, so why take a chance? Until you gain the experience necessary to judge for yourself whether a surplus EPROM is good or not, stay away from them.

EPROMs have evolved during the past few years. When they first came out they were failure-prone, subject to quality control problems and very expensive. With the great leaps in technology that have occurred during the past few years, the EPROM manufacturers have learned to overcome many, if not all, of the weaknesses of the early EPROMs. Today, EPROMs are top quality, long-lasting and very cost-effective. The price of EPROMs has come way down and at the same time the quality has gone way up. What all this means to you, the end user, is more reliable memory for less money.

The reason that we reflect back on the early days of EPROMs is because many people are simply not aware of the great advances that have been made. This also allows us concentrate our efforts in this book to writing about the newer and more reliable memory chips available today. Much of the information written in this chapter just was not true with the older EPROMs. When you are looking for EPROMs to buy, insist upon first quality, factory fresh EPROMs, not surplus or seconds. Fifteen years ago the floppy disk was considered unreliable and failure-prone. During the past few years we have seen advances come about where floppy disks have become very reliable, so it is with the EPROM. Don't let anyone tell you that EPROMs are unreliable or expensive - it just ain't so any more...

Let's get down to just what goes on inside an EPROM. We are not going to concern you here with a bunch of meaningless facts and figures. Rather than confuse and bore you, we would like to present an overview of just what is going on inside the chip. The information that we are presenting here is not going to make you program EPROMs better, but it may just clear up some of the mystery surrounding an EPROM's operation.

When we receive an EPROM from the factory it is fully erased. This means that all the bits have been set to a value of 1. EPROMs are erased by exposing the MATRIX to Ultra Violet (UV) light. This causes all the bits to return to the value of 1. The MATRIX is the part that is visible when you look through the window of an EPROM. When an EPROM is programmed, the MATRIX is the part that actually stores the data. The data is stored as a series of 1's and 0's in the MATRIX. Since the chip is fully erased before we program it (all bits 1), all that is necessary to do is create the 0 bits by programming. The 1 bits are left as is. When an EPROM is erased, it is the MATRIX that is affected and all the bits are returned to the value of 1.

In the following illustration we will be examining the 2764 EPROM (8K x 8). This is the most common type of EPROM that you will encounter. The 2764 contains 8K (8192) bytes of data and each byte is composed of 8 bits each (hence, 8K x 8). Each of the bytes contains 8 bits and 8 bits only. Each bit may only be a 1 or a 0. There is no other possible state for a bit. In an EPROM, any time a bit is refered to as being a '1' it also means that the bit is 'ON' or that an internal connection is made. When a bit is referred to as a '0', it means that the bit is 'OFF' or that an internal connection is broken (not connected). Now we're ready to see what goes on inside an EPROM.

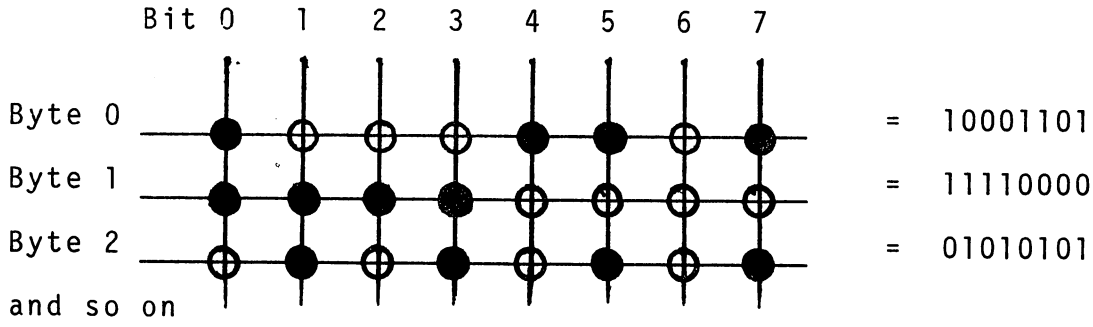
Let's take a simplified look at how an EPROM works. This will be a conceptual view, without getting into the actual physical construction of the MATRIX (although it is similar to the model described here). Consult figure 2-1. Each **byte** in the EPROM is represented by one **row** (horizontal line) in the diagram. One byte is eight bits. The 8 **columns** (vertical lines) represent the 8 **bits** of each byte. At each intersection of a row and column is a tiny 'fuse' representing a single bit. When the fuse is intact, electricity can flow from the row line to the column line. This results in a '1' bit. When the fuse is 'blown' no current flows, which gives us a '0' bit. In a **PROM**, these fuses are just like household fuses since once they're blown, they're blown for good. In an **EPROM**, the fuses are more like 'circuit breakers' since they can be reset after being blown, if desired. When you erase a chip with UV light, it resets all of the fuses back to the connected state (1's). Let the engineers worry about **why** UV light erases EPROMs - just be glad that it works.

Bytes are numbered (addressed) internally in the EPROM starting at 0, regardless of where in the computer's memory the EPROM is located. Remember, the internal memory of the EPROM **ALWAYS** begins at byte 0. It is the computer that determines where an EPROM resides (i.e. $2000 or $8000 or $E000). This is important to remember, since many people are confused about this point. In later chapters we will see how the address of the EPROM in computer's memory is determined. To read a byte of data from an EPROM, the microprocessor sends the byte's address to the EPROM's address input lines. Special 'row decoding' circuitry (not shown) selects the row corresponding to the requested byte. Electricity is then applied to that row line. Any intersection which has an intact fuse will allow current to cross over to the corresponding column line. Special detectors on each column line register a '1' bit whenever they detect this current on their own line. The output from these bit detectors is sent to the data lines of the EPROM, where the microprocessor can read it. In the top row of figure 2-1 (byte 0), the first fuse is intact, as shown by the solid circle, so we have a '1'. The second fuse is blown, indicated by the open circle, so we have a '0'. The complete byte is 10001101. That's all there is to it. Pretty simple, isn't it?

To 'burn' (program) a chip, a similar process is used. First the chip must be erased, by UV light, so that all fuses are intact. This means all bits are initially set to the value of 1 and that each byte is 11111111 ($FF in hex). The entire chip is programmed one byte at a time. First byte 0 is programmed, then byte 1, byte 2, etc. The address of the byte to be programmed is decoded by the chip and the proper row is selected. A relatively large voltage (appx. 25 volts) is applied to this row line. To program a "0" bit, the proper column line is grounded. This allows current to pass through the corresponding fuse, which 'blows' it. From then on, no current can pass through the fuse until it is reset by erasing the EPROM. No special process is needed to program a '1' bit. Since erased chips are already filled with 1's (intact fuses) the 1 bits will be left as is. Note that individual bytes are programmed one byte at a time, but the chip must be erased as a whole.

This simplified description ignores the control lines that must be manipulated in order to read and write the EPROM, as well as the underlying physical processes. For the person who wishes to use an EPROM programmer, no further explanation is necessary or warranted. If you find yourself desiring a more complete description of the physical and electrical subjects, see chap. 26 of the **PROGRAM PROTECTION MANUAL, Vol. II.**

Figure 2-1: EPROM Model



We now need to examine the process of erasing EPROMs. Erasing an EPROM restores all of the bits to a value of '1' (you may think of it as resetting the circuit breakers). In order to erase an EPROM, the MATRIX must be directly exposed to UV light. The UV light causes a reaction to occur within the MATRIX that restores all the bits to a value of 1. The actual process that occurs when an EPROM is erased is beyond the scope of this book. I feel it is not as important to know **WHY** an EPROM may be erased, as it is to know **HOW** to erase an EPROM properly.

Properly erasing EPROMs depends upon a number of factors including:

1. The type of UV light source (sun, fluorescent light, flame or commercial UV lights).
2. Distance of the EPROM from the source of light.
3. Manufacturer of chip (different manufacturers vary the internal structure).
4. MATRIX size (many manufacturers use different size MATRIXs for similiar chips).
5. Cleanliness of the window.
6. Length of exposure time to UV light source.

Of all factors affecting EPROM erasing, the two most important are the amount UV light reaching the chip and exposure time. The amount of UV light actually reaching the chip is influenced by the source of light, distance from the source and cleanliness of the window. All three of these factor may easily be controlled by cleaning the window before erasing and using a commercial EPROM eraser of known quality. The only real variable then becomes the time that the chip is exposed to the UV light of the commercial programmer.

An EPROM exposed to fluorescent light or to light from a flame (candle, match, etc.) may require many years to fully erase. An EPROM exposed to direct sunlight should be fully erased in 14 days or less. The same EPROM can be erased in as little as 2 minutes when exposed to the UV light from a commercial EPROM eraser. Typical erase times vary from 2 minutes (min.) to 20 minutes (max.) for commercial erasers. The EPROMs that we are currently seeing on the market normally may be fully erased in less that 10 minutes. For all our discussions in this book we will assume that you are using a commercial EPROM eraser. The only reason that we mention fluorescent light and sunlight is to make you aware of the potential for accidental erasure from these sources. We recommend that the window be covered with an opaque material or write protect tab to prevent accidental erasure.

Ordinarily, overexposure to UV light is not a problem, unless the EPROM is left in a commercial eraser for more than 2 days. The life of an EPROM is determined, to an extent, by the amount of time that it spends being erased. Generally speaking, the less time spent in erasing an EPROM the longer its life. If you can fully erase an EPROM in 2 minutes, don't expose it to 15 minutes of light. For example, let's assume brand 'X' EPROMs may be exposed to a total of 80 hours of UV light before damage to the chip occurs. If the time for full erasure is 2 minutes, we will be able to erase the chip 2,400 times before damage occurs. Whereas, if we routinely leave the same chip in the eraser for 15 minutes we will only experience 320 erasures before damage occurs to the same chip. While we are on the subject of EPROM erasing, we would like to recommend a low cost (currently $34.95), well-built EPROM eraser: the **DATARASE**. The DATARASE will erase two EPROMs at the same time and requires only 2 to five minutes on the average. The DATARASE is available directly from CSM SOFTWARE; call or write for current price and delivery.

For those of you who wish to build you own EPROM eraser here are some guidelines to use:

1. Be sure to make the eraser lightproof, thereby preventing **ANY** release of UV light from the unit. UV light can be hazardous to your health (eyes and skin in particular).
2. Recommended exopsure is: '15 WATT SECONDS PER SQUARE CENTIMETER OF 254 NANOMETER WAVELENGTH RADIATION'.
3. Practical exposure is achieved in most cases by placing the EPROM 1 inch from the UV lamp.

Again these are only guidelines by which to make your own EPROM eraser. We do not recommend that you attempt to build or modify an EPROM eraser.

Finally let's cover some of the more common items and/or mistakes that the beginners should be aware of:

1. Using surplus or used EPROMs - Don't do it, they just leave too much to chance.
2. Jumping in before they understand the basics of EPROM programming - Take a little time to learn everything you can before you burn your first EPROM.

3. Be sure to use the proper commands for your particular EPROM programmer - Each EPROM programmer uses new and different commands.
4. Watch out for static electricity - Static electricity can permanently damage EPROMs. Use care when handling EPROMs.
5. Don't bend the pins of the EPROM - you might just break one off.
6. Keep the pins of the EPROM clean and straight.
7. Keep the window covered when not erasing the EPROM - When erasing the EPROM be sure it is clean and free of all debris
8. An EPROM is fully erased when all the bits have returned to the value of '1'. This may be tested by reading an EPROM after erasing; all the bytes should be $FF (all bits will be 1).

I'm sure that you are all ready to move on now and actually burn an EPROM. So let's get to it.

BURNING EPROMS

Burning EPROMs, well, that's the whole purpose of this book. In this chapter we are going to try to communicate to you just how you actually program an EPROM. This will be a grass roots approach - making a copy of an EPROM, installing it in a cartridge and verifying that it actually works.

In this chapter and throughout the rest of this manual we will be working with the PROMENADE EPROM programmer. We feel that the PROMENADE is simply the best, easiest to use, most reliable, most versatile and cost-effective EPROM programmer on the market today. The PROMENADE will program many different kinds of PROMs, EPROMs and EEPROMs (Electrically Erasable Programmable Read Only Memory). We have seen a large number of very "sophisticated" and expensive EPROM programmers that could not even come close to the PROMENADE's capabilities. If you are using another brand of EPROM programmer it will be necessary for you to use a different set of commands that apply to your programmer. We will try tell you what each command does on the PROMENADE, so that you will be able to figure out the correct commands to substitute for your programmer.

THE PROMENADE

If you have just purchased a PROMENADE EPROM programmer, this chapter will help familiarize you with its most important features. If you're already familiar with your PROMENADE, you should still read this chapter in order to refresh and clarify your understanding. If you own some other type of EPROM programmer, you'll need to refer to your owners manual for specific commands, but you can still benefit from the general information in this chapter.

The PROMENADE EPROM programmer is manufactured by the JASON-RANHEIM COMPANY (J-R), and is available directly from CSM Software Inc. It is compatible with the Commodore VIC-20, C64 and C128. In this book we'll assume that you have a C64 (or C128 in C64 mode) since the VIC's limited memory is a real handicap. The PROMENADE is a very versatile and reliable instrument. In addition to all its capabilities, the PROMENADE is also very easy to use because of the PROMOS software (included with the PROMENADE). PROMOS gives you easy access to the PROMENADE's features by adding several commands to BASIC. These commands can also be used directly from a compatible machine language monitor (MICROMON, supplied on the disk with this book, is compatible with PROMOS). The retail price of the PROMENADE is $99.50 (as of 11/85), which includes an instruction manual and PROMOS diskette or tape. It is available directly from CSM Software Inc. CSM also offers cartridge boards including several bank-switched models, one of which is capable of holding up to 256K on a single board! The PROMENADE and other supplies such as EPROMs and EPROM erasers, cartridge boards and cases are all available from CSM Software Inc.

The PROMENADE itself consists of a circuit board enclosed in a brushed aluminum case which doubles as a static-free surface to rest EPROMs on. The device plugs into the user (modem) port at the left rear of the Commodore 64, 128 or VIC-20. A ZIF (zero insertion force) socket is mounted on the PROMENADE's circuit board. This makes inserting and removing EPROMs very easy. The ZIF has a small handle which locks the chip into the socket. Flip the handle down to lock the chip in place, lift up to release the chip. As shown in the diagram on the PROMENADE, chips must be placed in the ZIF with the notched end of the chip on the **LEFT.** The ZIF can accept both 28 and 24-pin chips. Chips with only 24 pins must be placed all the way to the **RIGHT** end of the ZIF. **CAUTION:** Inserting the chip incorrectly can damage it! The PROMENADE incorporates special circuitry to protect itself from improperly inserted EPROMs. Unfortunately, there is no way to protect the EPROM. Remember - notch on the left, chip to the right.

Three LEDs (lights) tell the status of the device at a glance. The green LED marked "PWR" indicates that power is being supplied to the PROMENADE. It should be stay lit steadily as long as the computer is powered on. If it dims or blinks, your power supply may be overloaded or on the verge of failure. When using the PROMENADE it's best to unplug any peripherals which draw power from the computer, such as a Datasette or a printer interface connected to the cassette port. Note that this problem is due to Commodore's power supply, not the PROMENADE. In particular, the power supply in the SX-64 is barely adequate for normal use, so we recommend that you do not use the PROMENADE with it.

The second LED is a red light marked "SKT". It is lit only when power is actually being supplied to the socket, namely during reading or programming a chip. Once again, the light should be steady any time it is lit. The third LED is yellow and marked "PGM". It serves two functions. During the process of programming a chip, it will be lit but may vary in brightness as programming pulses are applied to the chip. If any type of error is encountered while reading or programming a chip, the yellow light will begin flashing and the red socket light will go out. Common errors include trying to program a chip that is damaged or hasn't been completely erased, using the wrong control or programming words (see below), or inserting the chip in the socket incorrectly. The error condition should be cleared before using the PROMENADE again.

The "SKT" and "PGM" LEDs may also light up and stay lit when the computer is turned on or RESET, or when RUN/STOP-RESTORE is pressed. This is normal, but you should avoid doing any of these things while a chip is in the socket. You should also avoid removing or inserting a chip while either of these two lights are on. To extinguish these lights and initialize the PROMENADE for use, use the "Z" (zero) command described below. This command will also clear out an error condition (flashing "PGM" light).

PROMOS

The powerful features of the PROMENADE are accessed through a special program called PROMOS. You can think of PROMOS as a special EPROM programming language. It adds several new commands to BASIC that allow you to read and write EPROMS. The PROMOS commands can also be used from a compatible machine language (ML) monitor. The ML monitor MICROMON included on the program disk with this book is compatible with PROMOS, as is HESMON (tm) and HIMON (from the PPM Vol. I and II).

Rather than learn all of the PROMOS commands right now, we'll go ahead and burn our first EPROM using the most common commands. We'll explain these commands as we go, and they will be all you need to know for most of the work in this book. A later chapter on the PROMOS commands will cover each command in more detail.

Before we can use the PROMOS commands, we must load and execute the PROMOS software. PROMOS is supplied on the disk accompanying the PROMENADE. Both the VIC-20 and C64 versions are provided. PROMOS is also available on a cartridge for the C64 along with the HESMON (tm) ML monitor and the DOS wedge. This cartridge is available directly from CSM. In this chapter we'll cover just the disk version of PROMOS. The next chapter, on ML monitors and PROMOS, will cover how to use the PROMOS cartridge.

Use the following command to load the C64 disk version of PROMOS:

```
LOAD "PROMOS*", 8
```

There have been a couple of versions of PROMOS so far, but this command will work for both of them (version 1.1 is the latest as of the time this book is written, 11/85). PROMOS will load in like a BASIC program. Now enter a RUN command. When you run PROMOS, it will copy itself up to the high end of the BASIC area, and patch itself into the BASIC language. You will see the PROMOS start-up message appear on the screen. PROMOS is ready for action.

READING THE EPROM

Okay, once PROMOS is running we're ready to burn an EPROM. For this first example, we'll simply make a direct copy of another EPROM. Most of the projects and examples in this book will use the same type of EPROM, the 2764 (8K). We'll base our first example on that type of chip too. Try to find a commercial cartridge which has a 2764 chip in it. This type of chip has 28 pins (metal legs); other types of chips you encounter may have only 24 pins. If the chip in a cartridge has 28-pins, chances are it is a 2764. The number 2764 should appear on the chip somewhere, although it is often surrounded by other digits and letters. For example, one chip we came across was labeled HN482764G, but the only important point to us is that it does have 2764 in the name. Sometimes the chip's number will be covered up by a paper label.

Make sure the chip is mounted in a socket on the cartridge board. If the chip is not mounted in a socket, you shouldn't try to remove it. Although it is possible to desolder the chip from the board, it's very easy to damage the chip and the board doing so. If you can't find a cartridge with socketed 2764 in it, just practice on a blank 2764 and standard cartridge board. These materials are available from CSM. You'll need these materials for other projects in this book, so you should have some on hand anyway.

Removing a chip from its socket is pretty simple, but it should be done carefully so as not to bend or break the metal legs. A special IC pulling tool is made for this job, but you can also use a small screwdriver to pry up the chip. Work back and forth from one end of the chip to the other until the legs are free. Handle the chip carefully once it is out of the socket. Avoid touching the pins with your fingers, since static electricity can destroy a chip. You can safely set EPROMs on the metal surface of the PROMENADE, or use special antistatic foam.

You should always initialize the PROMENADE before you insert a chip into it. Type the letter Z and hit return. This does a PROMOS Zero command, which resets the PROMENADE's socket. If the red SKT or yellow PGM light was lit, they will be turned off. Now you may insert the EPROM. Since a 2764 has 28 pins, all you have to do is make sure the correct end of the chip is in the correct place. The end of the EPROM with the notch must ALWAYS go in the left side of the socket, to match the diagram on the PROMENADE. The handle on the socket should be pointing up when you insert the chip. Flip the lever down to lock the chip into the socket.

To make a copy of the chip, we will read the contents of the chip into the C64's memory and then burn the copy back out onto a blank EPROM. The PROMOS command to read a 2764 chip is:

```
£ 8192, 16383, 0, 5
```

The first character in the read command is the English pound sign, found near the upper right-hand corner of the keyboard. The first pair of numbers in the read command (8192, 16383) specify the area of the C64's memory to read the EPROM memory into. All numbers for PROMOS must be entered in DECIMAL (the numbers given above correspond to $2000 and $3FFF, respectively, in hex). The area from 8192 to 16383 is 8K long, the same size as a 2764 EPROM. The third number in the command (0) specifies where in EPROM memory to start reading. The memory in an EPROM is always numbered internally starting at byte 0, regardless of where in memory the EPROM normally resides. The 0 in this command specifies that we'll start reading at the very first byte of the EPROM.

The final number in the read command (5) is called the Control Word (CW). This number tells the PROMENADE what type of chip you have. It is very important to use the right CW for the chip you are trying to read. The wrong CW can damage your EPROM! The last page of the PROMENADE manual lists most common chip types and their associated CWs. For now, all you need to know is that 5 is the correct CW for a 2764 chip.

When you press RETURN after typing the read command, the red SKT light will come on briefly as the chip is read. If all goes well, the red light will go out when the command is finished. If the yellow light begins to flash instead, it means that an error has occurred. Maybe you inserted the chip in the PROMENADE wrong, or used the wrong CW, or the chip isn't a 2764. Check all of these possibilities before you try to read the chip again. Remember, don't just use trial and error when you're trying to read a chip; you could damage a valuable original EPROM. If you do encounter an error, initialize the PROMENADE with a Z command again to clear the flashing error light.

BURNING THE EPROM

Assuming all went well with the read command, we can turn around and write the memory back to a blank 2764 (by blank we mean fully erased - all bytes set to $FF). We could also save out a copy of the memory to a disk, but we would need an ML monitor to do it. Later on we'll cover some examples of saving EPROM memory to disk. For now, let's just burn the memory out onto a new EPROM. Insert a blank 2764 EPROM into the PROMENADE, with the notch on the chip to the left as always. Lock the handle down. Enter the following PROMOS command:

```
PI 8192, 16383, 0, 5, 7
```

DON'T use the letters "PI"; use the pi key on the keyboard (shift up-arrow, near the RESTORE key). You should recognize most of the other parameters in this command from the read command - the area of C64 memory to save onto the chip (8192, 16383), the starting address in the chip (0) and the CW (5). This command has one new parameter (7) that is only used when programming (writing to) a chip. It is called the Program Method Word (PMW) and it specifies the particular method to use in programming the chip. At this point, you don't need to worry about the differences between the various methods. There is no "right" method to use to program a chip, just recommended methods. Using the wrong PMW can't damage your chip, but it might not program the memory onto it reliably. For now, just follow our recommendation and use a PMW of 7.

When you press RETURN after typing the command, the red SKT and yellow PGM lights should come on. Depending on the type of chip, programming can take anywhere from 8 seconds to several minutes. If no errors occur, the two lights will go off again when the process is finished.

If an error occurs during the programming process, the yellow light will begin to flash. The most common errors are inserting the chip incorrectly in the PROMENADE, using the wrong CW and trying to program a chip that isn't blank. Check these possibilities before you try again. Be sure to use the Z command to initialize the PROMENADE before you try to program another chip.

Even if no errors are reported, it is good practice to verify that the chip was programmed correctly. For now, the best way to test the EPROM we've just burned is to put it into the cartridge we took the original EPROM out of and try it. Be sure to insert your EPROM copy in the cartridge socket the same way the original chip was - there is usually a notch or dot on the socket or board to match the notch on the chip. Also make sure all the pins of the EPROM are inserted into the corresponding holes in the socket. Get the pins started in the holes all around and then push straight down on the EPROM.

Now turn off your computer, insert the cartridge, and turn the computer back on. Most cartridges start themselves automatically when the power is turned on. You should check that the cartridge performs exactly the same as the original did by going through a couple of game screens or menu options, as the case may be. If the cartridge does not start up as usual, turn off the computer immediately! You may have inserted the EPROM in the cartridge improperly or made some other mistake. If you can't figure out what went wrong, try making another copy from the beginning - load PROMOS, read in the original chip, etc. Chances are, your EPROM copy will work perfectly the first time if you've followed instructions. Congratulations - you've just burned your first EPROM!

That was fun, wasn't it? What can we do for an encore? Well, once you've verified that your EPROM copy works, you probably don't need to keep it around. This brings us to erasing EPROMs. To erase the copy EPROM, you'll need a commercial EPROM eraser (unless you want to set the EPROM in the sun for a couple of weeks). Most common types of EPROMs, including the 2764 we used in this example, are erased using a particular wavelength of ultraviolet (UV) light. Sunlight contains some of this particular UV light, which is why it will erase an EPROM eventually if given enough time.

There are many commercial EPROM erasers on the market today. Some will erase 10 EPROMS at a time in only a few minutes, but they will cost you more than your PROMENADE did! There is an inexpensive alternative that will suit most users' needs. The DATARASE, available from CSM, will erase two EPROMS in 10 minutes or less. The DATARASE can erase all UV-erasable EPROM types, which includes all the EPROMs in this book. If you don't have one already, you should get a DATARASE or other type of EPROM eraser.

Before you insert your EPROM into the eraser, make sure that the quartz window on top of the EPROM is clean. Sometimes fingerprints or gum from labels will cloud the window. This can result in partial erasure of the EPROM or prolong the time needed to erase the EPROM. Other factors involved in the erasing process, such as the exact wavelength and intensity of the UV light, are discussed in the last chapter. With all of the other factors the same, exposure time is the most important factor in erasing EPROMs.

Erasure times will vary from one manufacturer to another, even with the same type of chip. Your experience with a particular brand of chip is the best guide. Start with an erasure time of 10 minutes. If that seems to work for your type of chips, try reducing the time a bit. If you aren't getting complete erasure (all $FF bytes), increase the time a little. A good habit to get into is setting a timer to remind you when to remove the EPROM. Although overexposing the EPROM a few times probably won't hurt it, it can cut down on the life of the chip eventually.

Well, this brings us to the end of our first lesson in burning EPROMs. If you're itching to try a real project, we've got some great ones lined up for you later on in the book. Before you tackle them, though, there's a few more things you should know. The next few chapters will cover these topics.

PROMOS

This chapter is intended as a reference to PROMOS and the PROMOS commands. In the other chapters, we always give you the appropriate PROMOS commands for whatever you are doing. However, you may want to refer to this chapter from time to time for a more complete explanation of some point. You'll also find this chapter valuable when you're working on projects that aren't from this book. For instance, if you encounter a new type of chip, this chapter can help you derive the correct control word to use with it. Or you may want to relocate PROMOS to another location because of a conflict with the file you are burning to EPROM. If you want to experiment with the PROMOS file commands, you'll find them covered in this chapter even though they aren't used elsewhere in the book. We suggest that you look over this chapter to see what's covered here, and then come back to it when the need arises.

LOADING PROMOS

In order to use the features of the PROMENADE, you must use the PROMOS software that is supplied with the PROMENADE. To start up PROMOS, put the PROMOS disk in the drive and type:

```
LOAD "PROMOS*", 8
RUN
```

PROMOS will respond with its startup message. In most cases, that's all you need to do to get started with PROMOS.

The startup message displays the PROMOS version number. As of the time this is written (11/85) the latest version is 1.1. You may have version 1.0 instead, in which case you should contact CSM for an update. All references in this book to PROMOS refer to version 1.1. The improvements in version 1.1 over version 1.0 are:

A) A "Z" command is done automatically when PROMOS is enabled.
B) 27512 EPROMS can be programmed.
C) The timing margin for MCM68764 is improved.
D) A bug in programming the last byte of 27256's is corrected.
E) Some of the bugs in the file commands may have been corrected.

Only one of these changes will affect us in this book. When PROMOS 1.1 is enabled, it automatically does a "Z" command. With version 1.0, you must do a "Z" command yourself BEFORE inserting a chip in the socket.

PROMOS is a machine language program which can be located almost anywhere in memory. When you load and run PROMOS, it copies itself up to the end of the BASIC area in memory. The end of BASIC is indicated by a pointer stored in locations 55-56 ($37-$38). This pointer is stored in standard low byte / high byte order. After PROMOS copies itself up below the end of BASIC, it sets the pointer down so that the BASIC area now ends where PROMOS begins. This protects PROMOS from being wiped out by BASIC variables.

You can use this "auto-relocating" feature to control where PROMOS resides. PROMOS normally puts itself at $96D0-9FFF (38608-40959). Sometimes this will conflict with the program you wish to "burn" onto EPROM. For instance, most standard cartridges reside at $8000-9FFF (32768-40959), which includes the normal PROMOS area. Although a program does not have to be "burned" onto EPROM from the same location in memory it normally resides at, that's usually the most convenient way to do it. If you wish, you can force PROMOS to relocate itself almost anywhere by changing the end of BASIC pointer before you run PROMOS. For example, if you alter the pointer at 55-56 so BASIC ends at $8000 (32768), PROMOS will relocate itself below this address. This will leave the area above $8000 free for your cartridge file.

To change the end of BASIC, simply poke the low byte of the new address into location 55 and the high byte into location 56. For example, to set the pointer to $8000, use POKE 55,0 and POKE 56,128 (since $80=128). Then load and run PROMOS as usual. PROMOS will relocate itself to the $76D0-7FFF area (30416-32767). Another convenient place to locate PROMOS is just below $CC00 (52224). This is a good place for PROMOS because the area above $CC00 is where the DOS WEDGE normally resides. Contrary to the PROMENADE manual, PROMOS and the wedge **are** compatible. To set the end of BASIC to $CC00, use POKE 55,0 and POKE 56,204 (since $CC=204). PROMOS will relocate itself to the $C2D0-CBFF area (49872-52223). When you locate PROMOS here (or anywhere above $A000=40960), you should set the end of BASIC pointer back down **after** running PROMOS. Set the pointer to a value in the regular BASIC area so that BASIC will be able to store its variables normally. Use POKE 55,0 and POKE 56,160 to set the pointer to its normal value ($A000). One other place to put PROMOS is worth mentioning. If you use POKE 55,192 and POKE 56,17 PROMOS will put itself at the lowest possible place in memory, $0890-11C0 (2192-4544).

After being relocated, PROMOS enables itself by patching into the IRQ vector at $0314-15 and the KERNAL vectors at $031A-33. Through these altered vectors it intercepts all references to its own commands. Hitting RUN/STOP-RESTORE will replace these vectors with their normal values and disable PROMOS. Using most monitors will also alter these vectors. To re-enable PROMOS without reloading it from disk, use SYS 668 (G $029D). PROMOS puts a jump command down at this address (in an unused area) that points to the start of its own code. If this pointer gets wiped out, you'll have to reload PROMOS from disk.

There is an alternative to loading PROMOS from disk. A special cartridge is available from CSM that contains the ML monitor HESMON 64 2.0 (TM) and the DOS WEDGE 5.1, as well as PROMOS 1.1. All this is included on one cartridge for about the same price as a HESMON cartridge alone. A HESMON manual is included too. With this cartridge you'll always have these three essential utilities available at a moment's notice. The cartridge resides in the usual 32768-40959 ($8000-9FFF) area for cartridges. This is a slight inconvenience in some cases, but the instant availability of HESMON, PROMOS and the wedge more than make up for it. If you are into EPROM programming and were considering buying a regular HESMON cartridge, by all means get this cartridge instead.

PROMOS COMMANDS

Let's look at a few of the PROMOS commands now. This is not a complete list of the commands that PROMOS supports, but these are the ones we'll use in the rest of the book.

Initialization command

Z - This command zeros (initializes) the PROMENADE. Must be used if you want to insert a chip in the PROMENADE and the "SKT" or "PGM" lights are lit. Also clears an error condition (flashing "PGM" light). This command will bring all of pins in the PROMENADE's ZIF socket to 0 volts. This insures that there won't be any problems when changing EPROMs.

Direct access commands

These are the only commands that we'll use to program chips in this book. With these commands you must specify a beginning and ending location in the C-64's memory for your information, as well as a beginning location on the chip. Bytes on a chip are always numbered starting from 0, regardless of where in memory the chip will ultimately reside. All addresses must be specified in DECIMAL for PROMOS.

£ start addr, end addr, chip start addr, CW - This command reads bytes directly from the chip, starting at the chip start address given. The bytes are stored in the area of C64 memory specified by the start and end addresses. The "CW" tells the PROMENADE what type of EPROM is being read (see below).

PI start addr, end addr, chip starting addr, CW, PMW - This command programs data onto a chip from the area of C64 memory specified by the start and end addresses. The data is put onto the chip beginning at the chip starting address you specify. Since our printer doesn't have the Greek letter pi, we'll use the letters PI to stand for it throughout the book. **Do not** type the letters PI yourself; use the pi symbol on the keyboard (shifted up-arrow, next to RESTORE). The "CW" tells the PROMENADE what type of EPROM is being programmed. The "PMW" specifies the method to use in programming the chip (see below).

CONTROL WORDS (CWs)

Almost all PROMOS commands require the use of a CONTROL WORD (CW). The CW is a number between 0 and 255 that tells the PROMENADE how to access the chip correctly. Each type of chip has a specific CW which must be used. **Using an incorrect CW can damage your EPROM!!** The appendix contains a list of the CW for many common chip types. This list is updated periodically. The most current version of the list may be obtained by writing to CSM. Please enclose a self-addressed, stamped envelope and a note explaining that you want the latest PROMOS CW list. Most chips you are likely to encounter are already on the list (as are all chips used in this book).

Even though the CW must be in decimal for all PROMOS commands, the easiest way to analyze it is to work in hex. The CW is always between 0 and 255 because it consists of a single byte. This byte is made up of four parts, each of which contributes two bits to the CW (see figure 3-1). The bits of the CW are numbered starting at bit 0 on the right end (least significant bit). The rightmost two bits (bits 0-1) control the voltage level used when programming the chip. Four voltage levels are available, which allows a large of number of different chips to be programmed.

Bits 2-3 of the CW control which pin of the **PROMENADE socket** to apply the programming voltage to (Vpp). This is the same as the **chip's** pin number for 28-pin chips. For 24-pin chips, the chip's pin number will be **lower** than the PROMENADE's socket number by two since 24-pin chips are inserted in the right end of the socket. Pin 1 of a **24-pin chip** corresponds to pin 3 of the **PROMENADE socket**, and so on. Bits 4-5 of the CW control which pin of the socket should be used for the programming enable signal (PGM). This line controls the length of the programming pulse. Finally, bits 6-7 determine how pin 20 of the socket will be used. Bit 7 specifies whether pin 20 (CE) will be set high or low normally. Bit 6 determines whether this pin will be left alone or inverted during a read operation.

Figure 4-1 summarizes the CW format. To create your own custom CW for a chip that isn't listed, select one option from each category (bits 6 & 7 are separate categories). Write down the bits corresponding to your choices (in order!) and then convert the resulting 8 bit number to decimal. There's another method which many people will find easier. Next to each bit or combination of bits is a number in parentheses. This gives the decimal value contributed to the CW when that particular option is chosen. To create a custom CW, simply add up the decimal numbers of the options you wish to select.

To break down a particular CW into its component options, convert the CW to binary and look up the options separately. Another way uses the decimal form of the CW. Go down the list starting with the first category (bit 7). Out of the choices in the category, subtract the highest possible decimal number (which may be 0) from the CW. The number you subtract will tell you which option in that category was specified by the CW. Take the result of the subtraction and go on to the next category.

Figure 4-1: CONTROL WORD (CW) FORMAT

Bits 7-6 - Pin 20 Level (CE)

Bit 7	Normal level
0 (0)	LOW
1(128)	HIGH

Bit 6	Action on read
0 (0)	No change
1 (64)	Invert level

Bits 5-4	PGM Pin Select
00 (0)	Pin 27
01 (16)	Pin 22
10 (32)	Pin 20
11 (48)	None

Bits 3-2	Vpp Pin Select
00 (0)	Pin 22
01 (4)	Pin 1
10 (8)	Pin 23
11 (12)	Pin 1 held LOW

Bits 1-0	Voltage Select
00 (0)	25 volts
01 (1)	21
10 (2)	12.5
11 (3)	5

Note: All pin numbers refer to the PROMENADE socket. Subtract 2 from the number given to obtain the corresponding pin for 24-pin chips.

Figure 4-2: PROGRAM METHOD WORD (PMW) FORMAT

Bit 5	File Write Protection
0 (0)	Not protected
1 (32)	Write protected

Bit 4	File Relocatability
0 (0)	Relocatable
1 (16)	Not relocatable

Program Method	Bits 3-0	Pulse length	Max. time
Standard	00 00 (0)	6 ms.	-
	01 (1)	12	-
	10 (2)	24	-
	11 (3)	48	-
Intelligent #1	01 00 (4)	.1 ms. & up	12 ms.
	01 (5)	same	25
	10 (6)	same	50
	11 (7)	same	100
Intelligent #2	10 00 (8)	0.25 ms.	75 ms.
	01 (9)	0.5	same
	10 (10)	1.0	same
	11 (11)	2.0	same
Intelligent #3	11 00 (12)	0.25 ms.	100 ms.
	01 (13)	0.5	same
	10 (14)	1.0	same
	11 (15)	2.0	same

PROGRAM METHOD WORDS (PMWs)

The PROGRAM METHOD WORD (PMW) is only used when you're programming a chip. Its main function is to specify the algorithm (method) used to program and verify the chip. Figure 4-2 details the PMW format. Bits 3-2 of the PMW select the programming method. Bits 1-0 select the programming pulse length and the maximum programming time per byte. The PMW is not as critical as the CW - there is no "right" PMW for a chip, although some PMWs may give more reliable results than others with the same chip. The PMW will affect how long it takes to program the chip. This will also depend on the size of the chip and the manufacturer. The standard programming method is the most reliable but takes the longest (about 7 minutes for 8K). The intelligent methods are much quicker and still quite reliable for most chips. The intelligent methods rank in reliablility from #3 (most reliable) to #1 (least reliable but fastest). If you have trouble programming a chip, try reverting to the standard method.

Bits 4 & 5 of the PMW are only used with the PROMOS file commands. Bit 4 controls whether the file will be relocatable when loaded back in from the EPROM. The shift-S (SAVE) command uses this bit to to make nonrelocatable files. Bit 5 controls whether the file is write protected or not. Write protecting your files is generally a good idea. Creating or analyzing a PMW can be done the same way we showed you for a CW.

PROMOS FILE COMMANDS

Most of the PROMOS file commands are simply extensions of the standard BASIC file commands (SAVE, LOAD, etc.). These commands allow you to treat the EPROM (or other chip) like a disk drive, complete with directory. Most PROMOS file commands are very similar to the corresponding disk commands, except that the PROMENADE is accessed as device 16. Generally, you can save and load program files and data files as usual. However, you must erase the entire chip to scratch, replace or rename a file (except with EEPROMS).

We have not used any PROMOS file commands in the rest of this book. We are covering them in this chapter for the sake of completeness only. We feel that these commands just aren't that useful when you come right down to it. Not only that, but they may not be entirely reliable either. We've pointed out some of the potential problems below. Overall, rather than create a directory on a chip with PROMOS, we prefer to use our MENU MAKER program to create a cartridge with a menu.

One disadvantage when using PROMOS files is that the PROMOS software must be loaded and run before you can access any files on the chip (of course, the PROMENADE must be plugged in too). Having to load PROMOS from disk seems to cancel out the convenience of loading your other files from chip. If you use PROMOS chip files a lot, you may want to get the PROMOS/HESMON/WEDGE cartridge, available from CSM. If you don't have PROMOS on cartridge, you'll have to make sure PROMOS isn't located in the same area as the program you are downloading. In any case, you may find that the program you download will not run properly with PROMOS enabled. Use RUN/STOP-RESTORE to disable PROMOS before running the program. If you are using a PROMOS cartridge, you'll probably have to switch it off too, to make the underlying RAM available (disable PROMOS before you switch off the cartridge).

Transferring program files from a disk to a chip is fairly simple, but data files are much more trouble. You'll have to write a special program to transfer a data file from disk to chip. Data files must be written onto the chip one byte or piece of data at a time. If you make a mistake while doing this, you'll have to erase the chip and start over. If you make a mistake opening or closing a data file, it is possible to mess up the chip directory. Although the previous files on the chip will still be OK, you may not be able to add any more files. If you want to put several programs and data files on a chip, your chances of making an error somewhere along the line are pretty good. To top it off, PROMOS 1.1 doesn't always handle numeric data correctly (see below).

In spite of the limitations, you may want to experiment with these commands to see if they suit your needs. The complete list of PROMOS file commands is given below. To help you get started, we'll go through a couple of examples. Turn off your computer, insert the PROMENADE and turn the computer back on. Load and run PROMOS (the following examples were tested with PROMOS 1.1; other versions may well give different results). We'll need a BASIC program to use in our examples, so type in the following short program:

```
10 REM SAMPLE BASIC PROGRAM FOR
20 REM PROMOS FILE COMMANDS
30 PRINT "HI THERE!"
```

First we'll save this program to an EPROM. Execute a Z command to initialize the PROMENADE (this may not be necessary but it's always a good idea). Now insert a 2764 EPROM in the PROMENADE (notch to the left as always) and lock down the handle. To save the BASIC program, use:

```
SAVE "SAMPLE:5", 16, 7
```

The ":5" in the filename is the CW (Control Word) for a 2764 chip. The "16" is the device number for the PROMENADE, and the "7" is a PMW (Program Method Word) suitable for the 2764.

While the program is being saved, the red SKT and yellow PGM lights on the PROMENADE will be lit. When it is finished, list the directory of the chip with the command:

```
$5
```

"$" is the directory command, and "5" is the CW. Press any key to step through the directory listing. You should see the following:

```
0     SAMPLE          P
99
```

The "0" tells where in the chip the program starts, and the "P" stands for a program file. The "99" tells where the next file you save will start, and also indicates the length of our SAMPLE program in bytes (99 - 0 = 99 bytes). This length includes a 21-byte directory entry. Rather than keep all the directory entries at the beginning of the chip, PROMOS puts each entry just before the program itself (sort of like tape directories).

Next let's load the file back off the chip. First do a NEW command to clear out the original program. Now enter the load command:

```
LOAD "SAMPLE:5", 16
```

Note the CW (:5) and device number (16).

The red SKT light will light up briefly while the program is loading. When it's done, list the program to see that it was actually saved and loaded correctly. You might also try using a VERIFY command to check that the file was saved correctly. Type:

```
VERIFY "SAMPLE:5", 16
```

The SKT light will come on again briefly. The message "VERIFYING" will be printed on the screen, and then the message "OK". **Note: "OK" will be printed whether or not there is a verify error.** If there was an error, the yellow PGM light will begin flashing. Use the Z command to clear the error.

Let's try to cause a verify error. List the program. Use the cursor to go up to line 30 and change the "HI THERE!" to "HO HO HO!". Now enter the verify command again. Surprised? There was no error reported at all - no flashing PGM light. **The VERIFY command is not reliable!** We recommend that you don't use it. We have deleted whole lines from a program and not gotten a verify error. This problem was encountered with PROMOS version 1.1 (the most recent update as this is written). The problem may be fixed in future updates.

Now let's try the special "nonrelocatable" file commands: shift-L (LOAD) and shift-S (SAVE). These can be used to load or save from any area of memory, not just the BASIC program area (shift-L is like LOAD"name",8,1). As a test program, let's use MICROMON 49152 from the program disk with this book. This program resides at 49152-53247 ($C000-CFFF) in memory. Put the program disk in the drive and load the program into memory:

```
LOAD "MICROMON 49152", 8, 1
```

Type NEW to reset the BASIC pointers and avoid potential "Out of Memory" errors. To save this program to the chip, enter:

```
(Shift-S)MICROMON:5, 49152, 53248, 7
```

The shift-S will look like a heart in the normal character set (uppercase/graphics). DO NOT put quotes around the file name, and DO NOT put any spaces between the shift-S and the name (unless you want the quotes or spaces to become part of the name). The ":5" is the CW, the next two numbers are the beginning and ending addresses of the area in the computer to save, and the "7" is the PMW. The ending address you use (53248) must be 1 byte past the actual ending address. Note that you don't have to specify device 16 (the PROMENADE), since it is assumed automatically with shift-S and shift-L.

When the program has been saved, list the directory again (use $5). You should see:

```
0      SAMPLE           P
99     MICROMON         P
4216
```

The MICROMON program now resides on the chip from byte 99 to byte 4215 (4096 bytes plus the 21-byte directory entry). To load the file back off the chip, use:

```
(Shift-L)MICROMON:5
```

The shift-L will look like a fat "L". DO NOT put quotes around the name or spaces after the shift-L. If you type the wrong name, the yellow light will flash. After the program is loaded, you might want to execute it in order to verify it. Use SYS 49152.

This brings us to data files. We mentioned above that PROMOS doesn't handle numeric data correctly in data files (at least version 1.1 doesn't). When you use PRINT# to save numeric data to a file on a chip, you must use a separate PRINT# statement for each number. This is not required when you write numeric data to a disk file. Also, you will not be able to read the number back correctly from a chip data file unless you PRINT# a comma or CHR$(13) (RETURN) character to the file after the number. This must be done in the following PRINT# statement, since the number must be in a PRINT# statement by itself. Even this technique doesn't work all the time (these problems may be fixed in a future update of PROMOS). PROMOS seems to handle string data correctly, but we can't guarantee it.

The full set of PROMOS file commands is documented below.

FILE COMMANDS

NOTE: All file commands except $, shift-L and shift-S can be used within BASIC programs. In BASIC programs or immediate mode commands, variables may be used in place of numbers or filenames. PROMOS commands may also be used from within many ML monitors (see the chapter on ML monitors and PROMOS).

$cw - Lists the chip directory to the screen. Each time you press a key another file entry is displayed. Use the cursor down key to step through the directory continuously. Each file entry consists of: the file location within the chip (starting from byte 0); the file name; the file type (Program or Data); and the file status (*=not closed). The number of bytes used so far on the chip appears at the end of the list. The "cw" is a special **c**ontrol **w**ord that varies depending on the type of chip used. Control words are discussed in the chapter on burning EPROMs.

SAVE "name:cw", 16, pmw - Saves a program onto the chip from the BASIC area, and inserts a file entry for the program into the chip directory. The "pmw" is a special **p**rogram **m**ethod **w**ord that varies depending on the type of chip (see the chapter on burning EPROMs).

LOAD "name:cw", 16 - Loads the specified program file into the BASIC area, reagrdless of the load address in the file. The file must be relocatable, i.e. it must have been created using the SAVE command above.

(Shift-S)name:cw, C64-start, C64-end, pmw - Saves a program file to the chip from any area in C64 memory, specified by the C64-start and end addresses. The ending address you supply must be 1 greater than the actual end of the program. The shift-S must be the first character on the line (it looks like a heart). No quotes are allowed around the file name, and the name must begin immediately after the shift-S. The file created by this command is non-relocatable, i.e., it must be loaded back into the same area of C64 memory that it was saved out from (using the shift-L command below).

(Shift-L)name:cw - Loads a file produced by shift-S back into the area of memory it was saved from. The shift-L must be the first character on the line (it looks like a thick L). No quotes are allowed around the file name, and the name must begin immediately after the shift-L. Ycan also use this command to load files created by the regular SAVE command.

VERIFY "name:cw", 16 - Verifies that a program stored on the chip with the SAVE command matches the program in the BASIC area. This command is not reliable!

OPEN filenum, 16, pmw, "name:cw" - Opens a **data** file on the chip. It cannot be used to open a program file. If the file does not already exist when you try to OPEN it, PROMOS will create a directory entry for it. Newly created files **MUST** be closed before any other files are created or you may corrupt the directory!. PROMOS will let you write to OR read from a file once it is open. NOTE: If you reopen a file that has already been created and closed once, writing to the file can corrupt the data already stored! To prevent this, add 32 to the usual PMW when you first OPEN the file to create it. Once it has been closed, the file will be protected from further writing.

PRINT#filenum, data - Writes data directly into the file specified by filenum. You should only write to files which have just been created by OPEN! Data can consist of quoted strings, string variables, numbers, numeric variables or CHR$() codes (in theory, at least - PROMOS 1.1 does not handle numbers correctly).

CMD filenum - Directs all output from regular PRINT or LIST statements to the specified file. More convenient than using PRINT# but may be undependable. Once again, use with newly OPENed files only!

INPUT#filenum, variable(s) - Inputs data from an open file on the chip and places it in the specified variable(s). Each piece of data on the chip must end with a RETURN character, colon (:) or comma. If you try to read past the end of file, the yellow error light on the PROMENADE will flash, but no error will be returnd to the program you are using.

GET#filenum, variable(s) - Gets data from the file into the specified variable(s). Only one byte will be put in each variable. Less convenient but more useful than INPUT#.

CLOSE filenum - Closes a file after all access is finished. Be sure to CLOSE every file after it is created!

Once again, these commands may seem very useful, but we've found that working with them can be quite a hassle. We recommend that you use a cartridge for storing programs.

ML MONITORS AND PROMOS

Although it is not absolutely necessary, a good ML monitor can come in very handy when programming EPROMs. Before burning a program onto EPROM, you'll usually have to make at least a few modifications, unless you are just copying another ROM. The EPROM projects in this book will require the use of a good ML monitor. You may already have a favorite monitor that you wish to use, or you may want to get the HESMON/PROMOS/WEDGE cartridge available from CSM. When choosing a monitor to use with PROMOS, look for one that lets you execute PROMOS commands without exiting the monitor. This is not an absolute necessity, just a convenience. Most monitors are compatible with PROMOS in this respect, including HESMON.

To use PROMOS and HESMON together, you should always start up HESMON first, then exit to BASIC with the "XC" command. Enable the cartridge version of PROMOS with SYS 9*4096 (or SYS 36864). If you just have the regular HESMON cartridge, load and run the disk version of PROMOS as usual. HESMON moves the end of BASIC down to $8000 (through the pointer at $37-38), so when PROMOS is run from disk it will locate itself before $8000 (at $76D0). Once PROMOS is enabled, you can reenter HESMON. DON'T use the RESTORE key, however, since this will "cold-start" HESMON and wipe out PROMOS's patches into the KERNAL. The best way to reenter HESMON is to execute a machine language BRK instruction. How do you do this from BASIC? Simple - just SYS to a location that contains a $00 byte (BRK instruction). We recommend using **SYS 8** for this on the C64 (it may not work on the C128 - try SYS 7168). When you do this you'll reenter HESMON with PROMOS still active.

All PROMOS and HESMON commands should function normally. One thing to be aware of, as pointed out in the PROMENADE manual, is that you'll get a "?" from HESMON anytime you enter a PROMOS command. In spite of this, the PROMOS command will execute correctly. Another thing to remember is that PROMOS will still require all its commands in DECIMAL, while HESMON commands will require HEX. Fortunately, HESMON has decimal/hex conversion functions! One more poit - if you exit HESMON with the "X" or "XC" commands, HESMON will restore the KERNAL vectors and so disable PROMOS. A better way to exit HESMON is with G A474. This pops you directly to the BASIC ready message, leaving PROMOS still hooked in. Remember - use SYS 8 to reenter HESMON and G A474 to exit it.

In case you don't already have a good monitor, we've included the public domain program MICROMON on the program disk with this book. Five versions of MICROMON are included, for maximum flexibility. They are identical except that each loads into a different area of memory. Since MICROMON is public domain, you are free to copy and distribute it as you wish. MICROMON was originally written by Bill Seiler in the days of the early PET computers, and there are many different versions of it floating around today. This particular version is the best we've seen. It was provided to us by Steven C. Schnedler of Schnedler Systems, 1501 N. Ivanhoe Dept R8, Arlington, VA 22305 (703-237-4796).

MICROMON is compatible with PROMOS, i.e., PROMOS commands can be executed from within MICROMON. To set this up, you should run PROMOS **before** you enter MICROMON. From then on you can enter and exit MICROMON normally without disturbing PROMOS (use "E" or "X" to exit). Most of the commands for MICROMON are the same "standard" commands used by many other monitors such as HESMON(tm) and HIMON (from PPM Vol. I & II). In addition, there are several commands in this version of MICROMON which are particularly useful when programming EPROMs.

A description of the MICROMON commands is given below. If you are already familiar with an ML monitor, you might want to skim over this material and note any extra commands and features. If you aren't familiar with an ML monitor yet, you should read the descriptions to get an idea of what a monitor can do. Don't try to memorize every command; you can refer back to this chapter as needed throughout the rest of the book.

MICROMON COMMANDS

NOTES:

1) All capital letters must be typed as shown. Lowercase letters stand for information to be supplied by the user (hh = any hex value).

2) All numbers must be entered in hex (with no $) except in the number conversion functions.

3) Brackets ([]) around an item mean that it is optional (don't type the brackets).

Starting up MICROMON

Five copies of MICROMON are included on the disk. Each loads into a different area of memory. The name of the file indicates where in memory the version resides. It also tells you the SYS address to use to start the monitor. For example, to use the version located at 49152 ($C000), first load it with:

```
LOAD "MICROMON 49152", 8, 1
```

To execute the monitor, use the command:

```
SYS 49152
```

This SYS command will cause a complete cold-start of MICROMON. If you've already entered and exited the monitor once (with a soft exit), you can reenter it with a slightly simpler command that is the same for all the versions. Like HESMON, MICROMON will be activated any time a BRK instruction is executed. You can force a BRK by using **SYS 8** on the C64.

Exit Commands

X - (SOFT EXIT) Exits MICROMON to BASIC, leaving the BRK and IRQ vectors as set by MICROMON. This allows you to 'hot-start' MICROMON by executing a BRK instruction from BASIC, using SYS 8 for instance. The "X" command will leave PROMOS enabled.

E - (HARD EXIT) Exits MICROMON to BASIC but restores the previous contents of the BRK and IRQ vectors (set by PROMOS). MICROMON must be cold-started to reenter it. This command will leave PROMOS enabled.

K - (KILL) Exits MICROMON to BASIC and restores the power-up values of the BRK and IRQ vectors. Same effect as RUN/STOP-RESTORE. This command will disable PROMOS.

G FCE2 - (RESET) Totally resets the computer.

File commands

L [alt. addr] "file" [,dev] - (LOAD) Loads a file from disk (no tape). If [,dev] is omitted, device 8 will be selected. If [alt. addr.] is specified, the file will be loaded at that address rather than its normal load address. This is very useful for EPROM work. After loading, the ending address of the program (+1) will be printed.

shift-L [alt. addr] "file" [,dev] - (Dummy LOAD) Same as L, but does not actually load the file. Used to find the ending address of a program (+1).

V "file" [,dev] - (VIEW) Prints the load address of the file, but does not load it.

S "file", start, end+1 [,dev] - (SAVE) Saves an area of memory to disk. Note that the end address you give must be 1 greater than the actual end.

Memory Commands

A addr instruction - (ASSEMBLE) Assembles one ML instruction into memory starting at the specified address. All addresses in the INSTRUCTION must be preceded by a $. Also automatically inserts an "A" on the following line to prepare for assembling another statement. Hit return or move to a blank line to end assembly.

D start [,end] - (DISASSEMBLE) Disassembles memory into hex and mnemonics. If only the start address is given, only one line of code will be displayed. If end is also specified, the entire range will be disassembled. In either case, the disassembly can be continued backward or forward in memory by going to the top or bottom (respectively) of the screen. Then use the up or down cursor key to scroll the screen. Illegal opcodes will disassemble as ???. The D command automatically puts a comma command (,) as the first character on the lines displayed. See the comma command (next).

, addr hex-code(s) - (MODIFY DISASSEMBLY) Allows a disassembled line to be modified. Changes must be made in hex only (use the "A" command to assemble mnemonics). Automatically redisplays line including new mnemonics.

M start [,end] - (MEMORY DISPLAY) Displays memory in hex and ASCII. Each line consists of 8 hex bytes followed by their ASCII equivalents (in reverse). Display can be continued backward or forward by scrolling screen as in the "D" command. A colon command (:) is automatically inserted as the first character of the line displayed. See the colon command (next).

: addr hh hh.. hh - (MEMORY MODIFY) Store 8 bytes in memory starting at address given.

R - (REGISTER DISPLAY) Displays the contents of the A,X,Y and status registers when MICROMON was entered. Also displays the stack pointer, program counter and IRQ vector. A semicolon (;) command is automatically inserted in the register display. See the semicolon command (next).

; pc irq status a x y stack - (REGISTER MODIFY) Allows you to edit the values saved for the registers, etc. These values are restored before a "G", "W" or "Q" command.

F start end hh - (FILL) Fills an area of memory with the hex value hh.

C startl endl start2 - (COMPARE) Compares two areas of memory. Any time corresponding locations in the two areas do not match, the areal address is printed. If areal is lower in memory than range2, the comparison will proceed backwards starting at the ends of the ranges. To avoid this, use the lower of the two areas for area2.

T source-start source-end dest-start - (TRANSFER) Copies the source area to the destination area. The source area is not altered (unless the areas overlap). Bytes are verifed as they are transferred and any mismatches are reported on the screen.

H start end hh hh... or **H start end 'string..** - (HUNT) Hunts through an area of memory for the hex or ASCII values given. A maximum of 32 bytes may be searched for. Do not use another ' at the end of an ASCII string.

Output Commands

P - (PRINT) Switches all output from the screen to the printer (device 4) or vice versa. You must hit return TWICE after the P before entering next command. Use P again to return output to screen.If the last line of printing is 'stuck' in the printer buffer (delayed printing), switching output to printer and back again should cause it to print.

OPEN filenum, dev [,s.a.] [,"string"] :CMD filenum - (General output) If the "P" command fails to work correctly with your printer, exit to BASIC and enter this line. Use the correct device number and s.a. (secondary address, if needed) for your printer. If a string is specified, it will be used as a title. Reenter MICROMON and perform the desired functions. When you're finished with the printer, exit to BASIC and enter **PRINT#filenum :CLOSE filenum**. This technique can also be used to store output in a disk file, by giving the appropriate device, etc. The string will be used as a file name. Be sure to open the file for writing by putting ",W" in file name after the file type (P, S or U).

Debugging Commands

G [addr] - (GO) Restores the values of the registers, etc. that were saved when MICROMON was entered (see R command) and then executes an ML routine. If [addr] is given, execution begins at that location. Otherwise, execution begins at the location given by the saved program counter. The ML routine must end with a BRK instruction in order to return to MICROMON.

W [addr] - (WALK) Restores the saved values of the registers, etc. and begins executing ML code one instruction at a time. Prior to executing an instruction (except the first), the following values are displayed: the status, A, X, and Y registers; the stack pointer; the program counter (address of next instruction); and finally, the hex and mnemonic for the next instruction. If [addr] is specified, execution begins at that location. Otherwise the saved value of the program counter is used. Pressing any key (except J and STOP) will execute the next instruction. Pressing the "J" key will execute a subroutine uninterrupted, until an RTS instruction is encountered. Executing a BRK instruction or hitting the STOP key will return to normal MICROMON command mode and save the current register, etc. values.

Q [addr] - (QUICK TRACE) Exactly like the "W" command except: execution is continuous; the instructions executed are not displayed; use STOP and "=" keys simultaneously to return to MICROMON; the "B" command can be used to set an optional "breakpoint".

B addr [hhhh] - (BREAKPOINT SET) Sets up a "breakpoint" for the "Q" command. When execution reaches the address given, a BRK instruction is executed to return to MICROMON. If [hhhh] is specified, the BRK is executed only after the given address has been executed that number of times.

Math Commands

N code-start code-end offset ref-begin ref-end [W] - (NEW LOCATION) Used to relocate a section of ML code - sort of like a RENUMBER command in BASIC. After moving the code with the "T" command, this command will adjust absolute memory references (not branch locations) to reflect the code's new location. The area specified by code-start and code-end will be searched for any (three-byte) instructions which reference the area specified by ref-start and ref-end. Any references found will be adjusted by adding the offset value. To calculate the proper offset, subtract the original start address from the new start address, using the "-" command. The [W] option specifies that a table of two-byte vectors (Words) is being searched rather than ML code. See the chapter on reusing the tape routines in the KERNAL for an example of using this command.

O instr-addr target-addr - (OFFSET) Used to calculate the offset for branch instructions. Instr-addr is the address of the branch instruction itself and target-addr where you want it to branch to. This command is pretty useless since MICROMON automatically calculates the offset for you when you assemble a branch instruction. Don't use this command for calculating offsets for the "N" command.

$ hhhh - (HEX CONVERSION) Converts the hex number hhhh into decimal, ASCII characters and binary.

 # ddddd - (DECIMAL CONVERSION) Converts the decimal number ddddd into hex, ASCII characters and binary.

% bbbbbbbb - (BINARY CONVERSION) Converts the binary number bbbbbbbb into hex, decimal, and ASCII characters.

" a - (ASCII CONVERSION) Converts the ASCII character a into hex, decimal and binary.

+ hhhh hhhh - (HEX ADDITION) Adds the two hex numbers and gives a one-byte hex result. If the result is over $FFFF, only the low byte is given.

- hhhh hhhh - (HEX SUBTRACTION) Subtracts the second hex number from the first.

& start end - (CHECKSUM) Checksums an area of memory by ADDing together all of the bytes. The result is limited to two hex bytes.

THE 6510 AND THE PLA

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 very similar 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 make up a VECTOR. 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 the RESET routine. We will cover the RESET routine that the C-64 uses later 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 6-1 we can see six pins of the microprocessor that are labeled P0, P1, P2, P3, P4, & P5. These pins correspond to bits 0 thru 5 of 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 levels) 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 1 and change the appropriate bit in location $0001 to a value of 1. 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 or $0001 it will actually be 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 6-1 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 AREAS** ($A000-$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 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 3 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. Thus a JMP instruction is accomplished 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 the 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).

(INTENTIONALLY LEFT BLANK)

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.

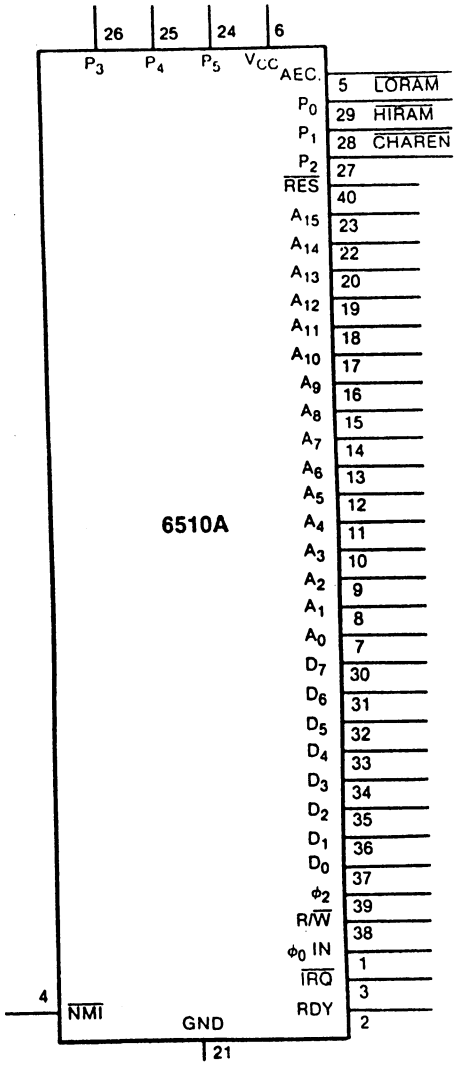


FIG 6-1

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. In other words, trying to turn off the BASIC ROM with LORAM when GAME and EXROM are both grounded **will** turn off the cartridge memory at $8000-9FFF but **will not** turn off cartridge memory 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 cannot 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 when 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 already covered all that we need to about the 6510 microprocessor and the PLA.

CARTRIDGES AND CARTRIDGE BOARDS

There are two main ways to use an EPROM in your Commodore computer system. You can use the EPROM on a plug-in cartridge board, or you can use it to directly replace one of the ROM chips in the computer or drive. We'll cover both of these topics, but let's concentrate on cartridges for now. There are several different kinds of cartridges for the C-64, including exotic cartridges used in some commercial products. To understand the differences among cartridges, we need to look at how cartridges are recognized and accessed by the computer. Before proceeding, be sure you have read the chapter on memory management. In that chapter we looked at the PLA and its relationship with the rest of the computer. In this chapter we'll look at how cartridges interact with the PLA.

The simplest type of cartridge is the 8K cartridge. Actually, you could put LESS than 8K of EPROM on this type of cartridge, but 8K is the maximum, so we'll loosely call it the standard 8K cartridge. A single 2764 EPROM (8K) is usually used in these cartridges (commercial cartridges may use PROMs instead). When this type of cartridge is plugged into the computer, the EPROM will be "seen" by the computer at memory address $8000-9FFF. The RAM which is normally there will "disappear". Of course, you'll get your RAM back when you unplug your cartridge. In fact, the contents of the RAM will be unchanged. (By the way, **NEVER** plug or unplug a cartridge when the computer power is on, unless you have a cartridge power switch. Doing so could destroy your computer and cartridge!)

The second type of cartridge can hold up to 16K of memory, so we'll call it the standard 16K cartridge. Two 2764 EPROMs usually supply the 16K. The first EPROM will appear in memory at $8000-9FFF, replacing the normal RAM there. The second EPROM will appear at $A000-BFFF, knocking out the BASIC ROM which is usually located there. This gives us 16K of continuous cartridge memory from $8000 to $BFFF. If you recall that the BASIC ROM is already "sitting" on top of 8K of RAM, you can appreciate how much is going on behind the scenes to keep all this straight.

The third type of cartridge is called an ULTIMAX or just MAX cartridge. ULTIMAX was a video game system produced by Commodore and sold only in Europe, and only for a short while. It used the same VIC II and SID chips as the C-64. The designers of the C-64 arranged it so that cartridges for the MAX would work on the C-64 too. On the C-64, MAX cartridges replace the KERNAL ROM located at $E000-FFFF with their own 8K of EPROM. MAX cartridges may also have another 8K of ROM memory if desired, which will appear at $8000-9FFF. One special feature of MAX cartridges is that all of the RAM of the computer disappears except for 4K at $0000-0FFF. Because we can't access most of the RAM, MAX cartridges really aren't useable in very many applications.

We just said that the EPROMs in these cartridges 'replace' different areas of memory. This is really only true for read operations. Write operations will vary in their effect depending on the type of cartridge. For example, with an 8K cartridge plugged in, the computer will be able to read the cartridge EPROM at $8000. If the computer tries to write to this location, however, the data will end up in the RAM "under" the cartridge. Likewise, with 16K cartridges write operations go to RAM automatically, even though the second EPROM at $A000-BFFF is two levels removed from RAM (the BASIC ROM is sandwiched in between the EPROM and RAM). With MAX cartridges, however, the RAM is truly "gone". Writing has NO effect on any area of RAM except the $0000-0FFF area.

How do the cartridge EPROMs and RAM chips know when to respond and when not to? Why does a read operation go to EPROM and a write operation go to RAM? We saw in a previous chapter that the C-64's PLA chip is in charge of memory management. In this chapter we'll see how the cartridge controls the PLA to produce these effects. At the same time, we'll address a related topic - what makes the three types of cartridges different? Why do the cartridge EPROMs appear at the locations they do?

The answers to these questions lie in the EPROM's **enable lines.** In order for a chip to be active, it must have a supply of power, first of all. It must also have **address and data lines** to communicate to the outside world. Most chips also have at least one **enable line.**Most of the EPROMS discussed in this book actually have two enable lines, called the **chip enable (CE)** and **output enable (OE)** lines. Both have to be controlled correctly in order to access the chip. An enable line is like a switch. The chip enable (CE) is a power switch. Even if power is available, the chip will not become active until the CE line is brought low (grounded; set to 0 volts). The abbreviation CE is usually written with a bar over it to indicate that the CE line performs its function only when it is brought low (this is called active low). When CE is held high (+5 volts), the chip is put into "standby" mode. In this mode the chip uses much less power than when active. A certain minimum amount of power is used in standby mode to keep the chip "warmed up". EPROMs don't require ANY power just to retain their data.

The other enable line, output enable (OE), controls the chip's data lines. OE is usually written with a bar over it too, since the chip will only put out data when the OE line is low. In order for the chip to be active, both OE and CE have to set low at the same time. On C-64 cartridges, the two enable lines from the chip are combined into a single line to make enabling the chip easier. The chip can be enabled by switching this one combined line high or low. Cartridges with two chips on board have a separate enable line for each chip. Each enable line is a combination of the OE and CE lines from one chip. From now on when we speak of THE enable line for a chip, we'll be referring to the combination of CE and OE.

Enable lines are used when several chips are connected to the same set of address and data lines. If more than one chip were active at the same time, there would be mass confusion (bus conflict) and possibly even physical damage to the chips. By controlling the enable lines, you can make sure only one chip at a time will be active on the address and data lines. Does this situation sound familiar? Of course - it's exactly the situation we have in the C-64 when we plug in a cartridge, since we could have EPROM, ROM and RAM potentially residing at the same address! Now we see that there is really a simple principal underlying the complexity of memory management. The PLA chip, in its infinite wisdom, knows which chip enable line to switch on, depending on what type of cartridge (if any) is plugged in and whether the operation is a read or a write. Remember, write operations are usually directed to RAM, except when writing to the I/O devices at $D000-DFFF. Read operations have to be sorted out and directed to the proper chip (EPROM, ROM or RAM).

The PLA has many lines coming into it (inputs) that it uses to sense the present state of the computer. It also has several lines coming out of it (outputs) that are used to control the memory chips. The PLA monitors its input lines continuously. Any changes in the input lines affect the output lines immediately. One input line called R/W is used to distinguish between read and write operations. Write operations are "easily" handled since they almost always go to RAM, so we'll concentrate on read operations. Of all the PLA's lines, we only need to be concerned about four right now: two input lines, GAME and EXROM; and two output lines, ROML and ROMH. The function of the GAME and EXROM lines is affected by other input lines, such as HIRAM and LORAM, but for this discussion we'll assume that HIRAM and LORAM are both held in their normal state (high).

GAME and EXROM are inputs to the PLA from the cartridge port. They are not connected to anything else in the C-64. GAME is pin #8 on the cartridge port. On a cartridge BOARD, this is the 8th pin from the left on the top side of the board (the component side, where the EPROMs are mounted). See the diagram in appendix D. EXROM is pin #9, right next to GAME. Both are active low, that is, when grounded (0 volts). With no cartridge plugged in, these lines are automatically held high (+5 volts). It's up to the cartridge to ground these lines or not, according to the memory configuration it wants the PLA to set up. In a nutshell, this is how the PLA knows what type of cartridge is connected. If EXROM alone is grounded, it indicates an 8K cartridge (regardless of how many EPROMs are actually on the board, as we'll see). If both EXROM and GAME are grounded, it indicates a 16K cartridge. Finally, if just the GAME line is grounded, it indicates a MAX cartridge (either 8K or 16K). Grounding a line is as simple as it sounds - just connect it to the computer's ground. Pins 1, 22, A and Z on the cartridge port are all grounds.

Okay, so GAME and EXROM tell the PLA what's going on. What does the PLA do about it? This brings us to the PLA output lines ROML (ROM Low) and ROMH (ROM High). ROML and ROMH are connected only to the cartridge port. They are both normally held high (+5v) unless a cartridge grounds EXROM or GAME (or both). Depending on which of these lines are grounded, the PLA will bring ROML and/or ROMH low too. What are ROMH and ROML? Nothing more than two EPROM enable lines! ROML is the combined OE/CE enable line for the first cartridge EPROM, located at $8000-9FFF in memory. With a cartridge (any type) plugged in, the PLA will bring ROML low any time a read operation tries to access the $8000-9FFF area. Remember, bringing an enable line low will activate the chip. ROML is **not** held low **all** the time, since then the chip would be active even when other areas of memory were being accessed (resulting in bus conflict). ROML is only held low momentarily until the read operation can be performed. This is how the PLA 'tells' the EPROM it is located at $8000 - the PLA only enables the EPROM when that area of memory is being accessed.

The ROMH enable line is just a little more complicated because the second EPROM appears at different locations in memory with different types of cartridges. With a standard 8K cartridge (EXROM grounded), the second EPROM is not used and so it's disabled by holding ROMH high at all times. With a standard 16K cartridge (both GAME and EXROM grounded), any read operations in the **$A000-BFFF** area will signal the PLA to bring ROMH low. Since ROMH is connected to the second EPROM's enable lines, this will make the EPROM appear at $A000 in memory. With a MAX cartridge (GAME grounded), read operations in the **$E000-FFFF** area will enable the second EPROM through ROMH and make it appear at $E000. Note that with a MAX cartridge, you MUST have an EPROM at $E000-FFFF since the microprocessor automatically looks there on reset. The $8000-9FFF EPROM is optional with MAX cartridges (and in fact, rarely if ever used).

At this point a little review is in order. The GAME and EXROM lines run from the cartridge to the PLA. They tell the PLA what type of memory configuration to set up. Based on the memory configuration, the PLA enables the cartridge EPROMs at the proper times using ROML and ROMH. Cartridge EPROMs are only enabled for read operations, never write operations. A cartridge EPROM is only enabled when its particular area of memory is referenced. The PLA controls which area of memory is assigned to the cartridge EPROMs, depending on the state of the GAME and EXROM lines. The PLA monitors the GAME and EXROM lines continuously.

The cartridge types we've examined so far by no means exhaust the possibilities. By manipulating the GAME, EXROM and other lines, many exotic cartridges can be created. The most common example of an exotic cartridge is a **bank-switch** cartridge. "Bank-switching" means turning memory chips on and off, so different chips can occupy the same addresses at different times. Sound familiar? The C-64 already uses bank-switching, controlled by the PLA, to select its different memory configurations. What's different about bank-switched cartridges is that the bank-switching is done on-board the cartridge itself, in addition to the memory selection done by the PLA.

A bank-switched cartridge looks like a standard 8K cartridge to the C-64. The cartridge will ground only the EXROM line, so the PLA thinks the cartridge contains 8K of memory at $8000-9FFF. This is accurate as far as it goes: only 8K of cartridge memory will be available at a time, and it will appear at $8000. However, the cartridge board may contain any number of EPROMs. Special circuitry on-board the cartridge picks out one EPROM at a time to appear at $8000. Accessing this memory is a two-stage process: the PLA brings the ROML enable line low and then the cartridge bank-switch circuitry passes this enable signal to its currently selected EPROM. The C-64 doesn't know about the second stage, of course; it just sees a standard 8K cartridge. The only time the C-64 has to do anything special is when it wants the board to change the current EPROM.

To tell the board to change the current EPROM, we have to send a special signal to the board. We can't use the regular address, data or enable lines for this, however. Instead, most bank-switch cartridges use a special line, not normally used for anything else on the C-64 (except the Z80 CP/M cartridge). There are actually two of these lines to choose from, called **I/O1** and **I/O2**. I/O1 is set low whenever a read OR write operation accesses the $DE00-DEFF area (note that the ROML and ROMH lines are set low only on read operations). I/O2 is similiar except it's triggered by references to the $DF00-DFFF area. I/O1 and I/O2 are pins 7 and 10 on the cartridge port, respectively. On a bank-switch board, one of these lines will be connected to a special piece of circuitry called the **bank-select register (BSR)**. Depending on whether I/O1 or I/O2 is used, the BSR will appear at $DE00 or $DF00 in memory. To switch the current EPROM all you have to do is trigger the BSR, usually by writing a particular value to it. That's all there is to it. Once you trigger the BSR, the EPROM selected will appear at $8000 immediately.

Bank-switch cartridges are especially useful for programs which are too large to fit in 16K (the maximum for a regular-type cartridge). Bank-switching can also provide considerable protection for a program, depending on how it is used. There are two main ways a cartridge can use bank-switching. The simplest way is to just download the program from the EPROMs into RAM memory, one 8K chunk at a time. After downloading, many cartridges can remove themselves from memory by "ungrounding" the EXROM line via special circuitry. This frees up the RAM at $8000 (under the cartridge) for use by the program. The simple download method is relatively easy to set up but doesn't offer much protection for the program. A second way to use bank-switching is to have different sections of the program on different EPROMs. Depending on which part of the code is needed, the board can select the proper EPROM. The code is never downloaded into RAM, but rather executed directly on the EPROM. This is much more complicated for the programmer to set up, but it is also a very solid program protection technique. To make a RAM executable copy from such a cartridge, if it could be done at all, the code would have to be modified extensively.

More and more commercial cartridges now use bank-switching. For example, MAGIC DESK (tm) from Commodore contains four 8K EPROMs that are simply downloaded into RAM (see the Dec. 1984 issue of the PROGRAM PROTECTION NEWSLETTER). The cartridge is still called upon to supply sprite data from time to time, so a little modification is required to make a totally RAM-based version. Another example of a commercial bank-switch cartridge is the COMAL 2.0 cartridge (COMAL is a programming language). The cartridge contains 64K of program code on EPROM, plus an empty EPROM socket for up to 32K of your own COMAL programs! Obviously, there is no way to fit all 96K of this into RAM at one time.

One final example is the FASTLOAD (tm) cartridge from EPYX, which uses bank-switching in a novel way. Its bank-select register (BSR) is located in the $DE00-DEFF area, through the **I/O1** line. The BSR gives the cartridge the ability to appear at different locations in memory or disappear from memory entirely. Amazingly, part of the EPROM can actually be read at $DF00-DFFF! This bit of magic is done by using the **I/O2** line to enable the same EPROM that normally resides at $8000. When I/O2 is brought low by reading from the $DF00-DFFF area, the EPROM will be enabled. This makes it appear in that area. Only part of the EPROM will be seen there - just the code normally found at $9F00-9FFF. This section of code contains a whole set of routines which are patched into the KERNAL.

If you want to get into bank-switch cartridges yourself, CSM carries a couple of models. The PC4 model has four sockets, each of which can hold a 2764 (8K), 27128 (16K) or 27256 (32K) EPROM. Different EPROM types can be used together on the same board. By using four 27256's, you can get 128K on a board the same size as a standard cartridge! The cartridge memory is accessed in 8K sections. By changing a few bits of the bank-select register (BSR), you determine which 8K section will appear at $8000. The BSR can be located at $DE00 or $DF00 as desired. The board also has the ability to remove itself from memory completely. The PC4 comes with software to download the whole board to RAM, automatically checking EPROM types as it goes. CSM also carries a PC8 board with 8 sockets, which allows up to 256K on the board! All of the other features of the PC4 are included. These are the boards to use if you want to put large programs on EPROM for personal or commercial use. Later on in the book we'll present an EPROM programming project using the PC4 board.

We'll close this chapter with a subject related to cartridges, namely cartridge expansion boards. These are peripherals that plug into the cartridge port and have a slot or slots for you to mount cartridges in. Expansion boards let you insert and remove cartridges without turning the computer off. This saves wear and tear on your computer and its cartridge port. Multi-slot boards allow you to select from several cartridges with a flick of a switch or two. Expansion boards are not only convenient, but they also let you do things you couldn't do without one. They can even be useful for investigating programs which aren't cartridge-based. If you work with cartridges much, a good expansion board is absolutely essential.

Beware! - all expansion boards are not created equal. Most types of expansion boards only let you turn the cartridge on and off. There are two particular types of expansion boards that are significantly better than most. CARDCO's Five Slot Board and CSM's Single Slot Board give you control over several important cartridge lines. Both of these expansion boards let you individually control the EXROM, GAME, enable and power lines. This capability is very useful. For instance, many non-cartridge programs use the cartridge autostart feature by putting a "CBM80" in RAM at $8004. If you reset the computer, this autostart feature will take control and prevent you from examining the program. You can defeat this if your expansion board has an EXROM switch. By grounding EXROM, the PLA will think there is a cartridge at $8000, and it won't let the computer "see" the autostart code in RAM. This happens even if there is no cartridge plugged in, as long as EXROM is grounded.

Both of the expansion boards mentioned above have a reset button built-in. They also have LEDs (lights) on the board to tell what type of cartridge you have inserted. If the rightmost LED is lit, it indicates that the cartridge grounds the EXROM line. This means it's a standard 8K cartridge (or bank-switch type). If both lights are lit, both EXROM and GAME are being grounded, which indicates a standard 16K cartridge. Finally, if just the left LED is lit, then only GAME is grounded, which indicates a MAX cartridge (either 8K or 16K). Not only do these LEDs help when investigating a commercial cartridge, they also help you make sure you have set GAME and EXROM correctly on your own cartridges.

AUTOSTART CARTRIDGES

Suppose you wish to set up a cartridge that runs automatically when the computer is powered up or RESET. To do this you'll have to interrupt the normal power-up (RESET) process somehow, and force the computer to execute your cartridge program. Depending on where you interrupt the normal process, however, your program may have to initialize some areas of the computer for proper operation. For instance, it may have to initialize the 6510, VIC or CIA chips, or the KERNAL or BASIC RAM areas. Thus it's important to know what initialization is done normally, when it's done and why. In this chapter we'll examine the various ways to interrupt the RESET process and autostart a cartridge, including the necessary initialization tasks.

There are three main methods you can use to autostart a cartridge. Each method involves interrupting the RESET process at a different point, and each method is best suited to a different kind of cartridge. We'll start with the easiest method first, which we call the $A000 method (you'll see why in a minute). The RESET process can be divided into two phases, KERNAL initialization and BASIC initialization. BASIC initialization is only necessary if your cartridge must be compatible with the BASIC system. For example, if your cartridge adds commands to the BASIC language or uses certain BASIC ROM subroutines, you'll need to initialize BASIC.

$A000 METHOD

If you don't need the BASIC system, you can "trick" the KERNAL RESET process into doing all your initialization for you and then autostarting your cartridge. After completing its own initialization tasks, the KERNAL RESET routine attempts to "cold-start" (initialize) BASIC. It does this by jumping to the location specified by the **BASIC COLD-START VECTOR.** Like all vectors, the BASIC cold-start vector consists of two consecutive bytes containing a memory address. The address is stored in lo-byte / hi-byte order, which means the low order (least significant) byte is first and the high order (most significant) byte is second. The KERNAL expects the BASIC cold-start vector to be found in locations $A000-01, which is normally at the very beginning of the BASIC ROM. The contents of these two locations in the BASIC ROM are $94 and $E3 respectively, which means they "point" to location $E394 (vectors are also called pointers). This location is the start of the BASIC cold-start routine.

If we could change the BASIC cold-start vector, we could make it point to our cartridge program. Our cartridge would start up automatically after all KERNAL initialization was finished. But since this vector is in the BASIC ROM, how do we change it? Answer: replace the BASIC ROM. Not physically, of course, but by using a standard 16K cartridge. Recall that standard 16K cartridges reside at $8000-BFFF. The PLA switches out the BASIC

ROM and selects the 16K cartridge configuration when it senses that both of the GAME and EXROM lines are grounded. Two 8K EPROMS are required for 16K of memory on the cartridge. The first EPROM resides at $8000-9FFF and the second EPROM resides at $A000-BFFF. All you have to do is put a vector at $A000-01 which points to the beginning of your program, and the cartridge will be started automatically at the end of KERNAL initialization.

If you only need 8K for your cartridge, you can still use this technique. Just use the second EPROM ($A000-BFFF) and leave the first EPROM ($8000-9FFF) socket empty. As long as GAME and EXROM are grounded, the PLA will still choose the 16K configuration. This means the RAM normally at $8000-9FFF will still be switched out, however (and the BASIC ROM too, of course). Since you're not using the first cartridge EPROM, you'll have a "hole" in memory from $8000-9FFF. If you try to read from this area, random data may appear there. You may even be able to use this phenomena as part of a protection scheme. As usual, data written to this area will be placed in the underlying RAM, even though you can't read it back out.

So, to review a bit, the $A000 method is the easiest autostart technique to use because all KERNAL initialization is done for you. You don't have to worry about BASIC initialization since you can't use BASIC anyway (the BASIC ROM is switched out). Of course, you don't have access to the BASIC ROM subroutines either (there's a lot of useful stuff in there). This makes the $A000 method most suitable for cases where you need the maximum 16K of cartridge memory, or where you only need 8K and don't need BASIC.

CBM80 METHOD

The second cartridge autostart method is by far the most common. It can be used by both 8K and 16K cartridges that reside at $8000. Although this method is sometimes called the "cartridge autostart option", it can be used equally well by RAM-based programs, and often is. You probably know it as the "CBM80" method. One of the first things the KERNAL RESET routine does is check locations $8004-08 for the string of characters "CBM80". If these exact characters are **NOT** found there, the KERNAL RESET process continues normally.

If the "CBM80" **IS** found, the RESET routine is interrupted and the processor immediately jumps to whatever location is specified by the **CARTRIDGE COLD-START VECTOR**. This vector is expected to be found at locations $8000-01. You must place a pointer here, in standard lo-byte / hi-byte order, directing the processor to the beginning of your cartridge code. From that point on, your cartridge must handle all the initialization itself for any functions it will use, such as the I/O devices or KERNAL or BASIC routines. Fortunately, you still have the KERNAL initialization routines available for use. Unless you know exactly what you are doing, your cartridge should use these routines to initialize the functions it needs.

Figure 8-1 presents a "generic" cartridge initialization routine. This routine duplicates most of the normal RESET process. In fact, it's taken right from the main parts of the KERNAL ($FCEF-FE) and BASIC ($E394-9F) RESET routines. This generic routine will be adequate for 99% of all cartridges.

Figure 8-1: CBM80 Cartridge Initialization

```
8000  09 80                    Cartridge cold-start vector = $8009
8002  25 80                      "   "   " warm   "   "   "   =  8025
8004  C3 C2 CD 38 30           CBM80 - Autostart key

                               KERNAL RESET Routine
8009  8E 16 D0   STX $D016     Turn on VIC for PAL/NTSC check
800C  20 A3 FD   JSR $FDA3     IOINIT - Init CIA chips
800F  20 50 FD   JSR $FD50     RAMTAS - Clear/test system RAM
8012  20 15 FD   JSR $FD15     RESTOR - Init KERNAL RAM vectors
8015  20 5B FF   JSR $FF5B     CINT   - Init VIC and screen editor
8018  58         CLI           Re-enable IRQ interrupts

                               BASIC RESET  Routine
8019  20 53 E4   JSR $E453     Init BASIC RAM vectors
801C  20 BF E3   JSR $E3BF     Main BASIC RAM init routine
801F  20 22 E4   JSR $E422     Power-up message / NEW command
8022  A2 FB      LDX #$FB
8024  9A         TXS           Reduce stack pointer for BASIC

8025  ........   .........     START YOUR PROGRAM HERE
```

The cartridge cold-start vector and autostart key (CBM80) have already been discussed. The warm-start vector at $8002-03 is a feature that allows you to re-enter your program after a full initialization has already been done. Once a cold-start has been done, it usually doesn't need to be done again. Pressing the RESTORE key calls the NMI routine (NON-MASKABLE INTERRUPT), which will see the CBM80 and jump to the location indicated bt the warm-start vector. This is why many programs restart themselves when you press the RESTORE key. In our initialization routine we have pointed the warm-start vector to the start of your program; you could also point it to $8009 to perform a full cold-start on RESTORE. If you want to disable the RESTORE key entirely, point the warm-start vector to $FEBC (return from NMI).

We have included the BASIC RESET process in this cartridge initialization routine too. Actually, the normal BASIC RESET routine dead-ends with a jump to the BASIC direct mode interpreter, also known as "READY" mode. This prints the "READY." prompt and then sits there waiting for you to type a BASIC command. You won't usually want to exit into READY mode at this point since BASIC will take over and your cartridge will lose control. If you do want to exit to BASIC now or later, you may do so with JMP $E386. By the way, the routine called at $801F (JSR $E422) prints the normal power-up screen and does a NEW command. If you want to skip the power-up message, just call the NEW command directly using JSR $A644 instead of JSR $E422.

To summarize, the CBM80 method can be used with either 8K or 16K standard cartridges (which start at $8000). The cartridge initialization routine above will be sufficient for the vast majority of cartridges. KERNAL initialization must be done at least once (on power-up or RESET). BASIC initialization can be skipped if you're not using BASIC, and **MUST** be skipped if you're using a 16K cartridge. Through the cartridge warm-start vector, the RESTORE key can be set up to re-enter your program or it can be disabled entirely. The CBM80 method is by far the most common cartridge autostart method.

MAX METHOD

The third and last autostart method is the MAX method. This requires the use of an ULTIMAX cartridge (one that grounds just the GAME line). The second EPROM in a MAX cartridge resides at $E000-FFFF, replacing the KERNAL ROM (the first EPROM appears at $8000-9FFF if used). MAX cartridges have many limitations and are rarely used commercially except for simple video games. One limitation of MAX cartridges is that only 4K of the computer's RAM is available for use, and one-quarter of that is required for screen memory. Another limitation is that the KERNAL ROM is switched off, which means you must write your own powerup/RESET initialization routines (although you may use the KERNAL routines as a guide). You don't need to worry about BASIC initialization since the BASIC ROM is switched out too. For these reasons, you'll probably never need to set up a MAX cartridge. Only an advanced user would want to consider using this method.

MAX cartridges autostart through a hardware function of the 6510 processor rather than through a software routine in the KERNAL like the other two methods. When the C-64 is powered up, a special circuit RESETs the 6510 microprocessor, SID and CIA chips, just as if you had performed a hard RESET yourself (with a RESET button). Two very important events take place immediately, before any instructions are even executed. First, the I/O devices (VIC, SID, CIAs and color RAM) are switched into memory by the PLA, as well as the KERNAL and BASIC ROMs (normally). Second, the 6510 processor fetches its **RESET vector** from locations **$FFFC-FD**, normally in the KERNAL ROM. The RESET vector is a two-byte value that points to the beginning of the KERNAL RESET routine. The processor **ALWAYS** gets its RESET vector from locations $FFFC-FD. This feature is "hard-wired" into the 6510 chip itself, and **CANNOT** be changed! You must make sure the processor finds a valid address in locations $FFFC-FD, which is why the KERNAL is normally switched in first on RESET. The processor does an indirect jump based on the RESET vector and normally begins executing the KERNAL RESET routine.

The only alternative to executing the KERNAL RESET routine is if a MAX cartridge is present. If the PLA senses that a MAX cartridge is plugged in (GAME line grounded), it switches in the cartridge at $E000-FFFF, replacing the KERNAL. Then when the 6510 fetches its RESET vector from $FFFC-FD, it will get it from the cartridge instead of the KERNAL. You simply place a vector at $FFFC-FD pointing to the beginning of your RESET routine, and the processor will start executing your routine automatically. This is how the MAX autostart method works - your cartridge is switched in and grabs control right from the start.

Now it's up to your cartridge to perform all necessary initialization. For instance, the VIC chip must be initialized in order to use the screen. Likewise, other I/O devices will have to be initialized if you want to use the keyboard, the joysticks, the sound chip, IRQ interrupts, etc. Initializing and controlling these devices can be quite complicated. We recommend that you use the corresponding KERNAL routines as a model for your own routines. In fact, many MAX cartridges contain almost byte-for-byte copies of KERNAL routines. While we can't cover the KERNAL routines in depth in this book, we can summarize the various initialization routines used in the normal RESET process. This will help get you started towards an understanding of what initialization your cartridge will require.

NORMAL RESET PROCESS

Recall that when the 6510 processor is RESET, it begins executing at the address specified by the RESET vector at $FFFC-FD. In the KERNAL ROM, these locations contain a pointer to the KERNAL RESET routine at $FCE2 (decimal 64738). The main part of the KERNAL RESET routine is shown in figure 8-2. Note the similarities to our CBM80 cartridge initialization routine (fig. 8-1). The KERNAL RESET routine takes three important steps right away. First, IRQ interrupts are disabled with a SEI instruction, so the routine won't be interrupted. Second, the stack pointer is initialized by transferring a value to it from the X-register, using a TXS instruction. The stack pointer indicates the next empty position on the stack, which grows DOWNWARD in memory from $01FF to $0100 (i.e. the pointer decreases as the stack is filled). The stack pointer contains a random value on RESET, so we should put a value here before any stack operations take place (i.e. a PHA, PLA, PHP, PLP, JSR, RTS, RTI instruction or an NMI, IRQ or BRK interrupt). The RESET routine sets the stack pointer to $FF, which starts the stack out at the very top (allowing it the maximum space).

The third step is to clear the decimal mode flag with a CLD instruction. This flag controls whether math instructions like ADC (add) and SBC (subtract) are performed in normal "hex" format (actually binary) or in BCD format (BINARY CODED DECIMAL). Like the stack pointer, this flag has a random value on power-up, so it is set to 0 to ensure that math is done in hex format. If you write your own initialization routine for a MAX cartridge you should also do these three things right away.

Figure 8-2: KERNAL RESET ROUTINE

```
FCE2  A2 FF     LDX #$FF     Stack pointer value
FCE4  78        SEI          Disable IRQ interrupts
FCE5  9A        TXS          Initialize stack pointer
FCE6  D8        CLD          Clear decimal mode
FCE7  20 02 FD  JSR $FD02    Check for CBM80 key
FCEA  D0 03     BNE $FCEF    Branch if not found
FCEC  6C 00 80  JMP ($8000)  Jump to cartridge cold-start
FCEF  8E 16 D0  STA $D016    Turn on VIC (Aç=$05)
FCF2  20 A3 FD  JSR $FDA3    IOINIT - Init CIA chips
FCF5  20 50 FD  JSR $FD50    RAMTAS - Clear/test system RAM
FCF8  20 15 FD  JSR $FD15    RESTOR - Init KERNAL RAM vectors
FCFB  20 5B FF  JSR $FF5B    CINT - Init VIC and screen editor
FCFE  58        CLI          Re-enable IRQ interrupts
FCFF  6c 00 A0  JMP ($A000)  Jump to BASIC cold-start ($E394)
```

After the first three steps, the RESET routine calls an important subroutine at $FD02. This routine checks locations $8004-08 for "CBM80" autostart key. If these exact characters are found there, the cartridge cold-start vector is fetched from $8000-01. Execution continues at whatever location is indicated by this vector. This is the point at which the CBM80 autostart method takes control.

If there is no "CBM80" found, the RESET process continues at $FCEF. The X register is stored into location $D016, which is the VIC control register. The value of X at this point is always $05 or less, since it was used as an index in the check for "CBM80" (which has 5 characters). Commodore says it's extremely important to make sure bit number 5 of this value is a 0, which it is in this case. A 0 in bit 5 supposedly turns the VIC chip on and a 1 turns it off (see p. 322 and p. 448 in the Prog. Ref. Guide). For safety's sake your own initialization routine should set this bit to 0 too. There's an interesting side-effect when values less than $05 are stored in this register. In those cases, bit 3 will also be a 0, which selects 38-column mode. That's why the screen "shrinks" when the computer goes through its normal RESET process - bet you always wondered about that!

Next, the four main KERNAL initialization routines are called. These are the same routines we called in our CBM80 initialization routine. The first routine is IOINIT, located at $FDA3. This routine can also be reached by jumping to $FF84 in the standard KERNAL jump table. IOINIT initializes the CIA chips. It also does some other minor initialization such as turning off the SID's sound, switching in the BASIC and KERNAL ROMs (redundant on a hard RESET) and sending a high clock signal ("1" bit) on the serial bus. Next, the KERNAL routine RAMTAS at $FD50 is called ($FF87 in the jump table). This routine clears and tests RAM. First, the routine fills locations $0002-0101, $0200-02FF and $0300-$03FF (pages 0, 2 & 3) with $00 bytes. This piece of code is responsible for the cassette buffer, etc. being cleared on RESET. Note that the stack is not cleared (except the bottom two bytes).

After this, RAMTAS sets the cassette buffer pointer and then begins testing RAM memory starting at $0400 (the screen). The purpose of this test is to find the start of non-RAM memory, i.e., to see if there is cartridge ROM at $8000. The test is supposed to be "non-destructive" in that RAM memory is not altered. First, the current contents of the location to be tested are saved in the X-register. A $55 byte is stored into the location and then the location is read back to see if the $55 was stored successfully. If the location now contains a $55 then it is obviously in RAM - or is it? What if the location is in ROM but happened to already contain a $55? To doublecheck this, the process is repeated with the value $AA instead of $55. If it passes both tests, the location is definitely in RAM. The original value saved in X is restored, and the test continues with the next byte.

Eventually, the routine will run into ROM (either cartridge ROM at $8000 or BASIC ROM at $A000). When ROM is encountered, a very undesirable side-effect occurs. The $55 byte that is written out goes into the RAM "under" the ROM, wiping out the value that was there. This is important to remember when you are trying to recover a crashed program by resetting it. Once the start of ROM is found, a routine at $FE25 (MEMTOP, jump table $FF99) is called to set the top of system RAM to the beginning of ROM. The top of system RAM is used in the calculation of the number of BASIC bytes free. Finally, the bottom of system RAM is set to $0800, and the start of the screen is set to $0400 for the screen editor.

Back in the main RESET routine, a routine at $FD15 is called (RESTOR, jump table $FF8A). This routine copies the KERNAL's indirect RAM vectors to $0314-33 from the table at $FD30-4F. Oddly enough, it also copies this vector table into the RAM under the KERNAL at $FD30-4F too! If you are trying to recover the contents of the RAM under the KERNAL after a RESET, you should remember this feature.

The final KERNAL initialization routine is at $FF5B (CINT, jump table $FF81). This routine initializes the VIC chip and screen editor variables. The VIC chip is initialized by calling a routine at $E5A0 which downloads a set of constants to the VIC from $ECB9-E6. This sets the border and background colors as well as the raster interrupt register, used in the PAL/NTSC check discussed below. The screen editor is initialized by a routine at $E518. This sets the character color, keyboard decode table vector, cursor blink and key repeat rates, and then clears the screen. The CINT routine ends with the PAL/NTSC check. NTSC is the North American TV standard and PAL is the International TV standard. There are more lines on the screen with PAL. This fact is used to detect which system you have, so the IRQ and RS-232 timing can be adjusted accordingly. The VIC raster (screen line) interrupt was set earlier to occur on a line which doesn't exist with NTSC. Later (at $FF63) the interrupt is checked to see if it happened. If it did, we're on a PAL system, otherwise it's NTSC. See the Prog. Ref. Guide pp. 150 & 447 for more information on the raster interrupt register.

That's about it for the KERNAL RESET routine. IRQs were disabled earlier, so they are re-enabled with a CLI instruction. Finally, BASIC initialization is begun by jumping based on the BASIC cold-start vector in the BASIC ROM at $A000-01. If a 16K standard cartridge is present, however, its second EPROM will have replaced the BASIC ROM in memory. In this case we must put a vector at $A000 (in the second cartridge EPROM) to point to the start of the cartridge program. This is the basis for the $A000 autostart method. Notice how all the KERNAL initialization has already been done for us in this case.

For you MAX cartridge users, this brief outline should be a guide to examining the RESET process yourself. A reference book such as ANATOMY OF THE COMMODORE 64 will be an invaluable aid. There is no substitute for studying the RESET process yourself. We can only give you a few guidelines about what you should and should not do in your MAX cartridge. The only things you HAVE to do are set the stack pointer to some value (usually $FF) and either clear or set decimal mode (usually clear). You'll almost certainly want to use the screen, so you'll need to set up the VIC chip. Look at the table of constants at $ECB9 to see what values are put into the VIC's registers normally. Also remember to turn the VIC on as done at $FCEF-F1, and select its memory bank as done at $FDCB-CF. If you want to do a PAL/NTSC check, the code at $FF5E-6A can be your guide. If you want to set up an IRQ interrupt, study the IOINIT routine and the code at $FF6E-7C. Disk, tape or RS232 communications will require enormous amounts of code. If you think you need any of these, you probably shouldn't be using a MAX cartridge anyway.

We can also point some initialization routines you WON'T need. The CBM80 check routine at $FD02 is not important since your MAX cartridge has already been autostarted. You won't need the RAMTAS routine ($FD50) which clears memory and tests for the end of RAM / start of ROM. Your program can easily clear memory itself, and the end of RAM test would always yield the same results (the only RAM is at $0000-0FFF). The RESTOR routine (FD15) is used to set up RAM vectors for KERNAL routines which aren't available anyway, so it can be dispensed with. In short, only the IOINIT and CINT routines (and related routines) contain useful MAX initialization code. A lot of this code is superfluous too. The best idea is to plan out which functions you'll need and then study how the KERNAL sets up just those functions.

To round out this chapter, we should look at one other subject of general interest, WARM-STARTs (via the RESTORE key). We've already covered the cartridge warm-start vector at $8002-03, which only applies to the CBM80 autostart method. There is one other warm-start method. If the STOP key is held down along with RESTORE, and there is no CBM80, the BASIC warm-start vector will be used. This vector is located at $A002-3, normally in the BASIC ROM. If we have a 16K cartridge ($A000 method), which replaces the BASIC ROM, we can put our own warm-start vector in at $A002. In fact, you should always put a valid vector there to guard against the user accidentally pressing the RUN/STOP-RESTORE combination.

Finally, there is no warm-start method for MAX cartridges unless you program it yourself. You may choose to use the RESTORE key for this or some other method. If you don't use the RESTORE key, you should set the 6510 NMI vector at $FFFA-FB to point to an RTI instruction in case the user accidentally presses RESTORE. The 6510 is hard-wired to use $FFFA-FB as its vector when an NMI is generated (RESTORE key pressed), just as it always uses $FFFC-FD for its RESET vector.

In other chapters of the book we present some utilities you can use to create your own cartridges. Refer back to this chapter if you have questions about how those utilities work or if you want to modify them.

PROTECTING CARTRIDGES

In one sense, putting a program on cartridge is a form of protection all by itself, at least as far as the average user is concerned. Almost everyone has a good selection of utilities to copy disks, but there are very few utilities available to copy cartridges (CARTRIDGE BACKER comes to mind). Since most cartridges autostart themselves, there is no easy opportunity to examine the cartridge memory. As far as "professional" pirates are concerned, however, ripping off an unprotected cartridge is like taking candy from a baby. So cartridge manufacturers have been forced to use some type of protection, just as manufacturers of disk-based software were. Most commercial cartridges are now protected in one way or another.

Cartridge protection methods can be classified into two general categories based on whether the protection scheme requires extra hardware in the cartridge or not. Extra hardware in the cartridge can provide very effective protection but it is more expensive to design and produce. The program has to be designed around the protection scheme in order for the protection to be effective. In this chapter we'll focus on what can be done using just standard cartridge boards, since these techniques are the easiest for average programmers to apply to their cartridges.

One limitation of using only standard hardware is that you can't protect against the pirate who simply duplicates the EPROM and sticks it in a standard board. However, making a duplicate of a cartridge this way can cost the pirate almost as much money as the original cartridge would. What most pirates really want is a copy of the cartridge that can be loaded from disk and run in RAM. Therefore our goal is to prevent the program from running in RAM (or else detect the fact that it was loaded from disk).

To prevent the program from running in RAM, we must use some method that can distinguish whether the program is running in RAM or ROM (EPROM). The simplest and most common technique depends on the fundamental difference between RAM and ROM: RAM can be altered by writing to it, but ROM cannot. By storing a value into the area where the cartridge resides, a RAM copy of the program can be made to crash, while a ROM copy will not be affected. This is almost absurdly easy to put into practice, as the following example illustrates. Assuming that there is a statement at $8040 which is required for proper operation of the program, all we have to do is make sure the following piece of code is executed sometime before the code at $8040 is executed:

```
LDA #$02
STA $8040
```

If the program is in ROM, this code will have no effect (on the ROM, at least). If the program is in RAM, however, the statement at $8040 will be altered and the program will crash when the statement is executed. The value $02 was chosen because it is not a standard 6502 opcode; it is an undocumented opcode which causes the processor to lock up. Depending on what the code at $8040 actually does, many other values could probably have been used instead of $02. The value you choose doesn't even have to make the program halt. For instance, in a program that depends heavily on sound, you could simply alter one key value that is placed in the sound chip. As long as the altered byte disrupts the normal operation of the program enough to make the copy unuseable, you have succeeded in protecting the program. Memory locations can be also changed with STX, STY, DEC, INC, ASL, LSR, ROL or ROR.

Many cartridges aren't any better protected than this. However, experienced pirates can find this code and defeat it in a matter of minutes. Here's how they do it. First, they make a RAM copy of the cartridge and run it. When it crashes, they RESET the computer and save out the crashed copy. By comparing the crashed copy with the original, using a machine language monitor, they can see at once that the only difference is the byte at $8040. Next they would use the hunt feature of the monitor to find all references to location $8040. By looking at the areas of code listed, they would quickly spot the code you used as the source of the protection. By changing the STA statement to a set of NOPs (or simply a LDA), the protection would be defeated.

The reason this protection code was so easy to find is that location $8040 was referred to directly. If we could "hide" the reference to location $8040, we could make the protection scheme much more difficult to find with a hunt command. One way to hide memory references is to use other ADDRESSING MODES. In the example above we used what is called DIRECT ADDRESSING: the location is specified directly in the STA command. Other addressing modes that we could have used instead of direct addressing include INDEXED ADDRESSING, INDIRECT ADDRESSING and two combinations of these modes called INDIRECT INDEXED and INDEXED INDIRECT.

INDEXED addressing uses the X or Y register as an offset (index). The value of the index is added to the address specified in the STA statement. The result gives the actual location to be stored into. For example, we could have used the following piece of code:

```
LDA #$02
LDY #$40
STA $8000,Y
```

In the STA statement, the value of Y ($40) is added to the address $8000 (called the base address) to calculate the final address to be used: $8000 + $40 = $8040. So this piece of code will do exactly what our previous example did. The value of the accumulator ($02) will be stored in location $8040. You could also have used the X register as the index instead of Y to accomplish the same thing.

In INDIRECT addressing, the address is specified indirectly by giving a location which contains the address, rather than the address itself. For instance, rather than use the address $8040 directly, we could store the bytes $80 and $40 in two consecutive locations, say $9000 and $9001. Actually, we have to store the bytes in reverse order - the low byte ($40) is stored in $9000 and the high byte ($80) is stored in $9001. Locations $9000-01 would then be called a VECTOR pointing to $8040. The vector is NOT the address itself; the vector is the locations that CONTAIN the address.

The only instruction on the 6502/6510 processor that uses pure indirect addressing is the indirect JMP instruction:

```
JMP ($9000)
```

Assuming locations $9000-01 are a vector pointing to $8040, as explained above, this instruction would cause a jump to $8040. The parentheses around $9000 are used to indicate that it is a vector, not the final address.

More useful than pure indirect is a hybrid form called INDIRECT INDEXED. This form combines indirect and indexed addressing. First an address is retrieved from the vector, and then the index is added to it to get the final address. There are two important things to remember about indirect indexed. The vector itself must be located in zero page ($0000-00FF), and the Y-register must be used as the index. Remember, the vector must be located in zero page but it can point to (contain the address of) ANY location in memory.

Let's go back to our earlier examples in which we want to store a $02 byte at $8040. This time we'll use indirect indexed addressing to do it. First we must set up an address in our vector. Since the index will be added to this address before the STA takes place, the vector does not have to point to $8040 itself. It can point to any location 0-255 bytes before $8040. Let's point the vector to, say, $8000. Remember that this vector must be located in zero page. We'll use locations $FB-FC as the vector, so we'll put a $00 byte in $FB and an $80 byte in $FC. Here's what the code looks like:

```
LDA #$00
STA $FB
LDA #$80
STA $FC
LDA #$02
LDY #$40
STA ($FB),Y
```

The first four statements set up the vector at $FB-FC. The fifth statement (LDA #$02) gets our crash value ready, and the next one initializes our index (LDY #$40). The STA statement first retrieves the address $8000 from the vector at $FB-FC, then adds the index value $40 to it: $8000 + 40 = $8040. This gives the final address, where the accumulator value ($02) will be stored.

Our next addressing mode is called indexed indirect. Both this mode and the mode in the last example use indirect and indexing, but in a different order. If you have trouble keeping their names apart, join the crowd. Just remember that the name tells in what order the two modes will be used. Indirect indexed (our last example) FIRST retrieves an address from the indirect location given in the statement, and THEN adds the index to the address retrieved to get the final address. Indexed indirect (our next example) FIRST adds the index to the address given in the statement, and THEN uses the result as the indirect location (vector) to retrieve the final address from. In this new mode, the address given in the statement is not the final address and it isn't even the location of the vector. It is a base address which the index is added to, in order to get the location of the vector. Then the final address is retrieved from that location.

In this new mode, indexed indirect, the actual vector must once again be located in zero page (remember, it can still POINT anywhere in memory). This time, however, the X-register must be used for the index rather than Y. Here's an example:

```
LDA #$40
STA $FB
LDA #$80
STA $FC
LDA #$02
LDX #$2B
STA ($D0,X)
```

Compare this code to the last example. In this case the vector is still located at $FB-FC, but now it does point to the exact final address, $8040, since no index will be added to it. The index will be used to FIND the vector. In the STA statement, first the index is added to the base address given: $D0 + $2B = $FB. The result, $FB, is the LOCATION of the vector. The CONTENTS of the vector $FB-FC are then retrieved to yield the final address, $8040. Note that the parentheses are put around BOTH the base address $D0 and the index X, in contrast to our last example.

We brought up these different addressing modes to show how they can be used to hide a memory reference from pirates. Even if the pirate knows that location $8040 was changed in their RAM copy, there won't be any direct reference to that locations in your program code.

Pirates are persistent folk, so they may try another approach. By searching for $80 bytes and $40 bytes separately, they might still be able to find your protection code. They would look for a place where the two occur near one another in your program. To defeat this technique, you can separate the use of $80 and $40 by putting them in widely scattered parts of the program, such as different subroutines. Or use $80 and $40 bytes all over the place in your program, to give them hundreds of places to check.

Another way to throw pirates off the track is to avoid using a $80 or $40 byte entirely (assuming the address we're trying to hide is $8040). Instead, take some value and manipulate it mathematically to get the $40 or $80 value that you store in the vector. This technique amounts to a form of encryption. There are many different instructions you can use to do this, such as INC, INX, INY, DEC, DEX, DEY, ADC, SBC, EOR, AND, OR, ASL, LSR, ROL and ROR. We'll cover a couple of examples just to get you thinking:

```
LDA #$7F
STA $FC
INC $FC
```

After the INC instruction, location $FC will contain an $80 byte, the correct high byte of the vector.

```
LDA #$A1
EOR #$21
STA $FC
```

EOR (exclusive-or) is the key in this example. The value used in the EOR instruction ($21) tells which bits in the accumulator ($A1) to flip (change from 0 to 1 or vice versa). Any bit that is a 1 in the EOR value specifies that the corresponding bit in the accumulator should be flipped. Converting the $A1 and $21 to binary will make this example easier to follow: $A1 = %1010 0001 and $21 = %0010 0001 ("%" indicates binary). Since $21 has 1's in bits 5 and 0, those bits will be flipped in the value $A1. This will give us %1000 0000, which equals $80.

Okay, so now the pirate can't find any references to $8040, $80 or $40. Next they'll try looking for all the instructions that use any of these "tricky" types of addressing. Fortunately, there are many instructions that can use plain indexed addressing to change a memory location: STA, DEC, INC, LSR, ASL, ROL and ROR. Not only that, but plain indexing is very commonly used in many programs for purposes besides protection. On the other hand, STA is only instruction which can use indirect indexed or indexed indirect addressing to change a memory location. If you are going to use these modes in your protection code, you should make a point of using them in many other parts of your program too.

There are many variations possible on the basic theme of changing a RAM copy. You can put one of the methods above inside a loop and use it to wipe out a whole section of code. You could store a value into a location in the cartridge area and then check to see if it was actually changed. Record the result of the check temporarily while you restore the original value to that location. This way the location that was used for the test won't show up different in the RAM copy, and pirates won't have that "foot in the door" to help them find your protection code. If the location did change when you stored to it, crash the program.

Now let's look at some other methods for protecting cartridges that don't involve changing the RAM copy. One set of methods involves checking or changing the memory configuration. At this point you might want to review the chapter on the PLA and the chapter on cartridges and cartridge boards. This will help refresh your memory about the relationship between the PLA, the cartridge lines GAME and EXROM, and the memory select lines LORAM and HIRAM. All these factors play a part in determining the memory configuration.

The GAME and EXROM lines are controlled by the cartridge board and force the PLA to select the correct memory configuration for the particular type of cartridge plugged in. Every cartridge must ground either the GAME or EXROM line, or both, in order to be switched into memory by the PLA. These lines are connected to the cartridge port ONLY and cannot be changed by software, so they are excellent indicators of whether a cartridge is actually plugged in. Unfortunately, you can't directly read the setting of these lines from software to see if a cartridge is present. As we'll see, you CAN check these lines indirectly.

The other two lines that play a part in selecting the memory configuration are LORAM and HIRAM. Unlike GAME and EXROM, LORAM and HIRAM can be checked or changed through software alone, by simply reading or writing location $0001. LORAM is bit 0 of location $0001 and HIRAM is bit 1. These lines are used to switch the BASIC and KERNAL ROMs, respectively, in or out of memory. Generally, setting one of these lines to a 1 (the normal setting) will switch the corresponding ROM of the computer into memory; setting the line to a 0 will switch the ROM out and uncover the RAM underneath it. However, HIRAM and LORAM interact with GAME and EXROM to some degree, so that they can have other effects besides just switching a ROM in or out. These side-effects can be used indirectly to determine the state of the GAME and EXROM lines and thereby check that the program is being executed from cartridge. To be on the safe side, always set location $0000 to $2F **AFTER** you change HIRAM or LORAM in location 1. This will make sure that these lines are enabled to be changed.

The first method we're going to discuss can be used with either 8K or 16K standard cartridges, but not with MAX cartridges. For now, let's assume we have an 8K cartridge. 8K cartridges ground the EXROM line only, which tells the PLA will enable the cartridge EPROM any time a read is done from the $8000-9FFF area. this means the computer will always "see" the cartridge memory in those areas when reading. When writing, however, the values will be put into the underlying RAM.

This assumes that the LORAM and HIRAM bits are set to their normal value, 1 (BASIC and KERNAL switched in). Interesting things happen if we set LORAM or HIRAM to 0. For instance, when an 8K cartridge is plugged in, setting LORAM to 0 still switches out the BASIC ROM, but **it also switches out the cartridge.** You can use this side-effect to check if your program is actually executing in cartridge.

Here's how to use this method. Have your program set LORAM to 0, using the following code:

```
LDA $01
AND #$FE
STA $01
```

This will change bit 0 (LORAM) of location $0001 to a 0 without affecting any other bits. Now read from a location in the $8000-9FFF area. If the program was executing in cartridge, any location in this area will change when you set LORAM to 0, since the cartridge will disappear and the RAM underneath will appear. If the program was executing in RAM, however, setting LORAM to 0 will have no effect on the $8000-9FFF area, and the location you check will NOT change. Just to be on the safe side, you should check several locations since the underlying RAM might accidentally contain the same value as the cartridge in one or two locations. Pick locations that don't contain values such as $00 or $FF in your cartridge, since these values commonly appear in RAM on powerup. If a number of different locations don't change, then the program was definitely in RAM. In that case your program should take appropriate steps to crash itself.

You may have noticed one problem with this method. If you switch the cartridge out, the routine that does the checking will be switched out too and the program will crash immediately. The solution is to download your checking routine into some area of RAM before you set LORAM to 0. Then the routine can safely change memory configurations without switching itself out. There are a couple of other things that should be done to increase the protection value of this method. One is to encrypt the version of the routine in the cartridge and then decrypt it as it is being downloaded to RAM. After the check, whether it passes or fails, the cartridge should be switched back in and the checking routine should be wiped out of RAM memory. Use the following code to switch the cartridge back in by changing bit 0 of location $0001 to a 1:

```
LDA $01
ORA #$01
STA $01
```

Be sure to choose an area of memory for your checking routine that is normally used by the program for other purposes, or choose a system area like the input buffer at $0200-58. This way pirates will not notice that you wiped an area of memory and go looking for the program code that does it.

HIRAM (bit 1 of location $0001) can be used much the same way we used LORAM in the last example. Setting HIRAM to 0 switches out the KERNAL ROM, but it also switches out 8K cartridges. In fact, HIRAM switches out BASIC too. Setting both HIRAM and LORAM to 0 will have the same effect, except that the I/O devices will be switched out too, leaving 64K of RAM. If you use HIRAM to switch out your cartridge, you must disable IRQ interrrupts first (with a SEI instruction). Be sure to enable interrupts again (with CLI) when you are ready to resume normal operations.

To set HIRAM to 0, use this code:

```
SEI
LDA $01
AND #$FD     (use #$FC to set both
STA $01       HIRAM and LORAM to 0)
```

To set HIRAM to 1, use this code:

```
LDA $01
ORA #$02     (use #$03 to set both
STA $01       HIRAM and LORAM to 1)
CLI
```

Now let's look at 16K cartridges, which ground both the GAME and EXROM lines. In this case the PLA will enable the first cartridge ROM when reading from the $8000-9FFF area, and enable the second cartridge ROM when reading from $A000-BFFF (the BASIC ROM is automatically switched out with 16K cartridges). The protection method we just covered can be used in exactly the same way with 16K cartridges. Setting LORAM to 0 will switch out the lower half of the cartridge at $8000-9FFF, but leave the upper half of the cartridge switched in at $A000-BFFF. Since the upper half remains switched in, your protection code can reside there instead of being downloaded to RAM. This may be counterproductive, though, because then you won't be able to encrypt the protection code. As with 8K cartridges, setting HIRAM to a 0 will switch out the KERNAL ROM and both ROMs of a 16K cartridge. BASIC is switched out too, so all you get is RAM there. Setting both HIRAM and LORAM to 0 will do all these things and switch out the I/O devices too.

16K cartridges offer the opportunity to use another protection method that involves LORAM. Recall that the PLA automatically switches out the BASIC ROM for 16K cartridges, without your cartridge having to set LORAM to 0. This means that if pirates want to run your cartridge from RAM, they must switch out the BASIC ROM themselves first, using LORAM. There are two things you can do to defeat them. The first is to simply check the LORAM bit at some point and crash the program if it is a 0. The other technique is to try to **switch the BASIC ROM back in** by setting LORAM to a 1. If the program is on cartridge, this will not harm anything. If the program is in RAM, however, the BASIC ROM will come back in on top of the second half of the program and prevent it from running. Remember to set location $0000 to $2F AFTER changing LORAM, to make sure the change will take effect.

Now a few words about MAX cartridges. These ground the GAME line only and appear at $E000-FFFF in memory (the KERNAL is switched out automatically). Optionally, the second cartridge EPROM will appear at $8000-9FFF if it is used (very rare). These cartridges can be protected by attempting to write over themselves to crash any RAM copy. Trying to switch in the KERNAL sounds like a good idea, but HIRAM and LORAM have NO EFFECT with MAX cartridges.

Our final set of protection methods is really open-ended, with many possibilities. If your 8K or 16K cartridge uses the CBM80 autostart key, you can take advantage of the fact that almost none of the normal RESET initialization is done before the cartridge is started (see the chapter on autostart cartridges). What good does this do you? Well, the computer must be initialized before a pirate can load your program from disk and run it in RAM. The initialization process will place particular values in many locations, especially low memory ($0000-03FF), the VIC chip and the CIA chips. By checking one or more of these locations, you can detect whether the computer has been initialized yet and crash the program if it has.

It may be best to check locations that your program itself does not use. At least, you should be able to postpone using the location or calling the system routine that initializes it. This way you can put your check further on into your program (the more deeply "buried", the better). Otherwise you'll probably have to do your check near the beginning of your program. This makes it too easy for a pirate who is tracing through your code to uncover your protection. Also, you should pick locations that don't change in normal use, so you'll know a definite value that indicates initialization has taken place. You probably shouldn't check locations that are initialized to $00 or $FF, since these values are often found in memory at random locations on powerup.

One memory area in particular is worth mentioning, if only because it has been used in many commercial cartridges. When the computer is RESET (as on powerup), the CIA chips are RESET too. This stops all of their timers from running and sets them to zero, including the timer that is used to generate the normal system IRQ interrupts. This is Timer A of CIA #1, located at $DC04-05. During normal RESET initialization, this timer is started up by the IOINIT routine ($FDA3). If this timer has a non-zero value when your program checks it, you can be sure that initialization has been done. Time to crash the program! Actually, this particular location has been used so much and is so well known among cartridge pirates that it should probably be avoided.

BASIC's system RAM locations may be perfect places to check, especially if you don't use the corresponding BASIC subroutines. Most of BASIC's initialization is done by subroutines at $E453 and $E3BF. BASIC's areas must be initialized for a LOAD or RUN command, which will have to be used to execute a pirated RAM copy from disk.

Some final words of advice, that can be applied to program protection in general as well as to cartridges. Don't put all your eggs in one basket - use several different forms of protection in your program. Combining protection methods multiplies the difficulty of breaking the program. Don't assume that pirates are as stupid as they are greedy, and don't forget about the psychological aspects of protection. After spending hours and hours, removing one protection scheme after another, pirates may get to wondering what else is in store for them. They may eventually decide that it's not worth the trouble and go on to someone else's program instead. Remember, if they only miss one of your protection methods, you've still protected the program. Good luck!

C128 CARTRIDGES

The Commodore 128 is a substantial upgrade of the Commodore 64. It seems as if nearly every feature of the C64 has been enhanced in the new C128, and cartridges are no exception. (In this chapter, "C128" generally refers to the new 128 mode of the computer. The 64 mode is virtually identical to the C64 machine). The maximum cartridge size has been doubled to 32K, an EPROM socket has been provided inside the machine in addition to the usual cartridge port, a more flexible cartridge autostart system has been implemented, two separate "logical cartridges" can reside on the same physical cartridge and will be executed independently of each other, and more! To top it off, all of the cartridge configurations can be selected through software alone rather than relying on the cartridge hardware (GAME and EXROM are not used in 128 mode). We'll try to give you an overview of all the new cartridge features and configurations, as well as the essential details you'll need to create your own C128 cartridges.

Let's start by looking at where cartridges can reside in the C128's memory space. Recall that an 8-bit processor can only "look" at 64K of memory space at one time. (the C128 contains a new 8502 processor that can be considered identical to the C64's 6510 processor for our purposes). On the C64, a special bank-switching technique is used to switch the BASIC ROM, KERNAL ROM, I/O devices and cartridges in or out of the memory space. Bank-switching is also used on the C128, for switching in 64K banks of RAM as well as system ROMs, cartridges, etc. The main difference on the C128 is that the bank-switching is much more elaborate and flexible. This is especially true for cartridges.

There are two areas of C128 memory that cartridges can appear in. They are called the **MID SPACE ($8000-BFFF)** and the **HI SPACE ($C000-FFFF)**. Note that each area is 16K long. This means up to 32K of cartridge memory can be switched into the C128 memory space simultaneously! In normal operation, the mid space contains part of the BASIC system ($8000-AFFF) and the built-in machine language monitor ($B000-BFFF). The hi space normally contains the screen editor ($C000-CFFF), the I/O devices ($D000-DFFF) and the KERNAL ($E000-FFFF).

The mid space and hi space are controlled completely independently of each other. Through software, you have a choice of four types of memory that can appear in each space: RAM, external cartridge ROM, special **INTERNAL ROM**, or normal system ROM (BASIC, etc.). The internal ROM is similar to an external cartridge. Inside the C128 is an empty 28-pin socket. To get at it, you'll not only have to void your warranty but you'll also have to unsolder the metal heat sink/RFI shield. The internal socket can hold either a 16K or a 32K chip. If a chip is inserted there, it will be recognized by the computer and treated just like an external cartridge. When we use the word cartridge in this chapter, we'll usually mean to include this "internal cartridge".

Let's see how you can select which type of memory will appear in the mid and hi spaces. Recall that on the C64, memory is controlled by the PLA. The PLA in turn is controlled by the GAME and EXROM lines of the cartridge port and the LOROM and HIROM lines that appear in location $0001. On the C128, memory is managed by a new chip called the **MEMORY MANAGEMENT UNIT (MMU).** The PLA and location $0001 are not used for memory management in 128 mode.

The MMU is located in memory at $D500-0B. In addition to the PRIMARY MMU REGISTERS in the $D500 area, the MMU can also be accessed through its SECONDARY REGISTERS located at $FF00-04. The secondary MMU registers are present in all memory configurations, regardless of whether system ROM, cartridge ROM or RAM has been selected for the hi space. They form a "hole" in memory that can't be covered up (don't put any cartridge code in the $FF00-04 area). The reason this was set up is that it's possible to switch out the entire I/O block ($D000-DFFF), including the primary MMU registers. The secondary registers are provided so you can switch the primary registers back in! You may need the primary MMU registers because the secondary registers don't have all the capabilities of the primary registers. We won't need the secondary registers for our purposes, however.

In fact, the only primary MMU register we really have to deal with is **CONFIGURATION REGISTER 1 (CR1).** The CR1 register is located at $D500 (the secondary register at $FF00 is called CR2 and follows the same format as CR1). Figure 10-1 lists the function of each bit of CR1.

Figure 10-1: MMU CONFIGURATION REGISTER

Bit	Function	Options
7-6	RAM bank	00=bank 0, 01=bank 1, etc.
5-4	Hi space	00=System ROM, 10=External ROM,
3-2	Mid space	01=Internal ROM, 11=RAM
1	Lo space	0=System ROM, 1=RAM
0	$D000 area	0=I/O devices, 1=System ROM

We need to look more closely at two particular pairs of bits. Bits 5-4 control the hi space and bits 3-2 control the mid space. Each bit pair follows the same format. Storing a $00 value into a bit pair will switch in the system ROM that normally occupies the corresponding space. Storing a value of $01 in a bit pair will switch in half (16K) of the special internal ROM instead of system ROM. Storing a $10 value will switch in half of the external ROM (cartridge) memory. Finally, storing a value of $11 in a bit pair will switch RAM into the corresponding space. That's all there is to it - just decide what you want to appear in the mid or hi space, then set the corresponding bit pair to the correct value.

Now that we know HOW to switch in cartridge memory, we have to consider how the system knows WHEN to switch in a cartridge. It can't depend on the PLA to switch the cartridge into memory automatically based on the GAME and EXROM lines. Neither the PLA nor the GAME/EXROM lines are used in the C128 mode. The C128 has to switch the cartridge into memory itself through the MMU when it's time to run the cartridge. But first it has to figure out that there's a cartridge plugged in at all.

To see if there are any cartridges plugged in, the system conducts a **poll** of all the possible cartridge areas (internal or external, hi space or mid space). This is done during the system RESET routine, using a subroutine called POLL located at $E242-BB in the KERNAL (all references to the C128 KERNAL are based on a version labeled 318020-03 on the chip). The first thing POLL does is check the GAME and EXROM lines. If either line is grounded, it indicates that a C64 cartridge is plugged in. In this case, POLL will automatically jump into 64 mode. Otherwise, POLL begins checking the 128 cartridge areas. POLL checks the cartridge areas in the following order: external mid, external hi, internal mid and finally , internal hi (i.e., external before internal, mid before hi).

In each of the cartridge areas, POLL checks for a special **cartridge key string** starting at relative byte $07. For example, in the mid space ($8000-BFFF) POLL will start checking at $8007. The cartridge key string on the C128 is the letters **CBM** (not CBM80 like the C64). Besides the CBM string, the cartridge key contains a cold and warm start entry point and a cartridge ID. Figure 10-2 shows the complete format for the cartridge key.

Figure 10-2: CARTRIDGE KEY

Bytes	Description
x000-02	Cold start entry
x003-05	Warm start entry
x006	Cartridge ID byte
x007-09	"CBM" string

Note: x = $8 (mid) or $C (hi)

If the CBM is found in a particular cartridge area, the POLL routine "logs in" the cartridge by recording the ID byte in a special table. This table is called the **PHYSICAL ADDRESS TABLE (PAT)** and it is located at $0AC1-C4. The ID's are stored here in reverse of the order they were checked, e.g., $0AC4 holds the first area's ID (the external mid ID). The PAT is cleared by filling it with $00 bytes before POLL begins checking for cartridges.

After storing an ID in the PAT, and before checking the next area, POLL checks to see if the cartridge it found is an autostart cartridge. **Any cartridge that has an ID = $01 is considered an autostart cartridge.** If POLL finds an autostart cartridge, it immediately switches in the cartridge area and does a JSR to the cold start entry in the cartridge key. Actually, POLL always switches in both the mid and hi space cartridge areas together. Your cartridge can switch the other cartridge space back out if so desired.

NOTE: The cold start entry is an entry point, NOT a vector (it is three bytes long, not two). POLL does not JSR based on the CONTENTS of the cold start entry, it JSR's directly TO the cold start entry point. POLL expects to encounter executable machine code there. Normally, this will be a JMP to the actual start of the cartridge code. The warm start entry is supposed to be set up the same way, but it's never used by the system.

Since POLL does a JSR to an autostart cartridge rather than JMPing to it, the cartridge has a choice of whether it wants to return to POLL or not. If all the cartridge needs to do at this point is some initialization, it can do so and then return back to POLL via an RTS. If the cartridge wants to take over completely, it can do so by simply never returning to POLL. This is more convenient than the arrangement on the C64.

If no CBM string was found in a particular area, the PAT will contain a $00 ID byte for that area. As far as the system is concerned, no cartridge exists there. You might actually have a cartridge plugged in, but without the CBM string the system will never switch it in or try to execute it. For that matter, even if there is a CBM string, if the ID in the key is $00 the cartridge will never be called. Of course, you still can execute a cartridge yourself "manually". You'll have to switch it into memory and jump to it using a special routine in low RAM (below $4000).

Unless an autostart cartridge takes over control, POLL will continue to search all four of the possible cartridge areas and log in any cartridges it finds. When POLL is done, the normal system RESET process will continue. The last major step in the RESET process (under normal conditions) is to call the BASIC system's initialization routine through the SYSTEM VECTOR. This vector is located at $0A00-01. It is initialized to point to the start of BASIC ($4000) by the RAMTAS routine at $E093-CC.

When BASIC has finished its initialization, it calls upon a special routine called PHOENIX at $F867-8F (also reachable through a new KERNAL jump table entry at $FF56). PHOENIX checks through the PAT to see if any cartridges have been logged in. It checks the PAT in the same order that POLL uses, starting at $0AC4 (external mid) and ending at $0AC1 (internal hi). Any PAT entry (ID) of $00 is skipped. If a non-$00 ID is found, PHOENIX immediately switches in the cartridge and does a JSR to the cartridge's cold start entry. It knows which cartridge to switch in (internal or external) by the ID's location in the PAT.

As with POLL, cartridges have a choice of returning to PHOENIX via an RTS (perhaps after patching into some system vectors), or taking over completely by never returning. If a cartridge is executed but returns to PHOENIX, then PHOENIX will continue stepping through the PAT, executing any cartridges it comes across. If and when PHOENIX finishes checking all four ID's in the PAT, it falls through to another new routine called BOOT CALL at $F890 (new KERNAL call $FF53). BOOT CALL checks the current drive for an autoboot disk (another new feature of the C128). If the special disk boot key (also a CBM string) is found at sector 1/0, the indicated program or set of sectors is loaded and executed. Otherwise, BOOT CALL will return to BASIC initialization, which will terminate in the familiar READY mode.

As you can see, the C128 handles cartridges much differently than the C64. Perhaps a quick review of the process would help at this point. There are four possible areas where a cartridge might reside. These areas are checked on RESET by the POLL routine. If POLL finds a CBM cartridge key in an area, the cartridge will be logged in by storing its ID in the PAT table. Otherwise a $00 ID will stored in the PAT entry for that area. Any cartridge with an ID of $01 will be executed immediately by POLL using a JSR to the cold start entry point. Unless an autostart cartridge takes over for good, POLL will continue to look for cartridges until all areas have been checked.

Eventually, BASIC will be initialized. As part of its initialization, it calls upon the PHOENIX routine. PHOENIX searches the PAT (not the cartridge areas) for logged-in cartridges and executes any it finds, again using JSR. If none of the cartridges take over control, the disk will be checked for a disk autoboot key. If it is found, the disk will be autobooted. Otherwise, control will be turned over to the user through BASIC's READY mode.

There are still a couple of loose ends that must be tied up to complete our discussion of C128 cartridges. For instance, when we speak of the system switching in a cartridge, it's important to know exactly what memory configuration will be chosen. The C128 has a set of configurations that can be referenced by number rather than having to figure out the correct value to put in the MMU's configuration register. These are the same configuration numbers that are listed for use with the built-in ML monitor on p. 372 of the System Guide that comes with the C128.

The actual value placed in the MMU for each configuration number (cfg #) is given in a table located in the KERNAL at $F7F0-FF. Cfg #0 uses the first value in the table ($3F) and so on. When the C128 switches in a cartridge to check for a CBM key or execute the cartridge, it uses a different cfg# depending on whether it is an internal or external cartridge. The same cfg# is used regardless of whether the cartridge is in hi space or lo space - both cartridge spaces are switched in together. For internal cartridges, cfg #4 is used, and the value $16 is placed in the MMU's configuration register. For external cartridges, cfg #8 is used, and the value $2A is placed in the MMU.

Note that both cartridge spaces (mid and hi) are always switched in together, even though it is possible to switch them in independently of each other. You may only want one of the cartridge spaces switched in for your cartridge. For example, if you are setting up a 16K cartridge in mid space, you may well want the system ROMs switched into hi space so you can use the KERNAL. Your cartridge itself will have to switch the system ROMs back into hi space. The cartridge can do this by simply putting the value $0A into CR2 ($FF00). In general, it's a good idea to always use CR2 rather than the primary configuration register CR1 because CR2 is ALWAYS in memory. Since CR1 can be switched out by switching out the I/O devices, using CR2 is better in general.

Other cartridges, internal or external, mid or hi, may also need to change the memory configuration after they are executed. Perhaps they need RAM in an area, or want to use the KERNAL. In any case, the cartridge should disable IRQ interrupts with an SEI instruction as soon as the cartridge is executed, before trying to change the memory configuration. The IRQ routine could potentially call on subroutines in both the KERNAL and BASIC ROMs, so it's best to disable interrupts in all situations. You'll probably want to enable interrupts again after changing the memory configuration.

We've included a sample C128 cartridge on the disk with this book. When it is executed, it demonstrates a couple new features of the C128 and then returns to BASIC. To set it up, you'll need a 27128 (16K) EPROM and a PC2 cartridge board (both available from CSM). By the way, when you buy EPROMS for the C128, you should get ones rated at **250 ns.** or faster, in order to be able to use them in the C128's new 2MHz mode. In this mode the processor runs twice as fast as normal, so the memory chips must be faster to keep up with it.

You'll have to burn the 27128 EPROM using a C64 or the C128 in 64 mode, since there is no C128 version of PROMOS (at least as of 11/85). Power off the computer, plug in the PROMENADE and power the computer on again. If you're using a C128, go into 64 mode by either holding down the C= key as you switch the computer on, or typing GO64 after the computer comes up in 128 mode. Load and execute both PROMOS and your machine language monitor as usual. Now insert the program disk for this book and load in the cartridge file by entering:

```
L "128 CARTRIDGE", 08
```

The cartridge will load in at $2000 as usual to make burning the EPROM easier, even though it will reside at $8000 in the computer eventually. Initialize the PROMENADE with a "Z" command. Insert the 27128 into the PROMENADE with the notch to the left as usual, and flip the lever down. The command to burn the EPROM is:

```
PI 8192, 8300, 0, 5, 6
```

DON'T type the letters PI, use the pi key on the keyboard (next to the RESTORE key). This command only programs a small amount of memory, so it should only take a few seconds. If you are using a 27128**A** EPROM, be sure to use a CW of 6 rather than 5 in the above command.

All you have to do now is set up the cartridge board for use on the C128. In order to use a regular C64 cartridge board to make a C128 cartridge, you'll have to make a couple minor modifications to it. Since board designs vary, different commercial boards may require different modifications. The following discussion applies specifically to the PC2 cartridge board available from CSM. A summary of the modifications you'll have to make appears below in figure 10-3.

Figure 10-3: CHANGES TO PC2 FOR C128 CARTRIDGES

1. Disconnect GAME/EXROM from ground (cut jumper A).

2. Disconnect GAME and EXROM from each other (cut jumper B).

3. Disconnect pin 26 from pin 27 on each socket (underside of board).

4. Connect halves of unlabeled jumper for pin 26 of HI ROM (from pin J of cartridge connector).

5. Connect pin 26 of one socket to pin 26 of the other socket with a jumper wire.

The first thing you must do is make sure the GAME and EXROM lines are not grounded by the board. If the C128 POLL routine sees that either of these lines are grounded, it will jump into C64 mode. To keep these lines hi (not grounded), you must cut through jumper **A** on the top side (EPROM side) of the PC2 board. This will separate pins 8 & 9 of the cartridge connector (GAME and EXROM) from the wide ground line.

On the C64, GAME and EXROM are only capable of being inputs to the computer. On the C128, these lines are bidirectional, i.e. they can be used as output lines too. In fact, on the C128 the GAME line has a dedicated use as an output line, not related to cartridges. To prevent possible interference, you should also separate GAME and EXROM from each other. This can be done by cutting jumper **B** on the PC2 board.

When a PC2 board is used on the C64, each socket holds an 8K EPROM (2764). On the C128, each socket holds a 16K EPROM (27128). There are a couple changes that must be made to the board in order to use 16K chips. Refer to the appendix for the pinouts for a 27128 chip. First, locate pin 26 of the HI ROM socket. This is the LEFT socket of the two when the board is right-side up (EPROM side up). Pin 26 is the third pin from the TOP RIGHT corner of the socket (the top is the end away from the cartridge connector). Put your finger by this pin. Keeping the end with the cartridge connector towards you, turn the board over.

On the bottom side of the board, the HI ROM socket will be on the RIGHT side and pin 26 will be the third pin from the top LEFT corner of the socket. Make sure you've got this straight before you go any further. On the underside, pin 26 will usually be connected to pin 27 (and pin 27 will usually be connected to pin 28). They may be connected by a drop of solder or a short line on the circuit board. Pin 26 isn't used on 2764 (8K) chips, but on 27128 (16K) chips it's used for address line 13. You must disconnect pin 26 from pin 27 by cutting the circuit line or removing the drop of solder. You should leave pin 27 connected to pin 28. Now follow the same procedure to disconnect pins 26 and 27 on the other socket (LO ROM) from each other.

Next, locate the jumper pad that is connected to pin 26 of the HI ROM socket by a short line. The pad is not labeled. There is also an unlabeled jumper pad connected to pin 27, so make sure you have the right one. The pad separates pin 26 of the socket from pin J of the cartridge connector (the 8th pin from the right on the underside of the connector). This jumper is usually open (cut) so there is no connection between the two halves. Scrape the jumper pad clean of green solder mask. Put a drop of solder on the jumper pad to connect the two halves. This will allow the entire 16K of memory in the HI ROM to be accessed. The ROM will appear at $C000-FFFF in memory (hi space) when the cartridge is enabled.

Finally, you have to run a jumper wire from pin 26 of one socket to pin 26 of the other socket. Use a short piece of insulated wire. Make sure that the wire doesn't make contact with any other pin on either socket. We recommend that you put your cartridge board in a plastic housing to protect it once it's finished.

To finish up our sample C128 cartridge, simply insert the EPROM into the RIGHT socket on the board. Make sure the notch on the chip matches the notch on the socket and all the pins are started in their holes before you press down on the EPROM. To try the cartridge, power off the computer and insert the cartridge. If you have an expansion board, you might want to plug it in first. With the cartridge slot on (if you're using an expansion board), power up the computer. You should see an interesting display on the screen (40 column screen only). When you are returned to BASIC, you might want to experiment with the screen. Try loading a disk directory and see what happens.

Of course, this little demonstration cartridge is only a start, but with the information in this chapter you may be able to create much more useful cartridges. We have tried to give you all the essential information on C128 cartridges in this chapter. The C128 is a powerful and complex machine, so you'll need to get to know it better before you can program it effectively.

(INTENTIONALLY LEFT BLANK)

8K BASIC CARTRIDGE

Our first cartridge utility program is a simple download-and-run routine for BASIC programs. BASIC programs up to 8K long can be put on cartridge with this routine. BASIC program size is limited to 8K since 16K cartridges replace the BASIC ROM in memory. Without the BASIC ROM, you can't run BASIC programs. The utilities featured in later chapters allow you to install larger programs on cartridge if you need to, but the routine in this chapter can't be beat for simplicity and ease of use.

The download routine has two options. If you activate the cartridge through a RESET (power-up, RESET button or SYS 64738), the BASIC program will be downloaded to the normal BASIC area (starting at $0801) and RUN automatically. The operating system and BASIC will be completely initialized first, to insure smooth operation of the BASIC program. One side-effect of the RESET option that you should be aware of is that the top-of-BASIC pointer MEMSIZ, located at 643-644 ($0283-0284), will be set to 32768 ($8000) rather than 40960 ($A000). This will give you less free space available for variables than normal. This happens because the RAMTEST routine, which sets up MEMSIZ, will detect the end of RAM when it reaches your cartridge at $8000, rather than when it encounters the BASIC ROM at $A000. Fortunately, this won't affect very many BASIC programs unless they use large arrays or lots of strings. If MEMSIZ your BASIC program has trouble because of MEMSIZ, use the RESTORE key option (below).

If you use the RESTORE key rather than RESET to activate the cartridge, the BASIC program will be downloaded but will not be RUN. This way you can list the program, modify it, save it, etc. You can use the RUN command to execute the program whenever you want. Another feature with RESTORE is that the BASIC program will be downloaded to whatever area is currently set for BASIC, which can be different from the normal BASIC area. If your BASIC program must reside higher up in memory than normal, simply set the MEMSTR pointer at 641-642 ($0281-82) to the new location before you hit RESTORE. Like all pointers, MEMSTR is stored with the low order byte first and high order byte second (reverse order). For instance, if you want the BASIC area to start at 4096 ($1000), the low byte would be 0 ($00) and the high byte would be 16 ($10). Enter POKE 681,0 and POKE 682,16 before you hit RESTORE.

The RESTORE option has one other feature. Since a RESET is not performed, the top-of-BASIC pointer MEMSIZ will be left at its current setting (see the RESET option). If you have an expansion board, you can switch on your cartridge, hit RESTORE and then switch the cartridge back off. The full BASIC area will be available for variables this way. Just enter RUN to execute the program. Note that you MUST turn the cartridge off after RESTORE because BASIC will try to use the cartridge area, since MEMSIZ indicates the area is available.

Setting up a cartridge with the download/run routine is simple. The process is summarized in figure 11-1.

Figure 11-1: 8K BASIC CARTRIDGE PROCEDURE

1. Load in the "8K BASIC" program at $2000.
2. Load in your BASIC program at $2100.
3. Put the ending address of the BASIC program into the download routine.
4. Select your screen colors.
5. Save the file out to disk if desired.
6. Load PROMOS and burn the EPROM.
7. Insert the EPROM into an 8K cartridge board.

You'll need an ML monitor to set up the cartridge. MICROMON is recommended because of its relocatable load feature. Load the monitor into memory at $C000. Clear the cartridge area by filling it with $FF's with the fill command:

```
F 2000 3FFF FF
```

Using MICROMON, load in the file "8K BASIC" from the program disk accompanying this book:

```
L "8K BASIC"
```

The "8K BASIC" routine will load in at $2000. Although the finished cartridge will reside at $8000 in memory, we'll put it together at $2000 to make burning it onto EPROM more convenient. Remember, unlike a program file, an EPROM has no idea where in memory it resides. As explained in the chapters on memory management and cartridge boards, it is the PLA that "tells" the EPROM where it is in memory.

Your BASIC program must now be loaded in starting at $2100. Use MICROMON's relocatable load command to do this:

```
L 2100 "program name"
```

Now you must tell the download routine where the end of the BASIC program is **(plus 1)**. This is easy because MICROMON always reports the ending address (plus 1) of the program after a load. Make a note of the address given. Since the BASIC program will begin at $8100 in the finished cartridge rather than $2100, you must add $6000 to the ending address given in order to translate it correctly. For instance, an ending address of $2500 would become $8500, and an ending address of $3500 would become $9500. An ending address over $4000 means the BASIC program is too large to fit in the cartridge! ($4000 itself is acceptable).

Once you have translated the ending address, it must be inserted into the download routine at locations $2009-0A. Use the command M 2009 to display this section of memory. The ending address must given in lo-byte/hi-byte order like other pointers. The address $8500 would be specified by putting the low byte ($00) at $2009 and the high byte ($85) at $200A.

Just after the ending address pointer in the download routine are three color codes at $200B-0D. These are the border, background and text colors, respectively, that will be used when the cartridge is activated. The default will be white text on an all black screen. If you have your own personal favorite colors (and who doesn't?) just fill them in at the same time you set the ending address for the BASIC program.

At this point the program is ready to burn onto EPROM. You may wish to save a copy to disk first, as a backup. Use the following MICROMON command:

```
S "program", 2000, 4000
```

To burn the program onto EPROM, exit MICROMON with the "K" command. Load in PROMOS from disk and run it. Now enter MICROMON again. You'll need to issue a PROMOS "Z" command to initialize the PROMENADE and extinguish the PGM and SKT lights. Now insert a 2764 (8K) EPROM into the PROMENADE with the notch on the EPROM to the **left**. Lock down the handle on the ZIF socket. One simple command is all you need to burn the EPROM:

```
PI 8192, 16383, 0, 5, 7
```

Note: Do not type the letters "PI" - use the pi key on the keyboard. This is a shifted up-arrow (not the cursor up key). It is located next to the RESTORE key. All addresses for PROMOS must be given in **decimal**: 8192 = $2000 and 16383 = $3FFF

It's a good idea to read the EPROM back in now and compare it to the original. Read the EPROM using:

```
£ 16384, 24575, 0, 5
```

(16384 = $4000; 24575 = $5FFF). Compare the copy to the original:

```
C 4000 5FFF 2000
```

Assuming everything matches, you're ready to mount the EPROM on a cartridge board. Make sure the cartridge board is set up for a standard 8K cartridge, with the EXROM line alone grounded. Insert the EPROM into the $8000-9FFF socket (usually the socket on the right if there is more than one). The notch on the EPROM must correspond to the notch or mark on the socket! Turn off the computer (or just the cartridge slot if you have an expansion board), plug in the cartridge, and turn it back on. If you've done everything correctly, your cartridge should respond when you RESET or use the RESTORE key.

Figure 11-2: 8K BASIC DOWNLOAD/RUN

```
:8000 10 80                    RESET entry = $8010
:8002 24 80                    RESTORE entry = $8024
:8004 C3 C2 CD 38 30           CBM80 Autostart key
:8009 00 A0                    SET TO END OF SOURCE PROGRAM!
:800B 00 00 01                 Colors - change if desired.
```

RESET Entry Point

```
., 8010 8E 16 D0 STX $D016     Turn on VIC
., 8013 20 A3 FD JSR $FDA3     IOINIT - init I/O devices
., 8016 20 50 FD JSR $FD50     RAMTAS - test/init RAM
., 8019 20 15 FD JSR $FD15     RESTOR - init system vectors
., 801C 20 5B FF JSR $FF5B     CINT - init screen editor
., 801F 58       CLI           Enable IRQ interrupts
., 8020 A9 00    LDA #$00      Set entry flag - RESET
., 8022 F0 02    BEQ $8026     Skip next statement
```

RESTORE Entry Point

```
., 8024 A9 FF    LDA #$FF      Set entry flag - RESTORE
., 8026 85 FD    STA $FD       Save entry method flag
., 8028 AD 0B 80 LDA $800B
., 802B 8D 20 D0 STA $D020     Set border color
., 802E AD 0C 80 LDA $800C
., 8031 8D 21 D0 STA $D021     Set background color
., 8034 AD 0D 80 LDA $800D
., 8037 8D 86 02 STA $0286     Set character color
., 803A 20 E7 FF JSR $FFE7     CLALL - Close all files
., 803D 20 53 E4 JSR $E453     Init BASIC vectors
., 8040 20 BF E3 JSR $E3BF     Init RAM for BASIC
., 8043 20 22 E4 JSR $E422     Init screen & do NEW
., 8046 A9 00    LDA #$00      Init source pointer to
., 8048 85 FB    STA $FB         start of BASIC program
., 804A A9 81    LDA #$81        ($FB-FC = $8100)
., 804C 85 FC    STA $FC
., 804E A5 2B    LDA $2B       Init destination pointer
., 8050 85 2D    STA $2D         to start of BASIC area
., 8052 A5 2C    LDA $2C         ($2D-2E = $2B-2C)
., 8054 85 2E    STA $2E
., 8056 A0 00    LDY #$00      Clear Y index register
., 8058 B1 FB    LDA ($FB),Y   Load next source byte
., 805A 91 2D    STA ($2D),Y     and save in destination
., 805C E6 FB    INC $FB       Increment source pointer
., 805E D0 02    BNE $8062
., 8060 E6 FC    INC $FC
., 8062 E6 2D    INC $2D       Increment destination
., 8064 D0 02    BNE $8068       pointer
., 8066 E6 2E    INC $2E
., 8068 A5 FB    LDA $FB       Compare source pointer
., 806A CD 09 80 CMP $8009       to ending value.
., 806D D0 E9    BNE $8058       Continue at $8058
., 806F A5 FC    LDA $FC         if more to do.
., 8071 CD 0A 80 CMP $800A
., 8074 D0 E2    BNE $8058
```

Figure 11-2: cont'd

```
., 8076 A9 00    LDA #$00     Clear Z flag for next call
., 8078 20 71 A8 JSR $A871    CLR - adjust BASIC pointers
., 807B 20 33 A5 JSR $A533    Re-link BASIC lines
., 807E A5 FD    LDA $FD      Check RESET/RESTORE flag
., 8080 D0 03    BNE $8085
., 8082 4C AE A7 JMP $A7AE    RESET - RUN program
., 8085 4C 74 A4 JMP $A474    RESTORE - READY mode
```

Figure 11-2 gives a commented listing of the 8K BASIC routine. As explained in the chapter on autostart cartridges, the CBM80 autostart key at $8004-08 causes the cartridge to be activated on RESET or RESTORE. The pointer stored at $8000-01 will be used when a RESET is done, and the pointer at $8002-03 will used when the RESTORE key is pressed. Thus the RESET entry point is $8010 and the RESTORE entry point is $8024.

On RESET, the routine performs the exact same sequence of events that would be done by the normal RESET routine if the check for a CBM80 had failed. Compare the code at $8010-1F with the code in the KERNAL at $FCEF-FE. This code initializes the I/O chips (VIC, SID & CIAs), system RAM, KERNAL vectors and screen editor. Next, a special flag location ($FD) is set to $00 to indicate that the routine was entered through RESET. The rest of the RESET method is identical to the RESTORE method, except the RESTORE entry point itself is skipped.

When RESTORE is pressed, the entry method flag is set to $FF (this statement is the only one skipped by RESET). Next the border, background and character colors are set from the values you entered. All open files are closed as a precaution, and then a full BASIC initialization is done. This is accomplished by the code at $803D-45, which initializes the BASIC vectors and RAM variables, prints the powerup screen and then does a NEW command.

Finally we enter the actual downloading phase. A pointer at $FB-FC is used to keep track of the location of the next byte to be transferred from the cartridge area (the source). This pointer is initialized to $8100, which is where you put your BASIC program in the cartridge. A pointer at $2D-2E keeps track of the location of the next byte in the BASIC area (the destination). This pointer is initialized to the beginning of the current BASIC area, which is obtained from the BASIC pointer TXTAB stored at $2B-2C. The code at $8056-67 transfers a byte from the source area to the destination area, and then increments the source and destination pointers (first the low byte is incremented, and then the high byte if necessary). Then the source pointer is checked against the ending value you put at $8009-0A when you made the cartridge.

When all the bytes have been transferred, the destination pointer will be left pointing to the next byte past the end of program in the BASIC area. The location that we chose to store this pointer in, $2D-2E, is called VARTAB and is used by BASIC to indicate the end of program text plus 1 (beginning of variable storage). The value left in this pointer will be exactly correct for your program to operate properly - that's why we used this pointer. All we have to do is call the BASIC CLR command through the subroutine at $A871 to set the other BASIC pointers based on VARTAB. Then we call on a BASIC ROM routine at $A533 to recalculate all the links in your BASIC program. This is a precaution to ensure that the program will run in its current location.

Finally, we check the entry method flag that was set earlier. If it is not $00, it indicates that the RESTORE entry was used. In this case the routine terminates by jumping into the BASIC READY mode loop at $A474, which waits for you to enter a command. If the entry method flag is $00, it indicates that the RESET entry was used. In this case the routine terminates by jumping to the program execution loop at $A7AE, which RUNs the downloaded program.

Besides being used in this cartridge, parts of the routine we have given you can be used in other situations where you must initialize the KERNAL or BASIC, or run a BASIC program from machine language. You'll see similar code in some of our other cartridge download routines.

8K/16K CARTRIDGE MENU

This chapter presents a menu program for standard 8K and 16K cartridges. These cartridges reside at $8000-9FFF and $8000-BFFF, respectively, in computer memory and usually use a "CBM80" key at $8000 to autostart themselves. Both cartridges "replace" the RAM at $8000-9FFF, and a 16K cartridge replaces the BASIC ROM at $A000-BFFF too. This normally limits the usefulness of these cartridges for storing and downloading programs, especially in the case of BASIC programs with the 16K cartridge. However, if you have an expansion board such as the CSM Single Slot or CARDCO 5-Slot, you can download a program from cartridge and then use your expansion board to switch the cartridge off. A switch can also be mounted directly on the cartridge for the same purpose (see the chapter on special cartridge hardware). You'll need one of these accessories in order to use our 8K/16K menu.

The menu program works very simply and conveniently. You may enter the menu by doing a RESET (power-up, RESET button or SYS 64738) or by pressing the RESTORE key. On power-up or RESET, the computer is completely initialized before beginning the menu screen. This includes initializing BASIC. When the menu is entered using the RESTORE key instead, no initialization is done at all, except that which is necessary to display the menu on the screen and accept input (KERNAL routine CLRCHN, $FFCC). Special care was taken to minimize the amount of memory disturbed by the menu program. If you enter the menu and then change your mind, you can exit to BASIC without downloading a program, without disturbing a BASIC program already in memory and without even CLOSEing any files!

The menu program displays a menu screen and then inputs an selection number. You are limited to a maximum of 10 selections on the menu, numbered 0-9. Number 0 is reserved for the "Exit to BASIC" selection. You may have less than 10 selections on the menu if you wish. The program automatically checks for a valid selection number in the correct range. When a valid selection is made, the corresponding file is downloaded to RAM memory. The user is then instructed to switch off the cartridge and hit RETURN. When RETURN is pressed, the menu program checks to make sure that the cartridge is actually switched off before executing the downloaded program.

Not only is the menu program easy to use, it is easy to set up too. The disk supplied with this book contains a program called "MENU MAKER". This BASIC program creates a cartridge for you, complete with menu and ready for burning onto EPROM! All you have to do is specify the programs to be put on the cartridge and answer a few questions about them. All necessary adjustments to the menu program are made automatically and a simple menu screen is created. The MENU MAKER program is probably all you'll ever need to set up your own 8K or 16K cartridges. We'll cover the inner workings of the "bare" menu program a little later, but for now, let's look at how to use MENU MAKER.

First, we recommend that you collect together the disks containing the programs you want to put on the cartridge. Note whether the program is in BASIC or machine language (ML), and write down the SYS command used for each ML program. Jot down the size of each program so you can be sure they'll fit on your cartridge. 8K of cartridge memory corresponds to approximately 32 blocks on disk, and 16K corresponds to 64 blocks. The menu program occupies 3 blocks of memory, so you have enough room left on the cartridge for the equivalent of 29 or 61 disk blocks respectively. This is enough for a large BASIC or ML program, or several smaller programs. If you need more memory than this, you'll have to use a bank-switch cartridge, since 16K is the most that can be put on a standard cartridge.

Now load in the cartridge maker program: LOAD "MENU MAKER",8 (the program only works with device 8, drive 0). RUN it. First you are asked if this is an 8K or 16K cartridge. The next three questions ask you what colors should be used for the border, background and characters on the menu screen. Colors are specified by number, with 0=black, 1=white and so on (see p. 139 of the C64 User's Guide for a complete list). Next, you are prompted for the MENU TITLE. A maximum of 20 characters can be entered for the title. You are limited to the uppercase/graphics character set for this title and all other menu text. The title will be centered automatically and printed at the top of the menu screen when the cartridge is run.

Now you must provide some information about each program to be put on the cartridge. To help you decide if you have enough room left for your next program, the number of bytes still available in the cartridge is printed out. Insert the disk containing the first program for the cartridge, and type in the program's file name (it must be a PRG-type file). The file will be loaded from the disk into the area where the cartridge is being assembled. Its location in the cartridge area will be recorded in a special download parameter table so it can be found later. If you run out of room while loading a file, you'll be notified and given a chance to load in a different file (or quit). Errors in the load process will not affect the programs already installed in the cartridge. By the way, it's normal for the drive to start and stop during the load process. It takes a while, so just be patient.

Next, you are asked if this is a BASIC program. BASIC programs require a special boot routine in order to RUN properly. The boot routine is modified to suit your program and then tacked on the end of the BASIC file in the cartridge. It will be wiped out after your BASIC program is downloaded and RUN, so it won't interfere with your program at all. BASIC programs MUST reside at the normal start of BASIC for our boot to work properly (2049=$0801). If your BASIC program runs higher up in memory, you'll have to write your own boot. Look at ours to see what's involved. Tack your boot on the end of the BASIC program ON THE DISK, answer NO to the question about being a BASIC program, and give the location of your boot as the execution address (see below).

If your program is in machine language rather than BASIC, you are prompted for the EXECUTION ADDRESS. This is the address to JMP to after the file is downloaded. The default value for the execution address is the LOAD ADDRESS that was found when the file was read off disk. That is, if you don't enter a value here the program will be executed at its beginning byte. If you want it be executed at some other location, you must enter the location in decimal.

The last prompt is for the MENU ENTRY. This is the line that will appear on the menu screen as the name of this program. The maximum length for the menu entry is 16 characters. The default for this entry is the filename from which the program was loaded. If you just hit RETURN that's the name that will appear on the menu.

At this point the first program has now been installed in the menu routine. The process repeats for each successive program, starting with the bytes free message and file name prompt. When you are finished loading in all the programs for this cartridge, press RETURN at the filename prompt. The MENU MAKER program will also stop automatically after you reach program number 9, since that is the highest number allowed. In either case, all remaining bytes in the cartridge area will be filled with $FF's for neatness. Then you'll be asked to insert a disk and enter a filename in order to save a copy of the cartridge to disk. You can a replace an existing file by using "@0:" in the name. If you don't use "@0:", MENU MAKER will notify you if there is a duplicate name and offer to replace the file for you. Any errors that occur in the save process will not disturb the cartridge area in memory.

That's all there is to it. Your cartridge is now ready to burn onto EPROM(s). If you want to prepare another cartridge now, you'll have to reload the MENU MAKER program to get a fresh blank menu. If you want to burn the cartridge you've just prepared, it's still there in memory. Just RESET the computer and you're ready to go (the cartridge routine won't be affected). Of course, you can also come back later and load the cartridge file from disk. Once you have your cartridge routine in memory, load in the PROMOS software from disk or flip in your PROMOS/HESMON cartridge. Either form of PROMOS can be used since an 8K cartridge routine resides at $2000-3FFF in memory and a 16K resides at $2000-5FFF. If you must load in your ML monitor separately from PROMOS, the best place for it in this situation is at $C000 (49152), although both $1000 (4096) and $9000 (36864) are also available.

An 8K cartridge requires one 2764 EPROM and a 16K cartridge requires two 2764's. In either case, you also need a PC2 cartridge board. Insert an EPROM into the PROMENADE (notch to the left) and lock down the lever. 8K cartridges and the first 8K section of 16K cartridges are stored at $2000-3FFF by MENU MAKER, but we must specify these addresses in decimal for PROMOS. The complete PROMOS command to burn the EPROM is:

PI 8192, 16383, 0, 5, 7

Note that PI stands for the pi key on the keyboard (shift up-arrow), not the letters P I. This command uses the intelligent programming method #1 (PW=7), which is a good compromise between speed and reliability. After the EPROM is programmed, it is good practice to read it back to make sure it was programmed correctly. There is room left in memory from $6000-7FFF to read in an 8K EPROM, so use the command:

£ 24576, 32767, 0, 5

Compare the copy with the original using the following command from your monitor:

C 6000 7FFF 2000

If there are any differences, double-check your commands and see the troubleshooting section in the appendices. Assuming the EPROM was programmed correctly, remove it from the PROMENADE and label it. If you have a 16K cartridge, insert the second EPROM and burn the rest of the cartridge code, located at $4000-5FFF, using the command:

PI 16384, 24575, 0, 5, 7

Read the code back in using the £ command above and compare it to the original with:

C 6000 7FFF 4000

Before you install the EPROM(s) on the PC2 cartridge board, you may have to do a couple of things. First, we highly recommend that you install sockets on the board if they aren't there already. Be sure to align the notch or dot on the socket with the corresponding mark on the board for pin 1. This will avoid confusion later and possible damage to your equipment should you insert an EPROM the wrong way.

Next, you have to make sure that the cartridge grounds the correct combination of the GAME and EXROM lines so the proper memory configuration will be selected. GAME and EXROM are pins 8 and 9 on the **cartridge board edge connector** (not on the EPROMs!). For 16K cartridges, make sure the jumper(s) on the board are **connected, not cut** so there is a continuous path from each pin to the wide ground line on the board. For more detailed information see the instructions that came with your PC2 board or the pinouts section in the appendices of this book. If GAME and EXROM are set correctly, both LEDs (lights) will come on when a 16K cartridge is inserted into your CSM Single Slot or CARDCO 5-Slot expansion board. For 8K cartridges, you must cut a jumper so that only the EXROM line (pin 9) is grounded. If an 8K cartridge is set up correctly, only the right LED will light up when the cartridge is inserted into one of these expansion boards.

All that's left is to mount the EPROM(s) on the PC2 board. The EPROM for an 8K cartridge goes in the right-hand socket of the board. For 16K cartridges, the first EPROM goes in the right socket and the second goes in the left. Make sure the notch on the EPROM is lined up with the notch or dot on its socket. Your cartridge is ready to use! Plug it into your expander board, flip it on, and hit RESET. You should see your beautiful new menu and the prompt to enter your selection. Verify that the programs on the menu do indeed run when selected. You may have to use RESET rather than RESTORE to restart the cartridge after some programs have run. Have fun!

Now let's take a look at the inner workings of the menu program itself. This is the key routine which MENU MAKER modifies to make its cartridges. If you want to change the appearance of the menu or just see how the routine works, you've come to the right place.

The "bare" 8K/16K MENU routine is found in the file of the same name on the program disk. Load it in and look at it with your monitor. It normally resides at $8000 in the computer. It uses a "CBM80" cartridge autostart key to boot itself up on RESET or RESTORE. On RESET (cold-start) the routine starts executing at $8009, while on RESTORE (warm-start) it begins at $8044. If you look at the code at $8009, you'll see that it calls the same initialization routines normally performed on power-up (compare it to the "generic" cold-start routine in the chapter on cartridge intialization). The only difference is the code at $8013-17. This code sets the end of RAM pointer to its normal setting ($A000). This is necessary since the RAM test routine (RAMTAS, $FD50) detected the cartridge at $8000 and set the end of RAM there instead.

Starting at $801F, the menu routine also performs a BASIC initialization. This is possible since the main BASIC initialization routines are actually in the KERNAL ROM! The only exceptions are the start-up screen print routine and the NEW command. Both of these routine reside in the BASIC ROM. However, the BASIC ROM is switched off when a 16K cartridge is plugged in. We don't need the start-up screen, but we do need to do a NEW command. Therefore we have written special code to perform this function. You may find the code at $801F-43 useful in other cartridges or just as a concise summary of BASIC initialization.

The RESET process continues into the main menu code at $8044. This is where the RESTORE entry point is too. The first thing we do is call the CLRCHN ($FFCC) routine to assure us that we can write to the screen and read from the keyboard. Next we set the border and background colors. Then the menu screen itself is printed by a nifty subroutine at $80B0. If you want to alter the appearance of the menu, you'll have to understand what this routine does. This routine can be very useful in other programs too.

The main problem with printing a nice-looking menu is that the menu text takes up a lot of room in the cartridge. In particular, centering the menu on the screen requires a lot of spaces at the beginning of each line. We can get around this by using a much-neglected KERNAL routine called PLOT ($FFF0). PLOT can either read the current cursor position or move the cursor to a new position . In this case we want to move the cursor. To indicate this we must clear the carry (CLC) and set up the new cursor position in the X and Y registers before calling PLOT. The **row** no. goes in **X** and the **column** no. in **Y**. Row and column are measured starting at 0,0 in the upper left-hand corner of the screen.

The print subroutine expects its data to follow a specific format. The first two characters are the row and column numbers, respectively, for PLOT. Next comes the text of the string to be printed. It can be of any length and contain any characters except $00 bytes. The end of the string is indicated by a $00 byte, which is not printed. After printing the first string, the routine will automatically go on and print any other strings that follow the first in memory. Each string must obey the same format as the first: PLOT row and column; string text; $00 byte. To terminate the printing process, you must place an $FF byte **after the $00 byte** of the last string. $FF bytes in the text of the string are printed normally (in case you want to print the pi symbol, which is $FF in ASCII).

The entire menu screen is printed with one call to the print routine. At $8051 in the main menu routine, we set up the X and Y registers to point to the beginning of the menu screen text ($8180). The **low** byte of the address must be put in **X** and the **high** byte in **Y**. When we call the print routine, it immediately stores this pointer in $0061-62 (BASIC floating point accumulator #1). Note that the print routine uses a subroutine at $80DC to increment this pointer as the string is processed. You'll need this little subroutine if you use the print routine in other programs.

Back in the main menu routine, after printing the screen, the CHRIN routine is used at $8058 to input a selection number from the user. This KERNAL routine conveniently handles all the cursor movement, INST/DEL, RETURN, etc. keys for us. The FIRST character typed by the user is returned in the accumulator when the user presses RETURN (this means "01" is interpreted as "0"). Then the character entered is checked against the range of allowable values. First the character is exclusive-or'ed with the value $30. This converts the high nybble into a 0 if and ONLY if it was a 3 originally, i.e. it converts values in the range $30-39 (ASCII "0"-"9") to $00-09. The result is compared with the value in location $80FE. This location contains the highest selection number allowed, **PLUS 1**. Only selections in the range from $00 up to but not including this value are allowed.

Once an allowable selection is made, it is pushed onto the stack to save it temporarily. Next, a routine at $8076 is called to download a special routine to RAM at $02A7. Then the selection number is pulled back off the stack and the selected program is downloaded to RAM using the $8076 routine. Finally, the menu program ends by jumping to $02A7, the special RAM routine. This routine is very important and requires a bit of explanation. We can't just jump to the selected program as soon as it is downloaded. While this would be okay for some programs, other programs may need the BASIC ROM or the RAM at $8000-9FFF available. In general, we want to let the user switch the cartridge off before the program is executed. This is "poor man's bank-switching" - it lets us use the much less expensive standard cartridges where we might otherwise need a bank-switched cartridge. Of course, you need a cartridge expansion board or equivalent to switch off the cartridge, but you should have one anyway if you're working with cartridges.

The special routine at $02A7 gives the user a chance to turn off the cartridge. This routine waits for the user to hit RETURN, checks that the cartridge is actually off, and then executes the downloaded program. The special routine must be in RAM since it has to function after the cartridge has been turned off. Also, the routine must be modified each time so it will jump to the correct address for the particular program that was downloaded. The download routine at $8076 is used once to install this special routine in RAM and then again to download the selected program. The download routine automatically inserts the execution address for the downloaded program into the special RAM routine.

The download routine gets its information from the DOWNLOAD PARAMETER TABLE at 8100. Each menu selection has an entry in this table. Each entry consists of 4 two-byte pointers, each stored in standard lo-byte/hi-byte order. The first pointer of an entry indicates the starting location of the selected program **within the 16K cartridge area**, and the second pointer indicates the ending location within the cartridge **plus 1**. The third pointer indicates where the program normally resides in memory, i.e., the start of the area to download the program to. The last pointer is the execution address. The download routine automatically inserts this address into the special RAM routine at $02C0-C1, where it is used eventually to start the downloaded program. To use the download routine, you simply put the selection number in the accumulator and call the routine. It picks out the correct entry from the download table and does the rest.

Out of all this, there are only a couple things you need to know if you want to set up cartridges yourself with the bare "8K/16K MENU" program rather than use the "MENU MAKER" program. Unless you want to undertake major surgery on the menu program, about all you can do easily is change the appearance of the menu. Of course, you'll also have to set up the other functions of the menu program corrrectly. Here's a checklist of the steps you have to take to set up a cartridge:

1) Collect the programs you wish to put on cartridge. Write down the number of bytes each requires.

2) Load in your monitor. Micromon is recommended for its ability to load a file into any area of memory. We also recommend that you locate it out of the way up at $C000.

3) Load the "8K/16K MENU" program into a suitable area, such as $2000.

4) Design a menu screen and store it into the menu program starting at $8180 (or $2180 if you're assembling the cartridge at $2000). Remember that each string to be printed must be preceded by the row and column number for PLOT, and followed by a $00 byte. Put an $FF byte after the $00 of the last string. You must set up your menu first so you'll know where you can start storing the programs within the cartridge.

5) Record the number of choices that will be on your menu, **PLUS 1**. This value normally goes at $80FE, but if you're assembling your cartridge at $2000 then the corresponding address is $20FE.

6) Fill the cartridge area with $FF bytes.

7) Load each program into the cartridge area, one by one. Jot down the starting and ending addresses of each program **in the cartridge**, and also its **normal** load and execution addresses. If you have room, you might want to start each program at the beginning of a page (e.g. $8500) rather than cramming them all together as the MENU MAKER program does.

8) If you are using any BASIC programs, you'll have to tack the BASIC boot routine from MENU MAKER on the end. Examine a cartridge prepared by MENU MAKER to see how the BASIC boot routine is modified for a particular program. Then make the corresponding changes for your program. The location of the BASIC boot routine should be used as the execution address of the BASIC program. Don't forget to count the boot routine when determining the ending address of the program in the cartridge.

9) When all the programs have been loaded into the cartridge area, fill in the download table with the parameters for each program. Remember to **ADD 1** to the ending address within the cartridge for each program.

10) That's it - save the cartridge file to disk. Follow the directions above for burning the EPROMs and setting up the cartridge board.

We hope you have a lot of fun with the menu utilities from this chapter. We want you to get the most from your PROMENADE, and we hope they help you do it.

DOWNLOAD/RUN ROUTINE

The program disk with this book includes a special DOWNLOAD/RUN routine, courtesy of Jason-Ranheim. The routine is also included on the PROMOS disk supplied with the PROMENADE. The DOWNLOAD/RUN routine (DL/RUN for short) is for use with the PC4 bank-switch cartridge, available from CSM. The PC4 allows you to put up to 4 EPROMs on one board, with each EPROM allowed to be up to 64K in size. This gives you a potential of 256K on a cartridge the size of a standard game cartridge! (it even fits in a standard case).

The DL/RUN routine lets you put a BASIC program up to 38K long on a PC4 board. The 38K limit is the maximum size for BASIC programs on the C64. Besides BASIC programs, this routine can be used with any machine language program that resides in the BASIC area and has a BASIC boot statement at the beginning (such as SYS 2061). When you RESET or power up the computer, the cartridge will start itself automatically. The DL/RUN routine will download the program from the cartridge into the BASIC area. Then the routine will switch off the cartridge, using the PC4's internal bank-switch register (BSR). This allows the 8K of RAM under the cartridge to be used by the program, if required. Finally, the program will be executed with a BASIC RUN command.

To set up a cartridge with the DL/RUN routine, you must first determine roughly how long your program is. A good estimate can be obtained by taking the number of disk blocks the program occupies and dividing it by 4. This will give you the approximate size of the program in **K** (1K = 1024 bytes = 4 disk blocks). Using this figure, you can choose the size and number of EPROMs you'll need. The DL/RUN routine allows you use any combination of the following chips to provide the amount of memory you need:

CHIP	SIZE
2764	8K
27128	16K
27256	32K

For instance, if your program is 21K long (approx. 84 disk blocks) you could use just one 32K chip, or one 16K chip and one 8K chip, or three 8K chips.

Once you've selected the chips, you need to load PROMOS. If your BASIC program is less than 30K (121 disk blocks), you can use the PROMOS/HESMON cartridge (available from CSM). If you are using the cartridge, skip the following directions on loading PROMOS. Just insert your cartridge, power up and enter PROMOS with

```
SYS 9*4096     (or SYS 36864)
```

If you don't have a PROMOS cartridge or your program is longer than 30K, you will have to use the disk version of PROMOS. This version automatically copies itself up to the end of the BASIC area when you run it. In our situation, we want to put PROMOS up at the end of the $C000-CFFF area to get it out of the way. We can fool PROMOS into relocating itself up there if we temporarily change the end of BASIC pointer stored in locations 55-56 ($37-38). First load in PROMOS from disk, but DON'T run it yet.

Next, change the end of BASIC pointer:

```
POKE 55, 0      (RETURN)
POKE 56, 208    (RETURN)
```

This sets the end of the BASIC area to $D000 (208 = $D0). Now run PROMOS:

```
RUN
```

You should see the PROMOS startup message appear. Next we have to set the end of BASIC area back to its normal position at $A000 ($A0 = 160):

```
POKE 55, 0
POKE 56, 160
```

Once PROMOS is ready (disk or cartridge version), you need to load in the DL/RUN routine. Insert the program disk into the drive and type

```
LOAD "DOWNLOAD/RUN", 8, 1
```

The routine will load in at $C000 (49152). To reset the BASIC pointers and avoid "OUT OF MEMORY" errors in the next steps, you must do a BASIC NEW command:

```
NEW
```

Now you're ready for the BASIC program. Insert the disk and load the program in as usual:

```
LOAD "program", 8
```

The DL/RUN routine needs to know where the end of the BASIC program is so it can tell when to stop downloading the program from the cartridge. After a load, BASIC automatically sets up a pointer to indicate the end of the program. This pointer is stored at 45-46 ($2D-2E). We'll take the information stored there and pass it along to the DL/RUN routine:

```
POKE 49161, PEEK(45)
POKE 49162, PEEK(46)
```

We'll also need this information ourselves later on. Like all pointers, this pointer is stored with the low byte first (at location 45) and the high byte second (at 46). What we need is the actual location of the end of the BASIC program in decimal, which can be calculated by the following formula:

```
PRINT PEEK(45) + 256*PEEK(46)
```

Jot down the result and save it for later.

Before you burn the program onto your EPROMs, you should consult the chart below for the proper **CW** (Control Word) and **PMW** (Program Method Word) for the chips you will use. These values are required by PROMOS. If you don't use the correct CW the EPROM may be permanently damaged! The PMW is not as critical; the values given below are just suggestions. For more information on CWs and PMWs, see the chapter on PROMOS commands.

CHIP	SIZE	BYTES	CW	PMW
2764	8K	8192	5	7
27128	16K	16384	5	6
27256	32K	32768	230	6

Once you've determined the correct CWs and PMWs, you're ready to burn the EPROMs. Always start by initializing the PROMENADE with the zero command:

```
Z    (RETURN)
```

Insert the first EPROM into the PROMENADE and flip the lever down to lock it in place. Make sure that the notch on the chip is to the left. The first chip will hold the DL/RUN routine and the first segment of the BASIC program. The following command will burn the DL/RUN routine onto the chip:

```
PI 49152, 49327, 0, CW, PMW    (RETURN)
```

Don't type the letters "PI". Use the pi key instead (shift-up arrow, next to RESTORE). In place of CW and PMW, be sure to substitute the proper values for the type of chip you are using, from the chart above.

The red SKT and yellow PGM lights will come on briefly while the chip is being programmed. When it is finished, you're ready to burn the first segment of the BASIC program onto the rest of the first EPROM. The BASIC program starts in the computer's memory at 2049 ($0801). We have to start putting it on the chip just past the end of the DL/RUN routine, at byte 176 (bytes in the EPROM are numbered starting at 0). Depending on the size and type of your first EPROM, you will have to use a different form of the following command:

```
PI 2049, AAAAA, 176, CW, PMW
```

You must use the proper CW and PMW, as given in the chart above. You must also substitute a different number for AAAAA according to the following chart:

CHIP	AAAAA	BYTES
2764	10064	8192
27128	18256	16384
27256	34640	32768

This command will completely fill the first EPROM. Remove it from the PROMENADE and label it so it won't get mixed up with any other chips.

From your initial estimate of the size of the program in disk blocks or K, you should know how many chips you have left to do (if any). You can tell more exactly by using the value you calculated earlier for the end of the BASIC program (from the pointer at 45-46). If the number given for AAAAA is greater than or equal to the end of the BASIC program, you're done already! Your first chip was large enough to hold the whole program.

If the entire BASIC program did NOT fit on your first chip, then you'll need to continue burning the program onto a second chip. You must start programming the second chip from the same point in the BASIC program that you left off at in the first chip. Use the following command:

PI AAAAA+1, BBBBB, 0, CW, PMW

You must fill in the correct values for AAAAA+1 and BBBBB. The value for AAAAA+1 is found by adding 1 to the value you used for AAAAA in the previous command (as if you hadn't guessed). The value for BBBBB is calculated by adding the size of the SECOND chip in BYTES to the number you used for AAAAA in the previous command. The size of the different types of chips in bytes is given in the chart above. Of course, you also have to fill in the correct values for the CW and PMW based on the type of the second chip.

Once again, if the number you used for BBBBB is greater than or equal to the end of the BASIC program (from 45-46), you're finished. If there is more to do, just continue the process. The third chip will require the command:

PI BBBBB+1, CCCCC, 0, CW, PMW

CCCCC is calculated by adding the size of the THIRD chip in bytes to the value you used for BBBBB. If a fourth chip is required, use:

PI CCCCC+1, DDDDD, 0, CW, PMW

By this time, you should be able to figure out what values to fill in this command. Since four chips is all you can put on the PC4 board, you HAVE to be done by now!

When you're done programming the chips, all you have left to do is insert them in the PC4 board. The four sockets on the board are labeled from 0 to 3. Socket #0 is the closest to the computer when the cartridge is plugged in. The first EPROM must be put in socket #0, the second EPROM in socket#1, etc. Insert each chip into its socket so that the notch on the chip will be to the LEFT when the cartridge is plugged into the computer. There is a corresponding notch or dot on the socket to help you. Since board designs might vary, ALWAYS match up the notch on the chip with the mark on the socket, even if this position is different from what we've given you. Likewise for the placement of socket #0, etc.; always follow the markings on the board or the instructions that come with it.

You should use a plastic housing for the cartridge to protect it and make it easier to insert. Place the cartridge, with the EPROM side up, in the bottom half of the housing. This is the half with the plastic pins sticking up. The cartridge board has two notches on each side that will fit right around the plastic pins and hold the board securely in place. Now snap the top half of the case on. Once you do this, it may be difficult to open the case again. If you wish, cut a little off the end of each plastic pin to make opening the case easier. Don't cut more than about 1/16" off the pins or the case will come open too easily.

If you mix different types of chips in your cartridge, you'll have to use our procedure to calculate the areas of the BASIC program for each chip. Many times, however, you'll find yourself using only one type of chip in the cartridge. In those cases, you can consult the following chart for the correct beginning and ending addresses, without having to do any calculations. Note that the first EPROM must also hold the DL/RUN routine, which is not shown in the chart. The first part of the BASIC program always starts at 2049 in the computer. It is put onto the first EPROM starting at byte 176 of the EPROM, to leave room for the DL/RUN routine. All of the other parts of the program are put on EPROM starting at byte 0 of the chip.

EPROM	2764 (8K)	27128 (16K)	27256 (32K)
First	2049, 10064	2049, 18256	2049, 34640
Second	10065, 18256	18257, 34640	34641, 40959 *
Third	18257, 26448	34641, 40959 *	not needed
Fourth	26449, 34640	not needed	" "

* The end address shown is the end of the BASIC area. If this last area is required, you may use a 2764 (8K) for it.

CUSTOMIZING THE KERNAL

Everybody has their own list of changes they'd like to make to their Commodore 64. Maybe you don't like the screen colors, or you're tired of typing ",8" every time you load a program, or you'd like the screen to say "WELCOME TO BOB'S COMPUTER" when you turn the computer on. These and other changes can be accomplished easily. You'll need an EPROM programmer, a 2764 EPROM and a 24/28-pin adapter. A reference book such as "The Anatomy of the Commodore 64" is also immensely helpful, especially if you want to go beyond the modifications we present here.

To start off with, you should get a copy of revision 3 of the KERNAL if you don't already have it (SX-64 owners can use their present KERNAL). You could proceed by copying the KERNAL down to RAM memory at, say, $4000-5FFF, but then you'll have to translate addresses back and forth. A better way is to copy the KERNAL ROM down to the RAM underneath it and switch out the ROM. You can run your system from the RAM copy as long as you don't try to modify a system routine as it is being executed (such as the screen editor and IRQ routines). This way you'll be able to test your changes as you make them. You'll have to copy the BASIC ROM down to RAM as well, since BASIC gets switched off with the KERNAL. You won't be able to use a cartridge based monitor either, since cartridges are also switched out. Of course, you should save your custom KERNAL to disk from time to time as a precaution, but this method works well for us.

Start by copying BASIC and the KERNAL down to RAM with T A000 BFFF A000 (BASIC) and T E000 FFFF E000 (KERNAL). Now switch off the ROMs by changing location $0001 to a $35. You shouldn't notice any change, but you're now operating out of RAM. Now change location $FDD6 to $E5 (be sure and change it back to **$E7** before burning an EPROM copy!). This change prevents the RESET routine from switching the ROMs back in when you do a 'soft' RESET (SYS 64738 or G FCE2). Since many modifications will change the startup defaults, you'll need a soft RESET to test them. A 'hard' RESET with a RESET button will still switch the ROMs back in, but it won't wipe out your RAM copy. Try a soft RESET now with G FCE2. After an extra-long pause you will get the familiar startup screen, only with 51216 bytes free! Relax, this just means the RAM test routine got all the way to the VIC control register at $D011 before finding any 'non-RAM'.

Figure 14-1 lists some short patches you can make, just for starters. Let's take a brief look at each of them, starting with everyone's favorite, the default colors. The default border and background colors are located in a table of values that get written into the VIC chip on powerup, RESET or RUN/STOP-RESTORE by the VIC initialization routine at $E5AA. The border and background color codes are located at ECD9 & ECDA, respectively. These colors are specified using the color RAM codes ($00=black, $01=white, etc.) NOT the ASCII print commands ($90=black etc.). The default character color is located at $E535, in the screen initialization routine (which calls the VIC chip init routine). The character color is also specified using color RAM codes.

Next on our list is a patch to automatically enable repeating for all keys. To do this we'll have to insert a JSR in the screen init routine and put a short piece of code in the unused area at $E4B7-D2. This simply stores an $80 into the key repeat flag and returns. While you're at it, you might want to adjust the delay before a key repeats and the speed at which it repeats (note two different locations to change speed). The new values given are just suggestions, so experiment.

Now we come to a real handy pair of changes. First, we modify the BASIC LOAD and SAVE commands so they automatically go to disk rather than tape when no device is specified (i.e. LOAD "PROG"). If you need to specify a secondary address, though, you'll still have to type the device number (LOAD "PROG",8,1). Likewise, you can default the OPEN command to go to the printer if no device is specified. You'll still have to supply the device number if you need a secondary address. Neither of these changes affect the corresponding KERNAL routines, since you must always specify the device number for them (unless you want to get really tricky).

Our next modification is to change the SHIFT-RUN/STOP (Sh-R/S) combination so that it loads the first program off the disk and runs it. To do this, you must first change the default LOAD device as given above. The Sh-R/S combination works by putting its command into the keyboard buffer and then jumping to the keyboard interpret loop. Since the keyboard buffer is only 10 characters long, this limits our possible modifications. To fit in a LOAD ":*" (return) RUN (return) command, we have to use the shifted shortcuts, e.g. R shift-U for RUN, shown as rU. The only way to get around this limitation is to patch into the keyboard interpret loop at $E5EE, and have it perform your custom command directly.

Our final modification involves the startup screen, which is printed in three separate parts by a routine at $E422-46. The first part, up to "64K RAM SYSTEM ", is stored as a table of ASCII characters at $E473-AB. The number of bytes free is calculated and printed next, and then the "BASIC BYTES FREE" message at $E45F-72 is printed. The first part printed can be altered by just changing the ASCII codes. To disable the bytes free and reuse its message area, make the last two modifications shown. You can then continue your screen message in the "BASIC BYTES FREE" area.

If you want to go much beyond the modifications here, you'll probably need some extra space to put code in. The only unused area left is at $E4B7-D2, but you could free up some space (nearly 2K!) if you don't need the cassette. Once you've made your changes, burn the code into a 2764 EPROM and install it in the computer using the AD adapter. Powerup and take your new computer for a test drive!

Figure 14-1: KERNAL MODIFICATIONS

LOCATION	ORIGINAL	MODIFIED	DESCRIPTION OF CHANGE
FDD6	E7	E5 see text	**CHANGE TO $E7 BEFORE BURNING EPROM!**
ECD9-DA	0E 06	Your choice	Default border and background colors
E535	0E 06	Your choice	Default character color
E536-8	STA $0286	JSR $E4B7	Call key repeat routine
E4B7-BF	AA ... . AA (filler)	STA $0286 LDA #$80 STA $028A RTS	Set character color Value so all keys repeat Save in key repeat flag Return to screen init routine
E53A/EB1D	04	03	Key repeat speed - note 2 places to change
EAEA	10	08	Key repeat delay
E1DA	01 (tape)	08 (disk)	Change LOAD/SAVE default device number
E228	01 (tape)	04 (printer)	Change OPEN default device number
E5EF	09	0A	No. characters in SHIFT-R/S command
E5F4-F5	E6 EC	BF E4	Location of command
E4C0-C9	AA ... AA	4C CF 22 3A 2A 22 0D 52 D5 0D	New version of command 10":*" (return) rU (return)
E473-AB	"**** COM.."	Your choice	Change startup message
E45F-72	"BYTES FREE"	Your choice	Reuse bytes free space
E430-32	LDA $37; SEC	JMP $E43D	Disable no. bytes free calc/print

REUSING THE TAPE AREAS IN THE KERNAL

The KERNAL ROM contains 8K of machine language routines. These routines handle all communication with the computer's peripheral devices: keyboard, screen, serial (disk or printer), RS232 and cassette tape. It may be hard to believe, but the cassette tape routines take up more ROM space than any other single function of the KERNAL. In fact, nearly 2K of the 8K KERNAL is dedicated to the tape routines! This may seem like a tremendous waste of space, since most C64 owners have disk drives and rarely if ever use tape. If you have a PROMENADE, you can dispense with the tape routines and install some other utilities in their place. Possible replacements include a DOS wedge, small ML monitor, simple track/sector editor, BASIC programming aid or even a fast load utility. As a sample of the possibilities, the disk accompanying this book includes a modified KERNAL with the DOS wedge installed.

You'll have to make sure your replacement routines will actually fit in the space available, of course. You'll probably also have to alter them so they will execute at the new location. Most ML programs are not relocatable as is; they are written to reside at a particular place in memory. If the program is one you have written yourself, you may be able to make the necessary changes without much trouble. On the other hand, if the program to be installed in the KERNAL is a public domain or commercial product, relocating it could be difficult to accomplish. Obviously, we can't cover all the different programs in this book that you might want to install in the KERNAL. All we can do is offer some general guidelines on relocating programs and point out the KERNAL areas that can be reused.

There are several considerations when modifying a program for its new location in the KERNAL. First, you must alter memory references so that they refer to the new area rather than the original area. Not all memory references will need to be altered, however. Only references to locations within the original program area should be altered. References to areas outside the original program area, such as KERNAL subroutines or system RAM locations, must stay the same.

References to locations within the original program area could occur as absolute, indexed or indirect references. Absolute references are easy to spot since they are always three-byte instructions, e.g. JSR $1234. Indexed instructions are also usually three byte instructions, such as LDA $1100,X. Indirect references may be three bytes, such as JMP ($2000) or two-bytes, such as LDA ($FB),Y - which is a combination of indexed and indirect. Indirect references will be the hardest to change, since they use the contents of some location (a vector) to specify the actual address. You must change the vector contents in order to adjust the instruction for a new location. Many times there is a section of code in the program which stores the correct contents into the vector. You must change this code too.

Branch instructions will not need to be modified, since they are all relative instructions. That is, the target address (location to branch to) is always found by adding or subtracting a certain number of bytes (the offset) to the location of the branch instruction itself. The actual target address is NOT stored directly, just the offset amount. As long as you don't insert or delete any code in between the branch instruction and its target address, the offset will be the same no matter where in memory the program code resides.

Another consideration when putting a program in the KERNAL is whether it alters itself as it runs. Many programs store values into instructions or data tables within themselves. In such a case, you must relocate the code or table outside the KERNAL area, since the KERNAL is in ROM, not RAM, and cannot alter itself. There are several areas in system RAM (below $0400) which are used only by the tape routines, and these areas can be reused when the tape routines are removed. A list of these locations is given in Figure 15-2. Unfortunately, these areas don't amount to much space. There are other, larger areas in memory that are "unused", such as $02A7-02FF and $0334-03FF (which includes the cassette buffer). So many commercial programs use these areas, however that incompatibility with some programs is certain. You may have better success with the area at $07E8-07FF, which is much less commonly used. Programs with sprites may use this area, so watch out.

Finally, you must also consider how you want to start up your program. You could just use a SYS call to start it when needed, or you could patch into the RESET process so that your program is started automatically. This would be especially suitable for programs such as a DOS wedge or a BASIC programming aid. Even better, you might check a particular key during RESET and only start your program if the key was being held down. This would be very useful when combined with a monitor program. You should always make sure there is a way to exit your program back to normal operation, in case your program is incompatible with other programs. Alternatively, you could put your modified KERNAL on a 16K EPROM with the original KERNAL. As described earlier in the special cartridge hardware chapter, you could go from one KERNAL version to the other with a simple flick of a switch.

There are tools available to help you relocate a program from one area to another. Most good monitors have a special relocate command, usually called "N" for "New location". Both the MICROMON monitor on the program disk and the commercial product HESMON have the "N" command. It is comparable to a RENUMBER command for BASIC programs. It allows you to automatically adjust all three-byte instructions within the program by adding an offset to the address referenced by the instruction. Many programs use tables of two-byte vectors which will also need to be adjusted. The "N" command has an option to handle vector tables too.

An example may help clarify the "N" command. Suppose we have a program which normally executes at $2000-2FFF and we want to move it up to $6000-6FFF. First we must copy the original version up to the new location with the transfer command: T 2000 2FFF 6000. Now we must adjust the memory references in the new version. Since we moved the code up $4000 bytes in memory ($2000 + $4000 = $6000), we would use the following command: N 6000 6FFF 4000 2000 2FFF. Put simply, the first two parameters (6000 6FFF) specify where the new copy is located, the third parameter (4000) specifies how far the code has been moved, and the last two parameters (2000 2FFF) specify where the code was located originally.

Some of these parameters may seem redundant, but they are there for more general purposes. The first two parameters actually define the area of **program code** for the "N" command to search ($6000-6FFF). It will not look outside this area, period. The last two parameters tell it what range of **address references** to look for ($2000-2FFF). It will ignore any reference to an address outside this range. This prevents it from altering calls to KERNAL subroutines, for instance (since we didn't move the KERNAL, those references must stay the same). The middle parameter ($4000) is the **offset** to **add** to the references that meet the other conditions.

Suppose now that we wanted to move the code the other direction, i.e. from $6000 down to $2000. What offset would we use? If you said -$4000, you're close. We can't use negative numbers directly, however. What we have to do is use a large enough offset so that when it's added to $6000, the result "wraps around" past $FFFF and ends up at $2000. The correct value to use is $C000, since $6000 + $C000 = $012000, which becomes $2000 when the highest byte is dropped. The easiest way to calculate offsets is to use the subtract command "-" (available in both HESMON and MICROMON). Always take the **new** location and subtract the **original** location from it (NEW - OLD), regardless of which direction you are moving in memory. The subtract command requires that the "-" come before the numbers to be subtracted, rather than between them, so the general form is: **- NEW OLD.** In our first case, moving up from $2000 to $6000, we would use **- 6000 2000** and get the answer $4000. In the second case we would use **- 2000 6000** and get the answer $C000.

One final word about adjusting vector tables with the "N" command. If we want to adjust a table of vectors rather than program code, we must put a "W" at the end of the "N" command. "W" stands for "Words", meaning two-byte values. Each vector must be a two-byte value, in standard lo-byte / hi-byte order. Let's go back to our first example, moving a program from $2000 up to $6000. Suppose the original program contains ML code at $2000-2DFF and a vector table at $2E00-2FFF. In the relocated version the code would be at $6000-6DFF and the vectors at $6E00-6FFF. The proper sequence of commands to use would be:

```
T 2000 2FFF 6000              Transfer the program
N 6000 6DFF 4000 2000 2FFF    Adjust the ML code
N 6E00 6FFF 4000 2000 2FFF W  Adjust the vectors
```

Note that in the two "N" commands we used different areas to search (first two parameters) but the same range of references to look for (fourth and fifth parameters). The offset stays the same too. What could be simpler, you say.

An even more powerful tool than a monitor for relocating programs is a type of program called a SYMBOLIC DISASSEMBLER. These disassemblers do more than a monitor's disassemble command, because they produce an output file suitable for use with an assembler package. They are called SYMBOLIC disassemblers because they invent symbolic labels for memory locations, rather than specifying the location directly. This means you can change the definition of the label at the beginning of the program, and all references to that label throughout the program will use the new value automatically when the program is reassembled. This makes it easy to move an entire program to a new location or even rearrange the parts of the program relative to each other.

One excellent utility of this type is THE SOURCE GENERATOR (available from CSM Software). This program can produce assembler files suitable for several different assembler packages, including Commodore's. It can also produce a CROSS-REFERENCE listing for the program being disassembled, showing every location referred to by the program and where in the program each location is referred to. This is tremendously useful for many different purposes. When relocating a program, for instance, you can use this feature to verify that all of the program's references to itself were actually changed as desired, and that other references were not changed. A symbolic disassembler was used to research the KERNAL and DOS wedge for this chapter. A complete assembly listing of the KERNAL was prepared and cross-referenced (the specifications for doing this are included on disk with the package). By examining the cross-reference, the areas of the KERNAL that are used only by the tape routines were identified, as well as the system RAM locations used only by tape.

In order to reuse the tape areas, we must make a few minor modifications to the KERNAL. After all, there is no longer any way to access tape, so we must intercept all references to it. Specifically, any attempt to access device 1 (tape) should produce an "ILLEGAL DEVICE" error. This is accomplished by modifying the LOAD, SAVE and OPEN routines. If device 1 is referenced, we force the routine to jump to $F713, which returns the "ILLEGAL DEVICE" error. You may also wish to change the default device for LOAD, SAVE and OPEN to device 8 rather than device 1 (see the chapter on KERNAL modifications). It is especially important to prevent the OPEN routine from setting up a file to device 1, because other routines such as CHKIN (prepare for input) and CHKOUT (prepare for output) depend on the parameters set up by OPEN. If you prevent a file from ever being OPEN'ed to device 1, then the other routines such as CHKIN and CHKOUT will never try to access that device either.

Figure 15-1: KERNAL TAPE AREAS

MEM. AREA	BYTES	DESCRIPTION		PATCH NEEDED
EA61-EA7A	26	IRQ - Check tape keys		EA61 JMP $EA7B
F0D8-F105	46	Tape messages		
F179-F1AC	52	Tape CHRIN routine		
F1E5-F1FB	23	Tape CHROUT routine		
F22A-F232	9	Tape CHKIN check	**	
F26F-F274	6	Tape CHKOUT check		
F2C8-F2ED	38	Tape CLOSE routine		
F38B-F3D2	72	Tape OPEN routine		F38B JMP $F713
F533-F5A8	118	Tape LOAD routine		F4B6 BCC $F4AF
F659-F68D	53	Tape SAVE routine		F5F8 BCC $F5F1
F72C-FCD0	1444	Main tape routines	**	F617 JSR $F22A
FD5F-FD66	8	Init. buffer pointer		NOP's or own code
FD9B-FDA2	8	Tape IRQ vectors		
	=1903			

** Transfer $FB8E-FB96 to $F22A-F232!

Figure 15-2: TAPE RAM LOCATIONS

$92	
96	
9B	
9F	
A6	
B0-B3	See $FD5F above
BE	
029F-02A0	

Figure 15-1 lists the complete set of areas used by the tape routines. The main set of tape routines is located from $F72C to $FCD0, which includes about 1500 bytes. Several smaller areas are also available, which can be used to store subroutines or data tables for your program. As explained above, some areas require that a patch be inserted in the KERNAL if they are to be used. These patches prevent the KERNAL from ever calling the routines you have removed from these areas. The best way to modify the KERNAL is to copy it down to RAM at $2000 and make your modifications to the RAM version. We suggest that you fill the unused areas of your modified KERNAL with some value such as $FF so that later on you can spot these areas easily. Then insert the corresponding patch if one is given. Make your modifications in the order given.

One patch in particular requires some explanation. At $FB8E-96, smack in the middle of the main 1500-byte block of tape code, are nine bytes of code that are also used by the serial bus SAVE routine. This breaks the block of reusable code into two pieces. While this may be acceptable in many cases, in other cases you may need an uninterrupted block of code. We can arrange this by moving the nine bytes of code somewhere else in the KERNAL and changing the call that references them. One of the small tape areas available, at $F22A in CHKIN, is exactly nine bytes long. Transfer the piece of code from the tape block to this area with your monitor (T FB8E FB96 F22A). Do this BEFORE you fill the main tape block with $FF's, of course. Then change the JSR $FB8E at $F617 to JSR $F22A. There is a slight risk of making your KERNAL incompatible with some programs by doing this, but it seems extremely remote.

Several RAM locations are also made available when you abandon the KERNAL tape routines. The complete list is given in Figure 15-2. Having extra zero page locations is especially useful. Note that the cassette buffer pointer at $B2-B3 is normally set to point to $033C by the KERNAL RAMTAS routine ($FF50). We have you disable this function by inserting NOP's at $FD5F-66. Again there is a very slight possibility of incompatibility when you do this.

There are other two areas not listed which only the tape routines refer to directly: $0100-013E and $033C-03FB. These areas may not really be of much use since so many commercial programs already use them. HESMON uses some of the $0100-3E area (the bottom end of the stack), for instance, and many protection schemes use the $033C-FB area (the cassette buffer). If the program you install in the KERNAL uses these areas, you should make sure there is a way to disable your program or switch off your modified KERNAL entirely. The DOS wedge installed in our sample KERNAL uses part of the cassette buffer for data storage ($0337-8A). If the wedge interferes with a program you wish to use, simply disable it with the Quit command @Q.

Our modified KERNAL with the DOS wedge is in the file called "KERNAL/WEDGE" on the program disk with this book. The tape routines were removed according to the table above, and the wedge was installed at $F800-FB8B in the main tape block. The wedge is patched into the BASIC cold start routine at $E39A, and into the NMI routine at $FE6F. This means that the wedge will be activated by either a RESET or RUN/STOP-RESTORE sequence. The file loads in at $2000-3FFF to make burning it onto EPROM easier. The PROMOS command to use for a standard 2764 (8K) EPROM is:

```
PI 8192, 16383, 0, 5, 7
```

That about wraps it up for this subject. If you absolutely have to have more room in the KERNAL, the RS232 routines are about the only other nonessential code you can get rid of. These amount to less than 1K, with the biggest block at $EEBB-FOBC being about 1/2 K. If you wish to pursue this, we recommend using a symbolic disassembler with cross-reference capability as discussed above. Good luck!

MODIFYING THE 1541 DOS

In this chapter we'll look at some of the important things to know if you want to modify the DOS ROMs in your 1541. The 1541 DOS is contained on two 8K ROMS. One ROM resides at $C000-DFFF in memory and the other at $E000-FFFF. For convenience we'll call them the C-D and E-F ROMs, respectively.

Any time you make modifications to either ROM, you must adjust or disable the **ROM checksum.** On a powerup, RESET, UJ or UI command each ROM is checksummed separately by adding up the bytes in it, using ADC, to get a one-byte sum. This sum must equal $E0 for the E-F ROM and $C0 for the C-D ROM. If the checksum is not correct, the drive goes into an endless loop to blink the red error light. By the way, you can tell roughly where the error was by watching the light. The light will blink a number of times, pause, then repeat the pattern. If it blinks twice between pauses, the error was in the E-F ROM; if it blinks three times between pauses, the error was in the C-D ROM (blinking only once bewteen pauses would mean there was an error in testing zero page RAM; four times would mean an error while testing the rest of RAM).

Any change in a ROM's ML code will change that ROM's checksum. The checksum test can be defeated entirely by simply putting NOP's in place of a couple of branch statements. The code for the ROM checksum is in the E-F ROM at $EAC9-E9. The locations to change are given figure 16-1. If your ROM copy is in the C64 starting at $2000, the number in parentheses tells where the corresponding location will be in C64 memory.

Figure 16-1: DISABLE ROM CHECKSUM

LOCATION	(C64 LOC.)	CONTENTS	CHANGE TO
$EAE4-E5	($2AE4-E5)	$D0 39	$EA EA
EAE8-E9	($2AE8-E9)	D0 DF	EA EA

Note that the checksum routine for **both** ROMs is in the E-F ROM. If you want to disable the checksum test, you'll have to modify the E-F ROM even if all your other changes are in the C-D ROM only. Fortunately, you don't HAVE to disable the routine. By simply adjusting a byte in the C-D ROM, you can make the checksum come out to the proper value. You only need to adjust one byte, and any byte will do. The very first byte of the C-D ROM (at $C000) is a scrap byte that Commodore itself uses for this purpose. However, some programs check the value of this byte as a way of deciding whether the drive is a 1541 or not. Other programs may check this byte to see if you have modified your DOS. It is probably best to use some other byte in the $C000-C0FF area, which is all unused filler bytes ($AA's). Now all you need to know is what value to put there.

The program disk with this book contains a program called "CHECKSUM DOS" (see figure 16-2). This routine will checksum your C-D ROM copy if it is located at $2000-3FFF in the computer. Start by loading the C-D ROM into this area. Next, change your scrap byte to $00 (this makes calculating its correct value easier - always do this!). Load the CHECKSUM DOS routine and execute it with **G 1000** from the monitor. The routine will calculate the current checksum and put it in memory location $FD.

Figure 16-2: CHECKSUM DOS ROUTINE

```
.1000  A9 20    LDA #$20      Hi byte, start of DOS copy
.1002  85 FC    STA $FC
.1004  A9 00    LDA #$00      Lo byte,  "  "  "  "  "  "
.1006  85 FB    STA $FB
.1008  A8       TAY           Zero index (and checksum)
.1009  A2 20    LDX #$20      No. pages to checksum
.100B  18       CLC           Start with carry clear
.100C  71 FB    ADC ($FB),Y   Add next byte to checksum
.100E  C8       INY
.100F  D0 FB    BNE $100C     Branch to do next byte
.1011  E6 FC    INC $FC       Next page
.1013  CA       DEX           Decrement page count
.1014  D0 F6    BNE $100C     Branch to do next page
.1016  69 00    ADC #$00      Add in last carry
.1018  85 FD    STA $FD       Store checksum
.101A  00       BRK
```

Suppose that the current checksum comes out to $B0 rather than $C0 (with your scrap byte set to $00!). Since the checksum is $10 too low, all you have to do is use $10 rather than $00 for the scrap byte. In other words, take the proper value $C0 and subtract the current value ($B0 in this case). The result will be the correct scrap byte value. You may use the MICROMON/HESMON subtract command to do the subtraction if you wish. Since the subtract command expects the values to be two bytes each, you have to put a $00 byte in front of each value:

```
- 00C0 00B0
```

The monitor will give the result $0010. Ignore the $00 byte and just use the $10. What happens if the current checksum is LARGER than $C0, say $D0? Do the subtraction in the same order (- 00C0 00D0). You'll get a strange looking result, $FFF0. The high byte of the result, $FF, is just a way of representing negative numbers (since you subtracted a larger value from a smaller one). When this happens, take the low byte of the result ($F0 in this case) and **subtract 1 from it.** Only do this if the high byte is $FF. This process will give you the correct scrap byte value (in this case, $EF). You should always check your scrap value by putting it in the ROM in place of the $00 and running the checksum again. It should give the result $C0 with the new scrap value. You can also use this procedure to adjust the checksum for the E-F ROM. Pick a byte from one of the unused areas in figure 16-3 for your scrap byte.

Now that we've tamed the mighty checksum, let's look at modifying the DOS itself. The first question is how much unused memory is available in the ROMs and where it's located. You'll need some space if you want to add new routines of your own. Even if all you want to do is modify existing routines, you'll probably need some space for patches. Figure 16-3 lists the major areas available in the ROMs. We assume that you have the latest revision of the E-F ROM (revision 5).

Figure 16-3: UNUSED DOS AREAS

LOCATION	SIZE	NOTES
$C000-FF	256	Filler bytes
E781-A1	33	Hardware utility loader - removed
FF2F-E5	183	Unused area for future DOS patches

Total bytes available = 472

The first area listed above is mostly filler bytes. You shouldn't change the first byte in this area unless you are only experimenting and don't care about compatibility with commercial software. The second area, $E781-A1, is where the hardware utility loader routine used to be located. This code was removed from revision 5 of the E-F ROM by Commodore. The routine was only activated if the clock and data lines were grounded on powerup. Its purpose was to load the first file on the disk into drive memory and then execute it. This feature is never activated in normal use. According to some authorities, it was removed to solve problems during initialization. The SOFTWARE utility loader, sometimes called the &-command, was left intact.

The last area, $FF2F-E5, was intentionally left unused by Commodore to reserve room for future patches. In fact, the area used to start at $FF01. A patch was made at $FF01 to allow the drive to be slowed down for the C64 or sped up slightly for the VIC-20 through the UI+ and UI- commands. Two other patches were added at $FF10 and $FF2F in revision 5 to solve some problems during initialization.

The 472 bytes may not seem like much room, but it should be enough for several small routines to be added. If you really want to experiment, you could rip out some of the normal routines that you don't need for your experiments. Probably the leading candidate in this regard would be the relative file routines, both because they are rarely used and because they take up a considerable amount of room. If you are planning anything this ambitious, we suggest you use a symbolic disassembler such as THE SOURCE GENERATOR (available from CSM) to create a cross-reference of the DOS. This will help you identify which routines are used only by the relative file commands.

To demonstrate modifying the DOS, we'll add a couple of features to the DOS that everyone can use. As you may know, there is a location in the 1541's memory that controls whether or not the drive will do a "bump" when it encounters an error. Location $6A (106), commonly called REVCNT, controls the bump feature. Many magazines have published a Memory-Write (M-W) command which disables the bump by setting REVCNT to 133 ($85). That change is only temporary, however. We can do much better.

REVCNT is initialized to $05 (binary %0000 0101) by a powerup, RESET, UJ or UI comand. Setting bit 7 of REVCNT to a 1 instead of a 0 will disable the bump. This change will not affect the bump required to format a disk. Since the normal value of REVCNT is $05, setting bit 7 to disable the bump gives us the value $85 (133) instead. REVCNT is initialized by the code at $EBD1-D4 in the UI command, which is called on powerup, etc. All we have to do is change the value placed in REVCNT to $85 and the bump will be automatically disabled. Specifically, **to disable the bump, change location $EBD2 from an $05 to an $85.** If your ROM copy is located at $2000 in the computer, this byte will be at $2BD2.

Remember, all we are doing is changing the value that is put in REVCNT during initialization. You can change this value back to the default value and re-enable the bump any time you wish. Just type in the following commands:

```
OPEN 15,8,15
PRINT#15, "M-W" CHR$(106) CHR$(0) CHR$(1) CHR$(5)
CLOSE 15
```

Why would you WANT to re-enable the bump? Well, some of the newer protection schemes may leave the head off-track after they are finished. When this happens, even turning the power off won't usually correct it. An IO command may get the head back on track, but more often a bump is required. You can use the M-W command above to re-enable the bump, but that command is not exactly convenient to remember. How about if we add a command to the DOS to enable (or disable) the head bump?

We'll do this through the USER command UO. The new commands will be:

```
UO+     Enable bump
UO-     Disable bump
```

The "U" commands are handled by a routine located at $CB5C in the DOS. This routine is reached through a vector in the E-F ROM. First we'll change the vector to point to an unused area ($FF30), where we'll put a special routine of our own. Whenever any U command is executed, the DOS will go to our routine first. Our routine will intercept the new commands and perform them. All other U commands will be passed along to the normal routines that handle them. Our routine is listed in figure 16-4. The routine is on the disk accompanying this book, under the name "BUMP COMMANDS".

The first thing our routine does is check that the U command is actually a U0 command. The command is stored in the INPUT BUFFER at $0200. A U0 command will have an ASCII "0" ($30) in the second byte ($0201). If the command isn't a U0, the routine immediately jumps to the normal code to handle the other U commands. If the command WAS a U0, we check to see that there were exactly three characters in the command (U0+ or U0-). Location $0274 in memory holds the length of the command. If the command is a U0 but it isn't exactly three characters long, the routine jumps to the normal U0 command.

Now we get the current value of REVCNT from location $6A. We want to change bit 7 of the value, but we must preserve the other bits (which are also used by the DOS). Before we change anything, though, we still have to check whether the third character in the command is a "+" or a "-". If it isn't either one, the routine jumps to the normal U0 command (for compatability). If the third character is a "-" (disable), we set bit 7 of the REVCNT value to a 1 by ORing it with the value $80. If the third character is a "+" (enable), we clear bit 7 of the REVCNT value to a 0 by ANDing it with the value $7F. Either way, we store the resulting value into location $6A. That's it. The routine terminates with an RTS.

Figure 16-4: BUMP COMMANDS ROUTINE

```
.FF30  AC 01 02  LDY $0201    Second char of U cmd
.FF33  C0 30     CPY #$30     Is it a 0 ?
.FF35  F0 03     BEQ $FF3A    If so, skip next stmt.
.FF37  4C 6C CB  JMP $CB6C    Not U0, jump to normal U cmds
.FF3A  AD 74 02  LDA $0274    No. characters in U0 cmd
.FF3D  C9 03     CMP #$03     Are there 3 chars?
.FF3F  F0 03     BEQ $FF44    If so, skip next stmt.
.FF41  4C 63 CB  JMP $CB63    Jump to normal U0 cmd
.FF44  A5 6A     LDA $6A      Get current REVCNT value
.FF46  AC 02 02  LDY $0202    Get third char of U0 cmd
.FF49  C0 2B     CPY #$2B     Is it a + ?
.FF4B  F0 08     BEQ $FF55    If so, branch to bump enable
.FF4D  C0 2D     CPY #$2D     Is it a - ?
.FF4F  D0 F0     BNE $FF41    If not, go to normal U0 cmd
.FF51  09 80     ORA #$80     DISABLE BUMP
.FF53  D0 02     BNE $FF57    Branch always
.FF55  29 7F     AND #$7F     ENABLE BUMP
.FF57  85 6A     STA $6A      Store new value in REVCNT
.FF59  60        RTS          Return from subroutine
```

YOU MUST ALSO CHANGE THE U COMMAND VECTOR:

LOCATION	(C64)	FROM	TO
$FE9A	($3E9A)	$5C	$30
FEA6	($3EA6)	CB	FF

DON'T FORGET TO DISABLE OR ADJUST THE ROM CHECKSUM AS EXPLAINED ABOVE.

You can use this same technique to add other commands to the DOS. Just use a different third character in the U0 command for each one. The new 1571 drive has several new commands which are accessed through the U0 command this way. You might also want to intercept other U commands such as U1 and U2. By altering other vectors in the same area, you can intercept commands such as M-W and M-R. See INSIDE COMMODORE DOS (p. 433) for help in locating the vectors for other commands.

The BUMP COMMANDS routine is located at $FF30 in memory, but the file on the program disk loads in at $3F30 to make burning the EPROM copy easier. If your copy of the E-F ROM starts at $2000 in the C64, then the BUMP COMMANDS file will load right into the correct position within it. Figure 16-4 shows the C64 locations corresponding to the U command vector in this case. Don't forget to disable the ROM checksum too. Now let's see how to replace the DOS ROM chips, starting with making a copy of the ROM to disk.

The ROM chips are located near the rear of the 1541's circuit board near the diamond-shaped heat sinks, just behind the microprocessor (a 40-pin chip which may or may not be labelled MOS 6502). Note that the ROMs are 24-pin chips. A pin-for-pin-compatible EPROM is available, called an MCM 68764. These chips are fairly expensive, even with chip prices coming down almost daily. However, you can use an inexpensive 28-pin 2764 EPROM instead of the 68764 if you also use a 28/24-pin AD adaptor (available from CSM). The cost for the AD and the 2764 is still about half the cost of the 68764 alone.

Of course, before you can replace your ROMs you have to be able to remove them first. The E-F ROM is socketed on almost every drive, but most drives do not have a socket on the C-D ROM. We don't recommend that you try to remove an unsocketed chip from the board unless you have a lot of experience in desoldering chips (and some spare cash for another drive just in case!).

The E-F ROM has had 5 different versions to date. There is only one version of the C-D ROM from the earliest 1541 drives with the white case to the latest "flip-lever" models. The version number of the ROM follows the dash after the Commodore part number on the chip. For the E-F ROM, version 5 would be labeled 901229-05. The C-D ROM is labeled 325302-01. If you are going to modify the DOS, you should start with a copy of version 5 of the E-F ROM.

There are two ways to read out the contents of the ROMs so that you can make changes to the code. The first is to use a drive monitor such as DRVMON64 from DI-SECTOR. This allows you to examine drive memory, and more importantly, transfer it to the C64. We recommend that you transfer the contents of each ROM separately and save them in separate files for convenience (depending on the changes you wish to make, you may need to alter only one of the ROMs). The following command can be used from DRVMON to transfer the C-D ROM to $2000-3FFF in the C64:

```
TD C000 DFFF 2000
```

The transfer should only take 25 seconds or so. DRVMON does lock-up occasionally, so if it seems to be taking much longer than this, you should RESET the computer and drive and start over. In any case, it is probably a good idea to do the transfer twice and compare the two copies. The E-F ROM can be transferred with a similar command:

```
TD E000 FFFF 2000
```

The second way to copy the ROMs is to use the PROMENADE to read them directly. Plug in the PROMENADE, load and run PROMOS from disk, and load a monitor such as MICROMON at $C000 (or use your PROMOS/HESMON cartridge). Remove the ROM from its socket and drop it into the PROMENADE. Since it is a 24-pin chip, make sure you put it all the way to the **right** end of the PROMENADE socket. The notch on the chip, as with all chips in the PROMENADE, should be to the **left**. The control word for reading the ROM is the same as for 68764 type EPROMs, namely 48. The PROMOS command to download the chip memory to $2000-3FFF in the C64 is:

```
£ 8192, 16383, 0, 48
```

(Addresses must be given to PROMOS in decimal: 8192 = $2000 and 16383 = $3FFF). Both chips can be read with the same command (be sure to save the first ROM to diskette before reading the second). The save command to use from MICROMON is:

```
S "name", 2000, 4000, 08   (the ,08 is optional)
```

From HESMON, use:

```
S "name" 08 2000 4000
```

Once you've made your modifications to the DOS code you're ready to burn the new version onto EPROM. With the code in the C64 at $2000-3FFF, the PROMOS command to burn it onto a 2764 EPROM is:

```
PI 8192, 16383, 0, 5, 7
```

(Don't use the letters PI; use the pi key from the keyboard, shift up-arrow). The command to burn a 68764 EPROM would be:

```
PI 8192, 16383, 0, 48, 6
```

If you are using a 2764, insert it into the AD adapter, making sure the notch on the chip and socket are matched up. Insert the 2764/AD or 68764 into the proper socket in the drive. The notch in the chip must be towards the **rear** of the drive. Power up the drive to test it. If the red light starts flashing, you probably forgot to disable or adjust the ROM checksum.

To test the changes given above, find a piece of commercial software that bumps the drive when it loads in. Another way to test the bump is to try to read an unformatted disk with a T/S editor. Start by turning the drive on. Load in the program or try to read the unformatted disk. The bump should have been disabled already on powerup, so you shouldn't hear any bump. Now send the drive the new U0+ command to enable the bump:

```
OPEN 15,8,15,"U0+" :CLOSE15
```

Try the error again - the drive should bump now (ouch!). Now disable the bump once again with the new U0- command:

```
OPEN 15,8,15,"U0-" :CLOSE15
```

Try the error one more time. NO BUMP!

As usual, this chapter was designed to get you thinking and experimenting on your own. Good luck!

FREEZE CARTRIDGE

The FREEZE cartridge described in this chapter is a handy utility for examining and debugging programs. It allows you to "freeze" the computer's memory at any point so you can examine it with your ML monitor. You should not confuse our simple FREEZE cartridge with other commercial products such as SNAPSHOT 64. SNAPSHOT 64, which is distributed exclusively in the U.S. by CSM, saves about as much of memory as it is possible to preserve, compacts it and eliminates the unused areas. SNAPSHOT 64 saves the compressed memory to disk and adds on an auto-boot automatically, so that in most cases the program can actually be restarted at exactly the same point it was interrupted. Our simple FREEZE cartridge merely collects some useful areas of memory and the processor into a more accesible area of memory for your examination.

Without a FREEZE cartridge or similar utility you can still usually examine most of memory after simply RESETing the computer. However, RESET re-initializes low memory ($0000-07FF), the I/O devices (VIC and CIA chips) and color RAM ($D800-DBFF). The RESET process also puts an annoying $55 byte at either $8000 or $A000 (see the AUTOSTART CARTRIDGES chapter for an explanation about this $55 byte). If you want to find out what these areas contained while a program was running, you must use a special utility such as our FREEZE cartridge. For convenience, another feature was built into the FREEZE cartridge: it automatically copies the contents of all "hidden" RAM down to more accessible areas. Hidden RAM refers to the RAM which lies under the BASIC and KERNAL ROMS and I/O devices. Finally, under the right conditions, the FREEZE cartridge can even preserve the contents of the 6510's registers, including the A, X, Y, stack pointer (SP), status register (SR) and the program counter (PC)!!

The FREEZE cartridge works by copying the areas of memory to be preserved into the $2000-7FFF area, and then returning control of the computer to you. After loading in a monitor you can examine and save the "frozen" memory. Figure 17-1 shows where in memory each preserved section is located.

The low memory area ($0000-0800) will be completely preserved except for a few bytes of the stack as explained below. The color RAM ($D800-DBFF) and hidden RAM ($A000-BFFF and $D000-FFFF) will be preserved exactly. FREEZE also saves the contents of two important RAM bytes, $0800 and $8000. Regardless of how it was entered, FREEZE terminates by going through the RESET process. This will place a $00 byte at $0800 (for BASIC programs) and a $55 at $8000 (as part of the RAM test routine). FREEZE preserves the contents of these two bytes before the RESET process in case you need to know what they were.

Figure 17-1: FREEZE CARTRIDGE TABLE

ORIGINAL AREA	SAVED AT	DESCRIPTION
$0000-07FF	$2000-27FF	Low memory including screen
$D800-DBFF	$2800-2BFF	Color RAM
$DC00-DCFF	$2C00-2CFF	CIA #1 (plus images)
$DD00-DD0F	$2D00-2DFF	CIA #2 " " " "
$D000-D02E	$2E00-2EFF	VIC II " " " "
**	$2F00-2F09	Processor registers, etc.
$A000-BFFF	$3000-4FFF	RAM under BASIC ROM
$D000-DFFF	$5000-5FFF	" " " I/O devices
$E000-FFFF	$6000-7FFF	" " " KERNAL ROM

** Breakdown of the $2F00 area :

00	PC	Program Counter High
01		" " " " " Low
02	A	A-register
03	X	X-register
04	Y	Y-register
05	SR	Status Register
06	SP1	Stack pointer (orig)
07	SP2	" " " (current)
08	0800	Original contents
09	8000	" " " " "

Bit	Proc Status Flag
0	Carry
1	Zero
2	IRQ Disable
3	Decimal Mode
4	Break
5	Not used
6	Overflow
7	Negative

With regard to the I/O devices, only the READ registers of the CIA and VIC chips will be preserved. These chips have some registers which can only be written to, or which return a different value when read than what was written there. Only the values returned when the chip is read will be saved. Also, since each of these chips has only a few registers and yet occupies a whole page of memory (256 bytes), "images" of the chip appear throughout the memory page. For the most part these images are identical to the original (the first copy), but certain registers such as timers may be different because of the time between reading one image and the next. This may tip you off that the program you're investigating enables a particular timer, for example. By the way, no attempt is made to preserve the SID chip because almost all of its registers are WRITE-ONLY. Only four registers are READable, and these are of little or no use without the others.

The 6510's registers may or may not be preserved, depending on how the FREEZE cartridge is activated (the different methods are discussed below). On RESET the contents will be lost, but in all other cases they will be saved. The A, X and Y registers are preserved intact. The program counter (PC) will indicate the next instruction to be EXECUTED, which is not necessarily the next instruction in sequential order. For example, if the last instruction executed was a JSR, the PC will contain the location of the SUBROUTINE, not the statement after the JSR instruction. By the way, the FREEZE cartridge records the PC in $2F00-01 in hi-byte / lo-byte order, rather than the usual reversed order. A PC address of $2000 would be recorded as 20 00 rather than 00 20.

A slight discrepancy in the PC value occurs with different methods for entering FREEZE, due to the workings of the 6502/6510 processor. The PC value placed on the stack by the processor is different with the JSR (Jump to SubRoutine) and BRK (BReaK) instructions than with an NMI interrupt (Non-Maskable Interrupt). The PC value pushed on the stack by JSR is the address of the **last byte of the JSR instruction** rather than the first byte of the next instruction, so it is one byte too low (the RTS statement compensates for this by incrementing the PC value pulled off the stack before using it). Thus the PC will be **one byte too low** when **JSR** is used to enter FREEZE.

The BRK instruction, on the other hand, starts with a PC value equal to the location of the BRK instruction itself, but then increments the PC **twice** before pushing it on the stack. Instead of pointing to the next byte immediately after BRK, the PC skips a byte and points to the second byte after BRK. This means the PC will be **one byte too high** when **BRK** is used to enter FREEZE. Finally, an NMI increments the PC only once before pushing it on the stack, so the PC points right to the next instruction to be executed. Thus the PC will be **exactly correct** when an **NMI** is used to enter FREEZE. An NMI (via the RESTORE key) is the most common method of entering FREEZE, so most of the time the PC will be exactly correct.

As far as the other 6510 registers are concerned, the status register (SR) will be saved intact except when FREEZE is entered through RESET and BRK. RESET will wipe out the SR entirely, while BRK will just change the BRK flag to a "1". The stack pointer (SP) is altered when you enter FREEZE, but the original value can be recovered (except with RESET). Before the FREEZE cartridge is entered, some values are pushed on the stack, which decrements the SP and wipes out a few bytes of stack memory. By pulling the correct number of items back off the stack, the SP is restored to its original value. The stack memory bytes cannot be recovered, however. FREEZE preserves both the original value of SP before anything was pushed on the stack, and its actual value when FREEZE was entered, after the values were pushed on the stack. The area of the stack from the original SP down to but not including the second SP value will have been wiped out. When using the NMI entry (RESTORE key), two more bytes starting at the second SP value will be wiped out too.

Putting the FREEZE cartridge on EPROM is easy. Turn off the computer and plug in the PROMENADE. Load in PROMOS and run it. Load in the file called "FREEZE" from the program disk. The FREEZE program normally resides at $8000-FF, but the file loads in at $2000-FF for convenience. Insert a 2764 EPROM (8K) into the PROMENADE (notch to the left) and lock down the handle. Now type the following PROMOS command:

```
PI 8192, 8447, 0, 5, 7
```

Since the program takes less than 256 bytes, you can use a smaller EPROM instead of the 2764, such as a 2732 (4K) or 2716 (2K). Consult the PROMENADE manual for the appropriate CW (control word) and PW (program word).

Now you must set up your cartridge board as a standard 8K cartridge. This means that the cartridge should ground only the EXROM line. Make sure the cartridge board jumpers are set so that pin 9 of the cartridge port (EXROM) is connected to the ground line, but pin 8 (GAME) isn't. Insert the EPROM into the cartridge board and you're all set. Depending on your cartridge board, you may have to change some other jumpers if you are using a 2732 or 2716 rather than a 2764.

The FREEZE cartridge can be activated a number of ways. One method may be better suited than the others for the program you are investigating, depending on how and where your program executes. The variety of methods may seem confusing, but in practice you'll usually only use one or two most of the time. All of the methods preserve pretty much the same RAM memory. The main difference between the various methods is whether the processor contents are preserved and how much of the stack will be disturbed. You may already know enough about your program to decide which method to use, or you may have to experiment to find the best method. Generally, you'll need a cartridge expansion board in order to switch in the FREEZE cartridge at the proper time. A RESET button may also be required. See the individual methods below for details.

RESET The FREEZE cartridge resides at $8000. The most important question, as far as activating FREEZE is concerned, is whether or not your program will be executing code in the $8000-9FFF area when you want to interrupt it. If it will be executing in this area, you must use the RESET method. **HOLD DOWN** the RESET button, switch the cartridge on, and then release the button. This prevents the processor from executing any instructions during the time you are moving the switch(es). With RESET you'll be able to preserve all the RAM memory, but not the processor registers or I/O devices. The top two bytes of the stack ($01FE-FF) will be altered too. If you don't care about the processor's or I/O devices' contents, RESET is the simplest method to use. If your program switches out the BASIC or KERNAL ROMs or the I/O devices, the FREEZE cartridge will also be switched out. You'll have to use the RESET method in these cases too.

RESTORE This is the main method for entering the FREEZE cartridge. If your program does not have either of the limitations above, requiring the use of RESET, you can probably use the RESTORE key. Wait until the program reaches the point at which you want to interrupt it, switch the cartridge on and press RESTORE. You can recover all of the processor's registers and most of the CIA and VIC chips' registers this way. Eight bytes of the stack will be wiped in the process, however. These will be the next eight unused bytes of the stack, so all of the stack actually used by the program will usually be preserved. The RESTORE key works by generating an NMI (Non-Maskable Interrupt) which saves the processor's contents on the stack and then jumps to the FREEZE cartridge using the CBM80 cartridge warm-start vector at $8002.

The RESTORE key has one quirk - it tends to "bounce". As you've probably noticed by now, you usually have to HIT the RESTORE key rather than just PRESS it to get it to respond. If you do this with the FREEZE cartridge, it will often respond as if you pressed the RESTORE key twice in a row. The first RESTORE will start the FREEZE process, but the second will interrupt the process and start it again.

There are two ways around this problem. Try pressing the RESTORE key slowly. Nothing will happen (try it!). Now release the key as quickly as possible - it should cause exactly one RESTORE to be registered. The other alternative is to generate an NMI through hardware. Pin D on the cartridge port is the NMI line. Looking at the edge of the **cartridge** with it right-side up (computer's-eye view), NMI is the fourth pin from the LEFT on the BOTTOM row. If you have a multi-slot exapnsion board, you can use the corresponding pin on one of the unused slots. By shorting this pin to ground (pin 1, 22, A or Z) you'll generate the equivalent of pressing the RESTORE key. A switch should be installed for this purpose if you'll be doing it often. You can even mount the switch right on the FREEZE cartridge and tie into the pins on the cartridge board.

The remaining ways to activate FREEZE require modifying the program being investigated. They are usually needed only if the program alters normal system vectors or must be interrupted at a specific instruction.

NMI Vector The RESTORE key or hardware NMI causes a jump to the NMI routine in the KERNAL. The NMI routine uses a vector in RAM at $0318-19 to decide where to go next. If you change this vector before starting your program, an NMI will jump directly into the FREEZE cartridge. However, many programs alter the NMI RAM vector when they execute, so that the check for a CBM80 autostart key will not be done. This will disable the FREEZE cartridge, since FREEZE uses the CBM80 key to start itself on RESTORE. Fortunately, many programs that alter the NMI RAM vector do so in an obvious manner near the beginning of the boot process. You may be able to remove their NMI vector value and replace it with the normal value ($FE47). Better yet, you can point the NMI directly into FREEZE by inserting the value $800B instead. This will preserve a few extra bytes of stack memory, and seems to prevent the RESTORE key "bounce" discussed above. Otherwise it is identical to the RESTORE method.

JSR $8009 If you can't find where the NMI vector is changed, or if the modified NMI vector is necessary for the operation of the program, or if you need to interrupt the program at a particular instruction, you may be able to use a JSR to enter the FREEZE cartridge directly. Simply insert a JSR $8009 statement at the point where you wish to interrupt the program. As with most methods, you must either be able to have the cartridge switched on throughout the whole program, or at least be able to judge from watching the program when it is ok to switch the cartridge on. This method wipes out only three bytes of stack memory. Remember, the PC value recorded by FREEZE will be 1 byte too low, as discussed above.

BRK Vector The final method involves the BRK vector. This would only be used if you just can't fit a JSR instruction in the code at the point required. A BRK instruction ($00) requires only one byte as opposed to three for a JSR. In addition to inserting the BRK instruction in the code, you'll have to alter the normal BRK RAM vector at $0314. Do this before starting your program. As with the NMI vector, your program may change this vector itself. If it does, you may be able to substitute your own value for theirs. In any case, use the value $8018 for the BRK vector. This enters FREEZE at the same point as the NMI methods, and so has the same characteristics, with one exception. As discussed above, the PC value recorded by FREEZE will be one byte too high.

Figure 17-2 shows a listing of the FREEZE cartridge program. The entry points for each of the various methods are labelled so that you may see the differences in the various methods. The main difference lies in the number of bytes that were pushed onto the stack by the NMI, BRK, JSR etc. that was used to activate FREEZE. A loop at $801F-25 is used to pull the proper number of bytes off the stack and store them at $2F00. This process restores the stack pointer back to its original value as a side effect. The remaining bytes of the $2F00 area are filled with $FF's for convenience in interpreting the results of the FREEZE process.

In addition to the main routine, FREEZE uses two special subroutines. The subroutine at $80C0 copies one area of memory to another. It always copies a whole number of memory pages (256-byte segments). The number of pages to copy must be placed in the accumulator before calling the routine. In a similar fashion, the X-register is used to specify the area to copy from (the source area). Only the high byte of the beginning of the source area is specified; the lo byte is assumed to be $00. The Y-register is used to specify the hi byte of the beginning of the area to copy to (the destination area).

The copy subroutine stores the beginning addresses of the source and destination areas in zero page at $FB-FC and $FD-FE respectively. A simple loop is used to load a byte from the source area and save it in the destination area. The pages are counted as they are copied and the routine terminates after the proper number of pages have been done. This routine is used do almost all of the copying in the FREEZE process, starting with the low memory area at $0000-07FF. This area of memory presents a problem, since the copy routine stores its source and destination addresses in this area at $FB-FE. When we use the routine to copy the low memory area, it will wipe out the values which were in $FB-FE previously. Therefore, these values are saved temporarily in $2F90-2F93 before the copy routine is called. This is done by the code at $8038-41 in the main routine. Then the copy routine is called to copy the low memory area up to $2000-27FF. Finally, the $FB-FE bytes saved earlier are moved into their corresponding positions in the low memory copy, at $200FB-FE.

Next, the ROMs and I/O devices are switched into memory if they weren't in already. Then the color RAM ($D800-DBFF) and VIC chip ($D000-FF) are copied down to $2800-2BFF and $2E00-FF respectively. In order to copy the hidden RAM areas, we must switch the BASIC and KERNAL ROMs and I/O devices out of memory. This presents another problem since switching out BASIC or the KERNAL will also switch out our FREEZE cartridge! To get around this, we download a second special routine to RAM. This routine is located at $8090-BE in the FREEZE cartridge but is downloaded to $2090-BE in RAM. The RAM routine immediately switches off the ROMs and I/O. Next it saves the contents of the two miscellaneous RAM bytes at $0800 and $8000 (the second of which was under the FREEZE cartridge before).

The RAM routine then copies the hidden RAM at $A000-BFFF (under BASIC) down to $3000-4FFF and the hidden RAM at $D000-FFFF (under the I/O devices and the KERNAL) down to $5000-7FFF. To do this it uses the copy subroutine we discussed above. This is possible because the copy routine was downloaded to RAM too, at $2FC0-DA. The copy routine does not refer to any locations in memory directly (except $FB-FE), so it is completely **relocatable.** That is, it can be moved to an area of memory and executed without having to be altered to adjust for its new location.

Finally, after all copying is done, the ROMs are switched back in. The stack pointer and decimal mode flag are initialized to their standard values, and a value is set up in X for the VIC chip. Then FREEZE terminates by jumping into the RESET process past the CBM80 check (otherwise the FREEZE cartridge would be started all over again!). This returns control of the computer to you, so you can examine the preserved memory.

You should find the FREEZE cartridge very useful, especially for downloading hidden RAM conveniently. With its ability to preserve the processor's registers and low memory, it should be a handy program debugging tool too.

Figure 17-2: FREEZE CARTRIDGE LISTING

```
8000 2E 80                  Cold start $802E - RESET
8002 18 20                  Warm start $8018 - RESTORE

8004 C3 C2 CD 38 30         CBM80 - Autostart key

8009 08        PHP          JSR Entry - Push proc. status
800A 78        SEI          Disable IRQ interrupts
800B 8D 03 2F  STA $2F03    NMI RAM Vector Entry - Save Acc.
800E 8E 04 2F  STX $2F04    Save X
8011 8C 05 2F  STY $2F05    Save Y
8014 A0 02     LDY #$02     3 bytes on stack - SR, PC lo, PC hi
8016 D0 03     BNE $801B    Always branch
8018 78        SEI          RESTORE and BRK Entry - Disable IRQ's
8019 A0 05     LDY #$05     6 bytes on stack - Y,X,A,SR, PC lo/hi
801B BA        TSX
801C 8E 07 2F  STX $2F07    Save current stack pointer
801F 68        PLA          Pull bytes from stack ...
8020 99 00 2F  STA $2F00,Y  ... and save
8023 88        DEY
8024 10 F9     BPL $801F
8026 BA        TSX
8027 8E 06 2F  STX $2F06    Save original stack pointer
802A A0 0A     LDY #$0A     Start clearing at $2F0A
802C D0 02     BNE $8030    Always branch
802E A0 00     LDY #$00     RESET Entry - Clear all $2F00 area
8030 A9 FF     LDA #$FF     Filler value for clear
8032 99 00 2F  STA $2F00,Y  Clear unused areas
8035 C8        INY
8036 D0 FA     BNE $8032
8038 A2 03     LDX #$03     Save 4 bytes
803A B5 FB     LDA $FB,X      from $FB-FE
803C 9D 90 2F  STA $2F90,X    to $2F90-93
803F CA        DEX
8040 D0 F8     BPL $803A
8042 A9 08     LDA #$08     Copy 8 pages (low mem)
8044 A2 00     LDX #$00       from $0000-07FF
8046 A0 20     LDY #$20         to $2000-27FF
8048 20 C0 80  JSR $80C0    Copy routine
804B A2 03     LDX #$03
804D BD 90 2F  LDA $2F90,X  Recover original $FB-FE bytes
8050 9D FB 20  STA $20FB,X    and store at $20FB-FE
8053 CA        DEX
8054 D0 F7     BPL $804D
8056 A9 37     LDA #$37     Make sure BASIC
8058 85 01     STA $01        and KERNAL ROMs
805A A9 2F     LDA #$2F       and I/O devices
805C 85 00     STA $00        are switched in
805E A9 06     LDA #$06     Copy 6 pages (color RAM)
8060 A2 D8     LDX #$D8       from $D800-DBFF
8062 A0 28     LDY #$28         to $2800-2BFF
8064 20 C0 80  JSR $80C0    COPY
8067 A9 01     LDA #$01     Copy 1 page (VIC)
8069 A2 D0     LDX #$D0       from $D000-FF
806B A0 2E     LDY #$2E         to $2E00-FF
```

```
806D 20 C0 80 JSR $80C0   COPY
8070 A0 4F    LDY #$4F    Hidden RAM routine
8072 B9 90 80 LDA $8090,Y   Download from $8090-DF
8075 99 90 2F STA $2F90,Y             to $2F90-DF
8078 88       DEY
8079 10 F7    BPL $8072
807B 4C 90 2F JMP $2F90   Execute hidden RAM routine


              Downloads the hidden RAM under BASIC,
              I/O devices and KERNAL to $3000-7FFF

8090 A9 34    LDA #$34    Turn off ROMs and I/O
8092 85 01    STA $01       devices (all RAM)
8094 AD 00 08 LDA $0800   Byte preceding BASIC area
8097 8D 08 2F STA $2F08     - save
809A AD 00 80 LDA $8000   Byte under cartridge area
809D 8D 09 2F STA $2F09     - save
80A0 A9 20    LDA #$20    Copy 32 pages (under BASIC)
80A2 A2 A0    LDX #$A0      from $A000-BFFF
80A4 A0 30    LDY #$30        to $3000-4FFF
80A6 20 C0 2F JSR $2FC0   COPY
80A9 A9 30    LDA #$30    Copy 48 pages (under I/O, KERNAL)
80AB A2 D0    LDX #$D0      from $D000-FFFF
80AD A0 50    LDY #$50        to $5000-7FFF
80AF 20 C0 2F JSR $2FC0   COPY
80B2 A9 37    LDA #$37    Turn ROMS and I/O
80B4 85 01    STA $01       devices back on
80B6 A2 FF    LDX #$FF    Set stack pointer
80B8 9A       TXS           to top of stack
80B9 D8       CLD         Clear decimal mode
80BA A2 00    LDX #$00    Prepare value for VIC
80BC 4C EF FC JMP $FCEF   Jump into RESET past CBM80 check


                Copy routine - copies whole no. of
                pages from one area to another.

80C0 86 FC    STX $FC     Hi byte - source area
80C2 84 FE    STY $FE      "   "  - destination area
80C4 AA       TAX         No. pages to copy
80C5 A9 00    LDA #$00
80C7 85 FB    STA $FB     Lo byte - source area
80C9 85 FD    STA $FD      "   "  - destination area
80CB A8       TAY         Zero index
80CC B1 FB    LDA ($FB),Y Load byte
80CE 91 FD    STA ($FD),Y Save
80D0 C8       INY
80D1 D0 F9    BNE $80CC
80D3 E6 FC    INC $FC     Next source page
80D5 E6 FE    INC $FE     Next destination page
80D7 CA       DEX
80D8 D0 F2    BNE $80CC
80DA 60       RTS


                End of FREEZE cartridge listing
```

ORACLE/PAPERCLIP CARTRIDGE

If you use DELPHI'S ORACLE (tm) database or PAPERCLIP (tm) word processor from Batteries Included, you know that these two programs are among the best of their type. Both programs are loaded with features, but you pay a high price for all that sophistication. We're not talking about the purchase price (they're worth every cent) but rather the length of time it takes to load the program in from disk. Unlike most commercially protected programs, the long load time isn't because of any fancy protection on the disk. In fact, the disk is totally unprotected. The long load is simply due to the length of the program - there's a lot of code there.

The protection scheme used on these products employs a hardware key or "dongle" which plugs into joystick port #1. Without this key, the programs will not run. This is a good compromise between convenience for the user and protection for the manufacturer. The user may make as many backup copies of the original program as desired - but with only one key the programs can only be executed on one computer at a time. This feature lets the user put the program anywhere they wish, such as on their data disk or a hard disk - or a CARTRIDGE!

That's right, you can put BOTH Oracle (tm) and Paperclip (tm) on the same cartridge (let's call them OR and PC from now on - note that the Oracle has been renamed The Consultant). With a push of the RESET button (or on powerup) you can have a menu appear and offer you a choice of programs. Simply choose the one you want, and seconds later the program is up and running (assuming the key is plugged in). Not only is this the ultimate in convenience, but it can be practical too. Time is money, as they say, especially if you use these programs for business (like we do). If you go back and forth from PC to BASIC, a cartridge can save you a LOT of time.

For the OR/PC cartridge, you'll need a PC4 bank-switch cartridge board. You'll also need four 27128 (16K) EPROMS. We also highly recommend that you get a plastic housing for the cartridge. These materials are all available from CSM. The program disk accompanying this book contains the menu program for this cartridge. It does NOT contain the OR or PC program files - these must be purchased from Batteries Included.

To start off, load in PROMOS and run it. PROMOS will put itself up at the top of BASIC memory in the $9000's. Don't use the HESMON/PROMOS cartridge because it starts in memory at $8000, and we'll need that area. Now load in MICROMON at $C000 and execute it. The first thing you should do is clear out the memory we'll use:

```
F 1000 8FFF FF
```

Now insert the program disk and load the menu program:

```
L "OR/PC MENU"
```

This program will load in at $1000. We have to start this far down in memory since the OR and PC files are so long. Now insert your OR program disk in the drive. Load the main OR program into memory with:

```
L 1400 "ORACLE*"
```

This loads in the file starting at $1400, past the end of the menu program. Depending on which version of the OR program you have, the ending address reported by MICROMON will be different. We'll just burn the whole $1000-8FFF area to EPROM so we won't have to worry about what the ending address is (as long as the OR program ends before $9000!). It will take two 27128 EPROMs to hold this much memory, and so we'll have to burn them in two separate steps.

Before inserting the first EPROM, you should initialize the PROMENADE. Type Z and hit RETURN. Now insert the first 27128 into the PROMENADE (notch to the left as always). Flip the lever down to lock it in place. Burn the first half of the memory area (16K) to the EPROM with the following command:

```
PI 4096, 20479, 0, 5, 6
```

Don't use the letters PI; use the pi key on the keyboard (shifted up-arrow, next to RESTORE). PROMOS requires its addresses in decimal: 4096 = $1000 and 20479 = $4FFF.

Burning this much memory to EPROM can take quite a few minutes, so be patient. Using a PMW (the last number in the command) of 10 or 14 instead of 6 will speed it up but may affect the reliablility. If your EPROMs have a **LARGE MATRIX ONLY**, you can try another shortcut that might save a lot of time. The matrix is the heart of the chip, visible through the window on top. A large matrix will be approx. 3/8" x 1/4"; a small matrix will be half that size or smaller. The command for a large matrix chip is:

```
PI 4096, 20479, 0, 4, 7
```

Don't use this command on a small matrix chip! It will probably damage the EPROM! If you aren't sure which chip you have, use the first version of the command. It will work with either type. The shortcut command works by using a higher programming voltage than usual (25 volts vs. 21 volts). The larger matrix chips can take the extra voltage, but the smaller one can't. This method works quite well for us but we can't guarantee that it will work for you.

By the way, if you have EPROMs labelled 27128**A**, they should be programmed at only 12.5 volts. They may or may not say this on them. You will damage them if you don't use the right CW (the next to last number in the command). The proper CW for these chips is 6, so they entire command should be:

```
PI 4096, 20479, 0, 6, 7
```

When the chip is finished, take it out of the PROMENADE and label it chip #0 (we start numbering them at 0 because that's the way the cartridge board is labeled). Now insert the next EPROM and burn the second 16K of memory with:

```
    PI 20480, 36863, 0, 5, 6    (Any size matrix)
or  PI 20480, 36863, 0, 4, 7    (LARGE MATRIX ONLY!)
```

(20480 = $5000 and 36863 = $8FFF)

Remove this chip from the PROMENADE and label it chip #1.

Now it's time to do the PC program. Start by clearing the memory area again:

```
F 1000 8FFF FF
```

Insert your PC disk and load in the PC program:

```
L 1000 "PAPERCLIP*"
```

Once again the ending address will vary from one version to another, so we'll just burn the whole $1000-8FFF area. The commands to burn the other two EPROMS are exactly the same as before. Always start by initializing the PROMENADE with a Z command. For EPROM #2, use:

```
PI 4096, 20479, 0, 5, 6    (Any size matrix)
PI 4096, 20479, 0, 4, 7    (LARGE MATRIX ONLY!)
```

For EPROM #3, use:

```
    PI 20480, 36863, 0, 5, 6    (Any size matrix)
or  PI 20480, 36863, 0, 4, 7    (LARGE MATRIX ONLY!)
```

Be sure to label the EPROMs after you remove them from the PROMENADE to avoid mixing them up.

Now you're ready to plug the EPROMs into the bank-switch board. The four sockets on the board are labeled from 0 to 3, and all you have to do is make sure each EPROM is inserted into the corresponding socket (EPROM #0 in socket #0 and so on). Socket #0 is the closest one to the computer when the cartridge is plugged in, just in case you're not sure. The others are in order from #0. The notch on each EPROM should be to the LEFT when the cartridge is inserted in the computer. There is usually a notch or dot on the socket to guide you.

Now put the board into the bottom half of the plastic housing (the half with the plastic pins). Make the sure the board is flipped so the EPROMs are on the UPPER side (still visible). Now put the top half of the case on. The cartridge is ready for use!

Let's take it out for a spin. Turn off the computer (or cartridge slot if you have an expansion board). Insert the cartridge and turn the computer power or cartridge slot back on. You should be presented with a menu showing three options:

0 - Exit to BASIC
1 - Delphi's Oracle
2 - Paperclip

At this point the menu program has been downloaded into the normal BASIC area as a regular BASIC program. If you choose option 0, the computer will be RESET and the normal powerup screen will appear. To reactivate the cartridge, you MUST use a hard RESET (powerup or RESET button). A software RESET (SYS 64738) will NOT work.

If you choose either of the other options, make sure you have the proper key for that program plugged into the joystick port before you hit RETURN! It will take only a moment for the program to be downloaded from the cartridge. PC takes about 13 seconds before it comes up because it is heavily encrypted. After the program is downloaded, the cartridge will automatically switch itself off so that all of the normal RAM is available for the program. It can do this because of the special bank-switching circuitry built into it. If you have a cartridge power switch or expansion board, you can switch the cartridge power off after the program is up and running. This will give your power supply box a bit of a break.

This menu program can also be used to put other large programs onto a bank-switch board. Unless you want to rewrite the download/run routine (not an easy task), you can only have two programs and they must occupy the same areas in the cartridge as OR and PC. They will be downloaded to the BASIC area and RUN like BASIC programs. Just follow the directions above for this cartridge, and load your programs at the appropriate times instead. Changing the titles on the menu is relatively simple if you examine the menu program with a monitor.

(INTENTIONALLY LEFT BLANK)

A P P E N D I C E S

APPENDIX A - CONTENTS OF PROGRAM DISKETTE

PROGRAM NAME	CHAPTER	DESCRIPTION
MICROMON 4096	5	ML Monitor at 4096 ($1000)
MICROMON 8192	5	" " " 8192 ($2000)
MICROMON 32768	5	" " " 32768 ($8000)
MICROMON 36864	5	" " " 36864 ($9000)
MICROMON 49152	5	" " " 49152 ($C000)
128 CARTRIDGE	10	Sample C128 cartridge
8K BASIC	11	Download/run for BASIC programs (8K)
MENU MAKER	12	8K/16K cartridge making utility
8K/16K MENU	12	Menu routine for 8K/16K cartridges
DOWNLOAD/RUN	13	For bank-switch cartridges
KERNAL/WEDGE	15	KERNAL with DOS wedge installed
KERNAL-TAPE	15	KERNAL minus tape routines
CHECKSUM DOS	16	Routine to checksum a DOS ROM
BUMP COMMANDS	16	Adds commands to enable/disable bump
FREEZE	17	Cartridge to preserve memory
OR/PC MENU	18	Menu for Oracle/Paperclip cartridge

Other programs may also be included.

APPENDIX B - CONTROL AND PROGRAM METHOD WORDS

CHIP TYPE	CONTROL WORD (CW)	PROGRAM METHOD WORD (PMW) #1	#2	#3	STD	PGM VOLTS	NO. PINS
EPROMs							
2758	40	7	10	14	3	25	24
2516	40	7	10	14	3	25	24
2716	40	7	10	14	3	25	24
2532	24	7	10	14	3	25	24
2732	224	7	10	14	3	25	24
2732A	225	7	10	14	3	21	24
2564	20	7	10	14	3	25	28
2764	5	7	10	14	3	21	28
27C64	5	7	10	14	3	21	28
2764A	6	6	10	14	*	12.5	28
27128	5	7	10	14	3	21	28
27128A	6	7	10	14	*	12.5	28
27256	230 **	6	10	*	*	12.5	28
	229 **	6	10	*	*	21	28
27C256	229	6	10	14	*	21	28
68764	48	6	11	15	*	25	24
68766	48	6	11	15	*	25	24
68769	48	6	11	15	*	25	24
EEPROMs							
2815	57	6	11	15	3	21	24
2816	57	7	*	*	3	21	24
X2804A	57	1	*	*	*	5	24
X2816A	57	1	*	*	*	5	24
X2864A	7	1	*	*	*	5	28
48016	40	5	9	13	2	25	24
5133	5	7	*	15	3	21	28
5143	5	7	*	15	3	21	28
5213	57	4	9	13	2	21	24
52B13	57	1	*	*	*	21	24
52B33	7	4	*	*	*	5	28
58064	7	7	*	*	*	5	28

* THIS METHOD IS NOT RECOMMENDED

** CHECK THE PROGRAMMING VOLTAGE!
(MAY BE WRITTEN ON CHIP)

APPENDIX C - TABLES OF HEX/DECIMAL EQUIVALENTS

All addresses in PROMOS commands must be specified in DECIMAL. The following table lists the decimal equivalent for various HEX addresses, starting at $0000 and going by steps of 2K.

HEX	DECIMAL	K	HEX	DECIMAL	K
$0000	0	0	$8000	32768	32
0800	2048	2	8800	34816	34
1000	4096	4	9000	36864	36
1800	6144	6	9800	38912	38
2000	8192	8	A000	40960	40
2800	10240	10	A800	43008	42
3000	12288	12	B000	45056	44
3800	14336	14	B800	47104	46
4000	16384	16	C000	49152	48
4800	18432	18	C800	51200	50
5000	20480	20	D000	53248	52
5800	22528	22	D800	55296	54
6000	24576	24	E000	57344	56
6800	26624	26	E800	59392	58
7000	28672	28	F000	61440	60
7800	30720	30	F800	63488	62

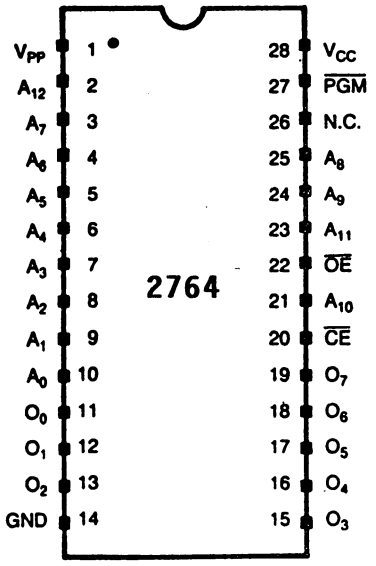
For most PROMOS applications, you can store the EPROM copy in the C64 starting at $2000 (8192). The following table lists the starting and ending addresses to use with PROMOS for the most common chip sizes, when the C64 copy starts at $2000.

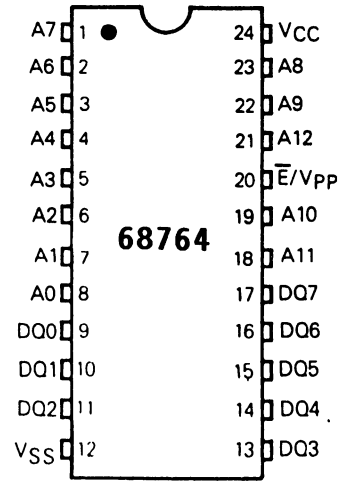
EPROM SIZE	NUMBER	HEXADECIMAL BEGIN, END	DECIMAL BEGIN, END
1K	xx08	$2000, $23FF	8192, 9215
2K	xx16	2000, 27FF	8192, 10239
4K	xx32	2000, 2FFF	8192, 12287
8K	xx64	2000, 3FFF	8192, 16383
16K	xx128	2000, 5FFF	8192, 24575
32K	xx256	2000, 9FFF	8192, 40959

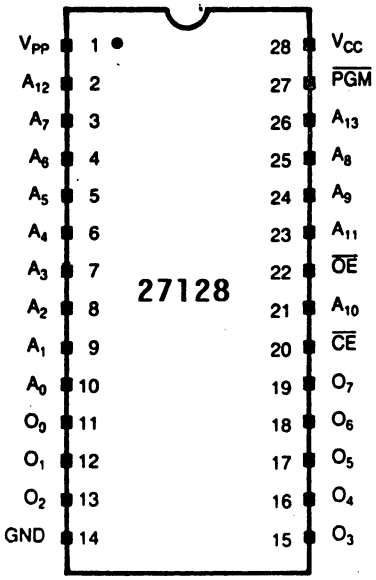
NOTE: **xx** = Chip family, e.g. **27** for 2708, 2716, etc.

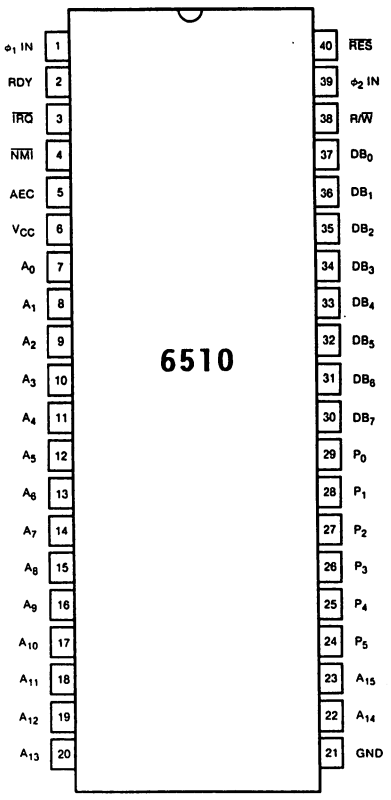
APPENDIX D - PINOUTS AND SCHEMATICS

For more detailed information on various EPROMs, see the data sheets at the end of the book.









CARTRIDGE EXPANSION PORT

The expansion connector is a 44-pin (22/22) female edge connector on the back of the Commodore 64. With the Commodore 64 facing you, the expansion connector is on the far right of the back of the computer. To use the connector, a 44-pin (22/22) male edge connector is required.

This port is used for expansions of the Commodore 64 system which require access to the address bus or the data bus of the computer. Caution is necessary when using the expansion bus, because it's possible to damage the Commodore 64 by a malfunction of your equipment.

The expansion bus is arranged as follows:

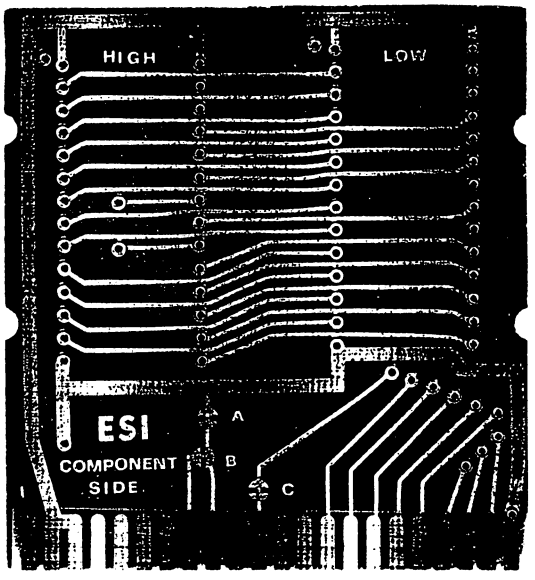
The signals available on the connector are as follows:

NAME	PIN	DESCRIPTION
GND	1	System ground
+5 VDC	2	(Total USER PORT and CARTRIDGE devices can
+5 VDC	3	draw no more than 450 mA.)
$\overline{\text{IRQ}}$	4	Interrupt Request line to 6502 (active low)
R/$\overline{\text{W}}$	5	Read/Write
DOT CLOCK	6	8.18 MHz video dot clock
$\overline{\text{I/O1}}$	7	I/O block 1 @ $DE00–$DEFF (active low) unbuffered I/O
$\overline{\text{GAME}}$	8	active low ls ttl input
$\overline{\text{EXROM}}$	9	active low ls ttl input
$\overline{\text{I/O2}}$	10	I/O block 2 @ $DF00–$DFFF (active low) buff'ed ls ttl output
$\overline{\text{ROML}}$	11	8K decoded RAM/ROM block @ $8000 (active low) buffered ls ttl output
BA	12	Bus available signal from the VIC-II chip unbuffered 1 ls load max.
$\overline{\text{DMA}}$	13	Direct memory access request line (active low input) ls ttl input
D7	14	Data bus bit 7
D6	15	Data bus bit 6
D5	16	Data bus bit 5
D4	17	Data bus bit 4 — unbuffered, 1 ls ttl load max
D3	18	Data bus bit 3
D2	19	Data bus bit 2
D1	20	Data bus bit 1
D0	21	Data bus bit 0
GND	22	System ground
GND	A	
$\overline{\text{ROMH}}$	B	8K decoded RAM/ROM block @ $E000 buffered
$\overline{\text{RESET}}$	C	6502 RESET pin (active low) buff'ed ttl out/unbuff'ed in
$\overline{\text{NMI}}$	D	6502 Non Maskable Interrupt (active low) buff'ed ttl out, unbuff'ed in
ϕ2	E	Phase 2 system clock
A15	F	Address bus bit 15
A14	H	Address bus bit 14
A13	J	Address bus bit 13
A12	K	Address bus bit 12
A11	L	Address bus bit 11
A10	M	Address bus bit 10
A9	N	Address bus bit 9
A8	P	Address bus bit 8 — unbuffered, 1 ls ttl load max
A7	R	Address bus bit 7
A6	S	Address bus bit 6
A5	T	Address bus bit 5
A4	U	Address bus bit 4
A3	V	Address bus bit 3
A2	W	Address bus bit 2
A1	X	Address bus bit 1
A0	Y	Address bus bit 0
GND	Z	System ground

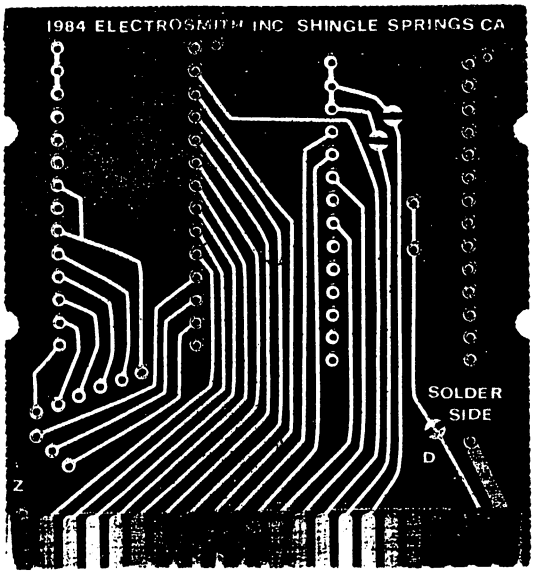
Overbar means active low

PC2 CARTRIDGE BOARD

EPROM SIDE



BACK SIDE



APPENDIX E - EPROM AND CARTRIDGE TIPS

At first when you work with EPROMs and cartridges, they may seem very mysterious. The more you work with them, the more you'll find they're really very simple to use. Chances are, you won't have any trouble as long as you observe the simple guidelines below.

One aspect of programming EPROMs that shouldn't give you any trouble is the PROMENADE itself. The PROMENADE is a very well-made and reliable instrument. We've sold many, many PROMENADEs and haven't seen a bad one yet.

EPROM TIPS

- Don't buy used or surplus chips. You just don't know what you're getting. The chips may be perfectly good, or they may be flaky. Tracking down a problem caused by a flaky chip can be fustrating and time-consuming for a beginner.

- Be sure to insert the chip into the PROMENADE correctly. The notch on the chip always goes to the LEFT when you insert it in the PROMENADE. A 24-pin chip must be placed all the way to the RIGHT end of the PROMENADE's ZIF socket.

- Make sure a chip is completely erased before you try to program it. A fully erased chip will contain all $FF bytes. Chips are supposed to arrive from the factory fully erased.

- Be sure to use the correct CW (Control Word) for the type of chip you're using (see appendix B). Using an incorrect CW can damage a chip permanently.

- Consult appendix B for a reliable PMW to use for your type of chip. If you are having trouble with one of the intelligent methods, say #3, try using a more reliable method, say #2. If all else fails, try using the standard method.

- Make sure you're using the correct starting and ending addresses in the PROMOS command. Appendix C contains a list of the most commonly used starting and ending addresses for different size chips, as well as a useful hex/decimal conversion table.

- Watch out for static electricity. Avoid touching the metal pins of a chip with your fingers as much as possible. You can blow a chip without even knowing it.

- Be very careful when you are inserting a chip into a socket. Make sure all the pins are started in the holes before you push down on the chip.

- When you remove an EPROM from a socket, do it slowly and carefully. Use a specially made IC puller or else use a small screwdriver to pry up one end of the chip and then the other until it is free. Avoid bending the pins back and forth, since they may break off.

- Keep the pins of your EPROMs clean. If necessary to clean them, use a high percentage (91%) alcohol and a Q-tip.

- Make sure the quartz window on top the EPROM is clean before you try to erase it. Alcohol can be used to remove grease or the gummy residue left over from labels.

CARTRIDGE TIPS

- Install sockets on your cartridge board if it doesn't have them already. The cost is minimal compared to the time and trouble of desoldering a chip if you ever want to make a change.

- Be sure to insert each chip into its socket correctly. There is usually a notch or dot by pin #1 to match up with the notch on the chip.

- Make sure the right chip is in the right socket on the board.

- Be sure to set up the GAME and EXROM lines correctly for your type of cartridge. If you have a CSM or CARDCO expansion board, you can use the LEDs (lights) to verify that GAME and EXROM are set correctly.

- If you are using a different size chip than a 2764, you may have to cut or connect some lines on the board. See the instructions that came with your board.

- If you are having intermittent trouble with a cartridge, try cleaning the cartridge's edge connector and the computer's cartridge slot with alcohol. If you have an expansion board, clean its connectors too.

- Never insert or remove a cartridge when the power is on. You could permanently damage your computer and cartridge. If you have an expansion board, you can insert or remove a cartridge with the computer on, as long as the cartridge slot is off.

GLOSSARY OF TERMS

ACTIVE LOW - Refers to a control line that is normally high (+5 volts), but which goes low (grounded, 0 volts) to perform its assigned function. Most EPROM enable lines are active low, that is, they enable the EPROM when they are brought low and disable it when left high.

ADDRESS DECODING - The process of breaking down a memory address in order to select the correct chip where the data resides. This is how chips are assigned their locations in memory.

ADDRESS LINES - The lines over which addresses are sent to a chip. The chip selects a byte of data from within itself based on the address sent.

AUTOSTART CARTRIDGE - A cartridge that starts automatically when the computer is turned on or a RESET is done. Most commercial cartridges autostart themselves.

BANK-SWITCHING - A technique used in some cartridges that allows them to switch "banks" of memory (memory chips) in and out of the computer's memory space. This allows a bank-switch cartridge to have access to more memory than a standard cartridge, although not all at the same time. Cartridges are limited to 16K of memory at a time on the C64 (32K on the C128). Bank-switching is also used by the C64 and C128 themselves to switch system ROMS, I/O devices, cartridges, etc. in and out of memory.

BINARY - A number system that corresponds very closely to the way computers actually store information. Binary numbers use only the digits 0 and 1. Binary is also called base 2.

BIT - The smallest unit of information that a computer can deal with. "Bit" stands for **B**inary digI**T** - something that is always either a 0 or a 1 (OFF or ON).

BURNING AN EPROM - Another term for programming an EPROM - storing information on it.

BYTE - The next level of memory organization above the bit level. One byte consists of 8 bits that are always stored or retrieved together.

CARTRIDGE - Refers to the combination of a cartridge board and the memory chips that reside on it. A cartridge is essentially a plug-in memory expansion unit containing a program stored on some type of ROM (read-only memory).

CW - A PROMOS **C**ontrol **W**ord. The CW is used to tell PROMOS what type of chip is being read or written, so that PROMOS can tell the PROMENADE the correct voltages, control signals, pins, etc. to use. Using the wrong CW can permanently damage a chip.

DATA LINES - The lines over which data is sent to or received from a chip. There are usually 8 data lines, corresponding to the 8 bits in a byte. Sometimes the same physical lines are used for both data and addresses, a technique called multiplexing. This requires precise timing.

DECIMAL - Our ordinary everyday number system, also called base 10. Numbers are made up of digits selected from the range 0 to 9. A particular digit varies in value according to its place in the number.

DIP - Stands for **D**ual **I**n-line **P**ackage. It refers to the most common way of arranging the pins on a chip - in two straight rows. All EPROMs in this book are DIP chips (and no, you can't serve these DIP chips at a party).

EEPROM - Stands for **E**lectrically **E**rasable, **P**rogrammable **R**ead-**O**nly **M**emory. Basically, a memory chip that doesn't lose its data when the power is removed, but which can be erased using an electrical signal rather than ultraviolet light (distinguishing it from an EPROM).

ENABLE LINE - A line which controls a chip, turning it on or off as needed. The enable line is used by the address decoding circuitry to turn on a chip when a certain range of addresses is referenced. This is how a chip is assigned a location in memory. Other types of chips besides memory chips also have enable lines.

EPROM - Stands for **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory. Basically, a memory chip that doesn't lose its data when the power is removed, but which can be erased with ultraviolet light and reused.

EXPANSION BOARD - A device that plugs into the cartridge port and has one or more "slots" on it that cartridges can be plugged into. The expansion board will allow cartridges to be switched on and off independently of the computer.

EXROM - Pin 9 of the cartridge port. If a cartridge grounds just EXROM, the PLA will select the 8K standard cartridge configuration. If other control lines such as GAME are manipulated, one of the other cartridge memory configurations may be selected. A good expansion board will allow you to switch GAME and EXROM separately.

GAME - Pin 8 of the cartridge port. If a cartridge grounds just GAME, the PLA will select the ULTIMAX memory configuration. Other memory configurations can be selected by manipulating other control lines such as EXROM.

HEXADECIMAL - A useful way of expressing numbers when dealing with computers, also called base 16. Numbers are made up of digits selected from the range 0 to 9 and A to F (A=10, B=11, etc.). Hexadecimal can be very easily converted to binary (base 2), and vice versa.

HIRAM - A control line accessible through bit 1 of memory location $0001 (which actually resides in the 6510 microprocessor). Generally, this line switches the KERNAL, BASIC and cartridge ROMs in or out of memory together. Its effect also depend on other lines such as LORAM.

LORAM - A control line accessible through bit 0 of memory location $0001 (which actually resides in the 6510 microprocessor). Generally, this line switches the BASIC and cartridge ROMs in or out of memory together. Its effects depend on other lines such as HIRAM.

MACHINE LANGUAGE - Also called ML, this is the microprocessor's "native language". ML instuctions are stored in a form similar to binary, but are usually converted to hexadecimal for convenience. An even more convenient form is called assembly language, which uses names and symbols rather than numbers. ML instructions are less powerful but much faster than "high level" languages like BASIC. High level languages must be converted to ML before they can be executed.

MATRIX - The rectangular piece of silicon that actually holds the information in a memory chip. On EPROMS, it is visible through the quartz window on top.

PLA - A special **P**rogrammed **L**ogic **A**rray chip used on the C64 to manage memory. This chip is responsible for switching cartridges, system ROMS, RAM, I/O devices, etc. in and out of memory.

PMW - A PROMOS **P**rogram **M**ethod **W**ord. The PMW tells the PROMENADE which algorithm (procedure) to use when it programs a chip. While there is no "right" PMW for a chip, different PMW's will be vary in reliability.

PROM - A **P**rogrammable **R**ead-**O**nly **M**emory chip. PROMs can be programmed (written) by the user, but only once. They cannot be erased.

PROMENADE - The best EPROM programmer available for the C64, and probably the best for any microcomputer. Also works on the VIC20 and C128.

PROMOS - A special program provided with the PROMENADE. PROMOS adds several commands to the computer that can be used to read and write memory chips such as EPROMs.

PROGRAMMING AN EPROM - The process of storing information onto an EPROM, using a special device called an EPROM programmer (such as the PROMENADE).

RAM - **R**andom **A**ccess **M**emory, or memory in which any byte can be individually read or written to at any time. Usually considered the opposite of ROM.

RESET (HARD) - A RESET initiated through hardware, either on power-up (by a special circuit in the C64) or by the user (through a RESET switch). Automatically switches the system ROMs into memory and clears most I/O devices.

RESET (SOFT) - A RESET initiated through software, usually with a SYS 64738 (BASIC) or G FCE2 (monitor) command. Does not necessarily clear I/O devices or switch the system ROMs in.

ROM - **R**ead-**O**nly **M**emory. Strictly speaking, a memory chip which is written with data when it's manufactured and which can never be altered later. Also used loosely to refer to any non-RAM device, such as PROMs, EPROMs, etc.

ROMH - A control line of the cartridge port (pin B) that is used by the PLA to enable the second ROM (ROM high) in a cartridge. When the PLA brings this line low (grounds it), the chip is enabled and will respond to a memory request.

ROML - A control line of the cartridge port (pin 11) that is used by the PLA to enable the first ROM (ROM low) in a cartridge. When the PLA brings this line low (grounds it), the chip is enabled and will respond to a memory request.

ULTIMAX - A game machine that was discontinued by Commodore. The C64 has a special memory configuration that allows it to run ULTIMAX cartridges.

UV LIGHT - **U**ltra**V**iolet light. A particular wavelength of UV light (2537 Angstroms) is used to erase EPROMs. This light is hazardous to the skin and eyes.

DATA SHEETS

The following information is provided courtesy of the various manufacturers. We would like to thank them for this information and for their assistance throughout the book.

GENERAL TECHNICAL INFORMATION

ERASING THE Am2764

In order to clear all locations of their programmed contents, it is necessary to expose the Am2764 to an ultraviolet light source. A dosage of 15 Wseconds/cm^2 is required to completely erase an Am2764. This dosage can be obtained by exposure to an ultraviolet lamp [(wavelength of 2537 Angstroms (Å)] with intensity of 12000μW/cm^2 for 15 to 20 minutes. The Am2764 should be about one inch from the source and all filters should be removed from the UV light source prior to erasure.

It is important to note that the Am2764, and similar devices, will erase with light sources having wavelengths shorter than 4000 Angstroms. Although erasure times will be much longer than with UV sources at 2537Å, nevertheless the exposure to fluorescent light and sunlight will eventually erase the Am2764, and exposure to them should be prevented to realize maximum system reliability. If used in such an environment, the package window should be covered by an opaque label or substance.

PROGRAMMING THE Am2764

Upon delivery, or after each erasure the Am2764 has all 65536 bits in the "1", or high state. "0"s are loaded into the Am2764 through the procedure of programming.

The programming mode is entered when +21V is applied to the VPP pin. A 0.1μF capacitor must be placed across VPP and ground to suppress spurious voltage transients which may damage the device. The address to be programmed is applied to the proper address pins. 8-bit patterns are placed on the respective data output pins. The voltage levels should be standard TTL levels. When both the address and data are stable, a 50msec, TTL low level pulse is applied to the $\overline{PGM}$ input to accomplish the programming.

The procedure can be done manually, address by address, randomly, or automatically via the proper circuitry. All that is required is that one 50msec program pulse be applied at each address to be programmed. It is necessary that this program pulse width not exceed 55msec. Therefore, applying a DC low level to the $\overline{PGM}$ input is prohibited when programming.

READ MODE

The Am2764 has two control functions, both of which must be logically satisfied in order to obtain data at the outputs. Chip Enable ($\overline{CE}$) is the power control and should be used for device selection. Output Enable ($\overline{OE}$) is the output control and should be used to gate data to the output pins, independent of device selection. Assuming that addresses are stable, address access time (t_{ACC}) is equal to the delay from $\overline{CE}$ to output ($t_{\overline{CE}}$). Data is available at the outputs $t_{\overline{OE}}$ after the falling edge of $\overline{OE}$, assuming that $\overline{CE}$ has been low and addresses have been stable for at least $t_{ACC} - t_{\overline{OE}}$.

STANDBY MODE

The Am2764 has a standby mode which reduces the active power dissipation by 80%, from 525mW to 105mW (values for 0 to +70°C). The Am2764 is placed in the standby mode by applying a TTL high signal to the $\overline{CE}$ input. When in standby mode, the outputs are in a high impedance state, independent of the $\overline{OE}$ input.

OUTPUT OR-TIEING

To accommodate multiple memory connections, a 2 line control function is provided to allow for:

1. Low memory power dissipation
2. Assurance that output bus contention will not occur.

It is recommended that $\overline{CE}$ be decoded and used as the primary device selecting function, while $\overline{OE}$ be made a common connection to all devices in the array and connected to the READ line from the system control bus. This assures that all deselected memory devices are in their low-power standby mode and that the output pins are only active when data is desired from a particular memory device.

PROGRAM INHIBIT

Programming of multiple Am2764s in parallel with different data is also easily accomplished. Except for $\overline{PGM}$, all like inputs (including $\overline{OE}$) of the parallel Am2764s may be common. A TTL level program pulse applied to an Am2764's $\overline{PGM}$ input with VPP at 21V will program that Am2764. A high level $\overline{PGM}$ input inhibits the other Am2764s from being programmed.

PROGRAM VERIFY

A verify should be performed on the programmed bits to determine that they were correctly programmed. The verify must be performed with $\overline{OE}$ and $\overline{CE}$ at VIL. Data should be verified tOE after the falling edge of $\overline{OE}$.

SYSTEM APPLICATIONS

During the switch between active and standby conditions, transient current peaks are produced on the rising and falling edges of chip enable. The magnitude of these transient current peaks is dependent on the output capacitance loading of the device. A 0.1μF ceramic capacitor (high frequency, low inherent inductance) should be used on each device between V_{CC} and GND to minimize transient effects. In addition, to overcome the voltage droop caused by the inductive effects of the printed circuit board traces on EPROM arrays, a 4.7μF bulk electrolytic capacitor should be used between V_{CC} and GND for each eight devices. The location of the capacitor should be close to where the power supply is connected to the array.

TMS2532, SMJ2532
32,768-BIT ERASABLE PROGRAMMABLE READ-ONLY MEMORIES

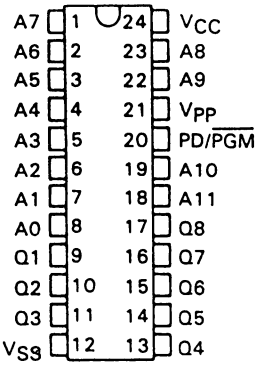
DECEMBER 1979 – REVISED SEPTEMBER 1983

- Organization . . . 4096 X 8
- Single +5-V Power Supply
- Pin Compatible with Existing ROMs and EPROMs (8K, 16K, 32K, and 64K)
- JEDEC Standard Pinout
- All Inputs/Outputs Fully TTL Compatible
- Static Operation (No Clocks, No Refresh)
- Max Access/Min Cycle Time:

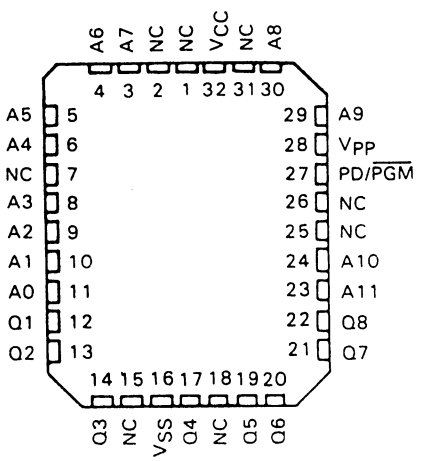
'2532-30	300 ns
'2532-35	350 ns
'2532-45	450 ns

- 8-Bit Output for Use in Microprocessor-Based Systems
- N-Channel Silicon-Gate Technology
- 3-State Output Buffers
- Low Power Dissipation:
  - Active . . . 400 mW Typical
  - Standby . . . 100 mW Standby
- Guaranteed DC Noise Immunity with Standard TTL Loads
- No Pull-Up Resistors Required
- JSP4 Version Available with 168 Hour Burn-in and Guaranteed Operating Temperature Range from 0°C to 85°C (TMS2532–__JSP4)
- Available in Full Military Temperature Range Version (SMJ2532)

TMS2532 . . . J PACKAGE
SMJ2532 . . . J PACKAGE
(TOP VIEW)



SMJ2532 . . . FE PACKAGE
(TOP VIEW)



PIN NOMENCLATURE	
A(N)	Address Inputs
NC	No Internal Connection
PD/PGM	Power Down/Program
Q(N)	Data Outputs
V_{CC}	+5-V Power Supply
V_{PP}	+25-V Supply
V_{SS}	0-V Ground

description

The '2532 series are 32,768-bit, ultraviolet-light-erasable, electrically-programmable read-only memories. These devices are fabricated using N-channel silicon-gate technology for high speed and simple interface with MOS and bipolar circuits. All inputs (including program data inputs) can be driven by Series 54/74 TTL circuits without the use of external pull-up resistors, and each output can drive one Series 54/74 TTL circuit without external resistors. The data outputs are three-state for connecting multiple devices to a common bus. The TMS2532 series are plug-in compatible with the TMS4732 32K ROM.

The TMS2532's are offered in a dual-in-line ceramic package (J suffix), rated for operation from 0°C to 70°C. The SMJ' devices are offered in a 24-pin dual-in-line ceramic package (J) and in a 32-pad leadless ceramic chip carrier (FE). The J package is designed for insertion in mounting-hole rows on 15,2 mm (600-mil) centers, whereas the FE package is intended for surface mounting on solder pads on 1,27 mm (0.050-inch) centers. The FE package offers a three-layer rectangular chip carrier with dimensions 11,42 × 13,97 × 2,54 mm (0.450 × 0.550 × 0.100 inches).

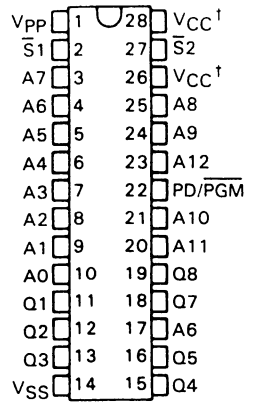
TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

TMS2564, SMJ2564
65,536-BIT ERASABLE PROGRAMMABLE READ-ONLY MEMORIES

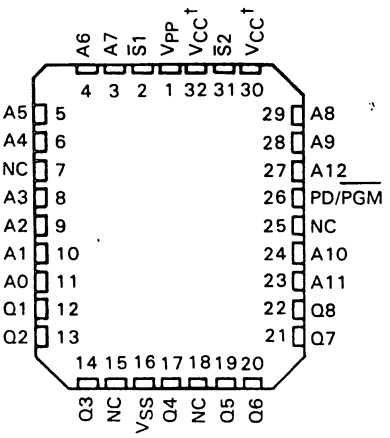
MAY 1981 – REVISED SEPTEMBER 1983

- Organization . . . 8192 X 8
- Single +5-V Power Supply
- Pin Compatible with Existing ROMs and EPROMs (8K, 16K, 32K, and 64K)
- All Input/Outputs Fully TTL Compatible
- Static Operation (No Clocks, No Refresh)
- Max Access/Min Cycle Time:
  - TMS2564-35 . . . 350 ns
  - TMS2564-45 . . . 450 ns
  - SMJ2564-45 . . . 450 ns
- 8-Bit Output for Use in Microprocessor-Based Systems
- N-Channel Silicon-Gate Technology
- 3-State Output Buffers
- Guaranteed DC Noise Immunity with Standard TTL Loads
- No Pull-Up Resistors Required
- Low Power Dissipation:
  - Active . . . 400 mW Typical
  - Standby . . . 125 mW Typical
- JSP4 Version Available with 168 Hour Burn-in and Guaranteed Operating Temperature Range from 0°C to 85°C (TMS2564-__JSP4)
- Available in Full Military Temperature Range Version (SMJ2564)

TMS2564 . . . J OR JD PACKAGE
SMJ2564 . . . J PACKAGE
(TOP VIEW)



SMJ2564 . . . FE PACKAGE
(TOP VIEW)



†Connected internally, V_{CC} need be supplied to only one of these two pins.

description

The '2564 is a 65,536-bit ultraviolet-light-erasable, electrically-programmable read-only memory. This device is fabricated using N-channel silicon-gate technology for high-speed and simple interface with MOS and bipolar circuits. All inputs (including program data inputs) can be driven by Series 54/74 TTL circuits without the use of external resistors. The data outputs are three-state for connecting multiple devices to a common bus.

PIN NOMENCLATURE	
A(N)	Address Inputs
NC	No Connection
PD/$\overline{PGM}$	Power Down/Program
Q(N)	Input/Output
$\overline{S}$(N)	Chip Selects
V_{CC}	+5-V Power Supply
V_{PP}	+25-V Power Supply
V_{SS}	0-V Ground

The TMS2564 is offered in a dual-in-line ceramic package (J or JD suffix) rated for operation from 0°C to 70°C. The SMJ2564 is offered in a 28-pin dual-in-line ceramic package (J) and a leadless ceramic chip carrier (FE), rated for operation from −55°C to 125°C. The J package is designed for insertion in mounting-hole rows on 15,2 mm (600-mil) centers, whereas the FE package is intended for surface mounting on solder pads on 1,27 mm (0.050-inch) centers. The FE package offers a three-layer rectangular chip carrier with dimensions 11,43 × 13,97 × 2,54 mm (0.450 × 0.550 × 0.100 inches).

ADVANCE INFORMATION
MILITARY PRODUCTS (SMJ) ONLY

This document contains information on a new product. Specifications are subject to change without notice.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

TMS2708, TMS27L08, SMJ2708, SMJ27L08
1024-WORD BY 8-BIT
ERASABLE PROGRAMMABLE READ-ONLY MEMORIES

DECEMBER 1979 - REVISED AUGUST 1983

- 1024 X 8 Organization
- All Inputs and Outputs Fully TTL Compatible
- Static Operation (No Clocks, No Refresh)
- Max Access/Min Cycle Time

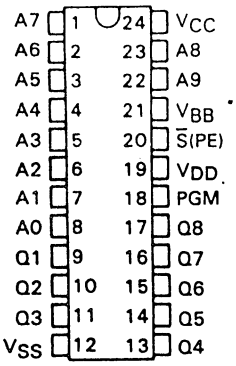
'2708-35	350 ns
'2708-45	450 ns
'27L08-45	450 ns

- 3-State Outputs for OR-Ties
- N-Channel Silicon-Gate Technology
- 8-Bit Output for Use in Microprocessor-Based Systems
- Power Dissipation

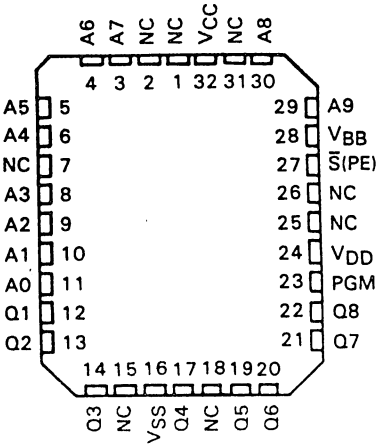
'27L08	580 mW Max Active
'2708	800 mW Max Active

- 10% Power Supply Tolerance (TMS27L08-45 and all SMJ' versions)
- Plug-Compatible Pin-Outs Allowing Interchangeability/Upgrade to 16K With Minimum Board Change
- Available in Full Military Temperature Range Versions (SMJ2708)

TMS2708 . . . J PACKAGE
SMJ2708 . . . J PACKAGE
(TOP VIEW)



SMJ2708 . . . FE PACKAGE
(TOP VIEW)



NC – No Connection

PIN NOMENCLATURE	
A0-A7	Address Inputs
NC	No Connection
PGM	Program
Q1-Q8	Data Out
S(PE)	Chip Select/Program Enable
VBB	−5-V Power Supply
VCC	+5-V Power Supply
VDD	+12-V Power Supply
VSS	0-V Ground

description

The '2708-35, '2708-45, and '27L08-45 are ultraviolet light-erasable, electrically programmable read-only memories. They have 8,192 bits organized as 1024 words of 8-bit length. The devices are fabricated using N-channel silicon-gate technology for high speed and simple interface with MOS and bipolar circuits. All inputs (including program data inputs) can be driven by Series 54/74 TTL circuits without the use of external pull-up resistors. Each output can drive one Series 54/74 or 54LS/74LS TTL circuit without external resistors. The '27L08 guarantees 200 mV dc noise immunity in the high state and 250 mV in the low state. The data outputs for the '2708-35, '2708-45, and '27L08-45 are three-state for OR-tying multiple devices on a common bus.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

THOMSON SEMICONDUCTORS

PRELIMINARY

ETC 2716 16,384-BIT (2048 x 8) UV Erasable CMOS PROM

Parameter/Part Number	ETC 2716-1	ETC 2716	ETC 2716-5	ETC 2716-6
Access Time (ns)	350	450	550	650
Active Current (mA @ 1 MHz)	5	5	5	5
Standby Current (mA)	0,1	0,1	0,1	0,1

General Description

The ETC 2716 is a high speed 16k UV erasable and electrically reprogrammable CMOS EPROM ideally suited for applications where fast turn-around, pattern experimentation and low power consumption are important requirements.

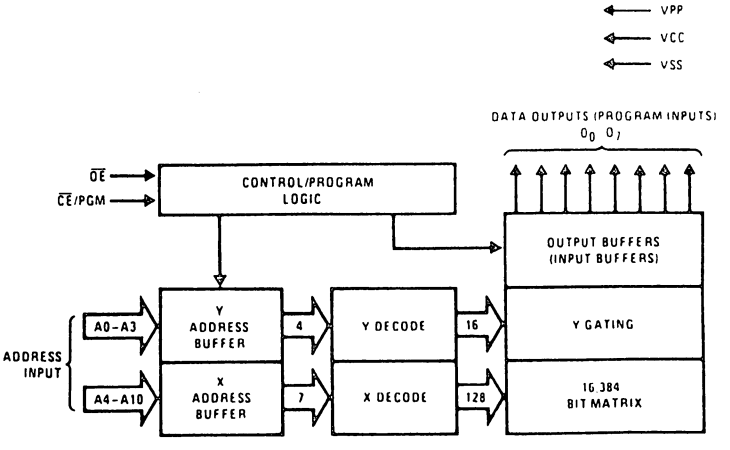
The ETC 2716 is packaged in a 24-pin dual-in-line package with transparent lid. The transparent lid allows the user to expose the chip to ultraviolet light to erase the bit pattern. A new pattern can then be written into the device by following the programming procedure.

ThisEPROM is fabricated with the reliable, high volume, time proven, P²CMOS silicon gate technology.

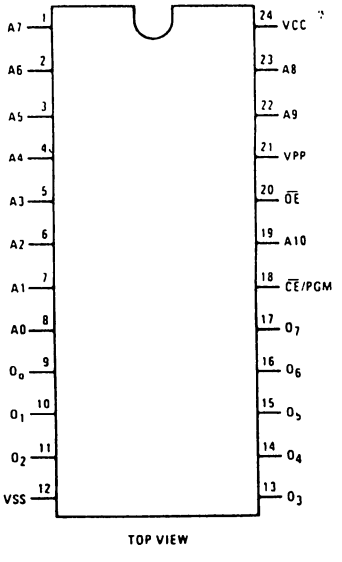
Features

- CMOS power consumption
- Performance compatible to ETC800 CMOS Microprocessor
- 2048 x 8 organization
- Pin compatible to 2716
- Access time down to 350 ns
- Single 5V power supply
- Static - no clocks required
- Inputs and outputs TTL compatible during both read and program modes
- Three-state output with OR-tie capability

Block and Connection Diagrams



Dual-In-Line Package



TOP VIEW

Pin Connection During Read or Program

Mode	Pin Name/Number				
	CE/PGM 18	OE 20	VPP 21	VCC 24	Outputs 9-11, 13-17
Read	VIL	VIL	5	5	DOUT
Program	Pulsed VIL to VIH	VIH	25	5	DIN

Pin Names

A0-A10	Address Inputs
O_0-O_7	Data Outputs
CE/PGM	Chip Enable/Program
OE	Output Enable
VPP	Read 5V, Program 25V
VCC	5V
VSS	Ground

THOMSON SEMICONDUCTORS
U.S. HEADQUARTERS
301 Route 17 North
RUTHERFORD, NJ 07070 - Tel. (201) 438-2300

THOMSON COMPONENTS

32,768-BIT ERASABLE PROGRAMMABLE READ-ONLY MEMORY

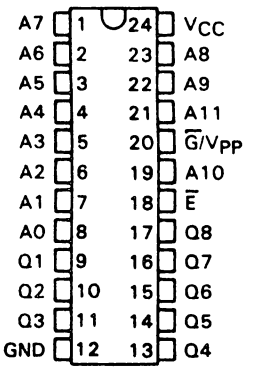
AUGUST 1983 – REVISED SEPTEMBER 1984

- **Organization . . . 4096 X 8**
- **Single 5-V Power Supply**
- **All Inputs and Outputs Are TTL Compatible**
- **Performance Ranges:**

	ACCESS TIME (MAX)	CYCLE TIME (MIN)
TMS2732A-25	250 ns	250 ns
TMS2732A-30	300 ns	300 ns
TMS2732A-35	350 ns	350 ns
TMS2732A-45	450 ns	450 ns

- **Low Standby Power Dissipation . . . 158 mW (Maximum)**
- **JEDEC Approved Pinout . . . Industry Standard**
- **21-V Power Supply Required for Programming**
- **N-Channel Silicon-Gate Technology**
- **8-Bit Output for Use in Microprocessor Based Systems**
- **Static Operation (No Clocks, No Refresh)**
- **JSP4 Version Available with 168 Hour Burn-in and Guaranteed Operating Temperature Range from 0°C to 85°C (TMS2732–__JSP4)**
- **Available with MIL-STD-883B Processing and L(0°C to 70°C), E(–40°C to 85°C), or S(–55°C to 100°C) Temperature Ranges in the Future**

TMS2732A . . . J PACKAGE
(TOP VIEW)



PIN NOMENCLATURE	
A0 – A11	Addresses
$\overline{E}$	Chip Enable
$\overline{G}/V_{PP}$	Output Enable/21 V
Q1 – Q8	Outputs
V_{CC}	5-V Power Supply

description

The TMS2732A is an ultraviolet light-erasable, electrically programmable read-only memory. It has 32,768 bits organized as 4,096 words of 8-bit length. The TMS2732A only requires a single 5-volt power supply with a tolerance of ±5%.

The TMS2732A provides two output control lines: Output Enable ($\overline{G}$) and Chip Enable ($\overline{E}$). This feature allows the $\overline{G}$ control line to eliminate bus contention in multibus microprocessor systems. The TMS2732A has a power-down mode that reduces maximum power dissipation from 657 mW to 158 mW when the device is placed on standby.

This EPROM is supplied in a 24-pin dual-in-line ceramic package and is designed for operation from 0°C to 70°C.

ADVANCE INFORMATION

This document contains information on a new product. Specifications are subject to change without notice.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

TMS2764
65,536-BIT ERASABLE PROGRAMMABLE READ-ONLY MEMORY

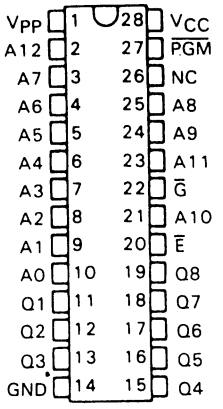
JULY 1983 – REVISED JANUARY 1984

- **Organization . . . 8192 X 8**
- **Single +5-V Power Supply**
- **Pin Compatible with TMS2732A EPROM**
- **All Inputs and Outputs are TTL Compatible**
- **Performance Ranges:**

	MAX ACCESS/ MIN CYCLE TIME
TMS2764-20	200 ns
TMS2764-25	250 ns
TMS2764-30	300 ns
TMS2764-35	350 ns
TMS2764-45	450 ns

- **Low Active Current . . . 100 mA (Max)**
- **JEDEC Approved Pinout**
- **21-V Power Supply Required for Programming**
- **Fast Programming Algorithm**
- **N-Channel Silicon-Gate Technology**
- **8-Bit Output for Use in Microprocessor-Based Systems**
- **Static Operation (No Clocks, No Refresh**
- **JSP4 Version Available with 168 Hour Burn-in and Guaranteed Operating Temperature Range from 0°C to 85°C (TMS2764-__JSP4)**
- **Available with MIL-STD-883B Processing and L(0°C to 70°C), E(−40°C to 85°C), or M(−55°C to 125°C) Temperature Ranges in the Future**

TMS2764 . . . J PACKAGE
(TOP VIEW)



PIN NOMENCLATURE	
A0-A12	Addresses
$\overline{E}$	Chip Enable
$\overline{G}$	Output Enable
GND	Ground
NC	No Connection
$\overline{PGM}$	Program
Q1-Q8	outputs
V_{CC}	+5-V Power Supply
V_{PP}	+21-V Power Supply

description

The TMS2764 is an ultraviolet light-erasable, electrically programmable read-only memory. It has 65,536 bits organized as 8,192 words of 8-bit length. The TMS2764-20 only requires a single 5-volt power supply with a tolerance of ±5%, and has a maximum access time of 200 ns. This access time is compatible with high-performance microprocessors.

The TMS2764 provides two output control lines: Output Enable ($\overline{G}$) and Chip Enable ($\overline{E}$). This feature allows the $\overline{G}$ control line to eliminate bus contention in microprocessor systems. The TMS2764 has a power-down mode that reduces maximum active current from 100 mA to 35 mA when the device is placed on standby.

This EPROM is supplied in a 28-pin, 15,2 mm (600-mil) dual-in-line ceramic package and is designed for operation from 0°C to 70°C.

ADVANCE INFORMATION

This document contains information on a new product. Specifications are subject to change without notice.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

27128A
Advanced 128K (16 x 8) UV Erasable PROM

- **Fast 150 nsec Access Time**
  — HMOS* II-E Technology
- **Low Power**
  — 100 mA Maximum Active
  — 40 mA Maximum Standby
- **Two Line Control**
- **Int$_e$ligent Programming™ Algorithm**
  — Fastest EPROM Programming
- **Int$_e$ligent Identifier™ Mode**
  — Automated Programming Operations
- **Compatible with 2764A, 27128, 27256**
- **±10% V_{CC} Tolerance Available**

The Intel 27128A is a 5V only, 131,072-bit ultraviolet erasable and electrically programmable read-only memory (EPROM). The 27128A is an advanced version of the 27128 and is fabricated with Intel's HMOSII-E technology which significantly reduces die size and greatly improves the device's performance, reliability and manufacturability.

The 27128A is available in fast access times including 200 ns (27128A-2) and 150 ns (27128A-1). This ensures compatibility with high-performance microprocessors, such as Intel's 8 MHz iAPX 186 allowing full speed operation without the addition of WAIT states. The 27128A is also directly compatible with the 12 MHz 8051 family.

Several advanced features have been designed into the 27128A that allow fast and reliable programming—the int$_e$ligent Programming Algorithm and the int$_e$ligent Identifier Mode. Programming equipment that takes advantage of these innovations will electronically identify the 27128A and then rapidly program it using an efficient programming method.

Two-line control and JEDEC-approved, 28 pin packaging are standard features of all Intel higher density EPROMs. This ensures easy microprocessor interfacing and minimum design efforts when upgrading, adding or choosing between non-volatile memory alternatives.

*HMOS is a patented process of Intel Corporation

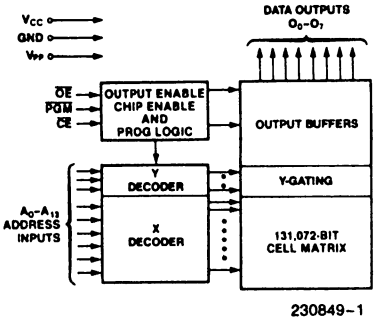
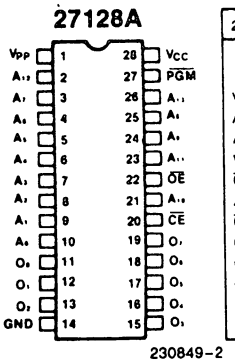


Figure 1. Block Diagram

27256	2764A	2732A	2716
V_{PP}	V_{PP}		
A_{12}	A_{12}		
A_7	A_7	A_7	A_7
A_6	A_6	A_6	A_6
A_5	A_5	A_5	A_5
A_4	A_4	A_4	A_4
A_3	A_3	A_3	A_3
A_2	A_2	A_2	A_2
A_1	A_1	A_1	A_1
A_0	A_0	A_0	A_0
O_0	O_0	O_0	O_0
O_1	O_1	O_1	O_1
O_2	O_2	O_2	O_2
Gnd	Gnd	Gnd	Gnd



2716	2732A	2764A	27256
		V_{CC}	V_{CC}
		PGM	A_{14}
V_{CC}	V_{CC}	N.C	A_{13}
A_8	A_8	A_8	A_8
A_9	A_9	A_9	A_9
V_{PP}	A_{11}	A_{11}	A_{11}
OE	OE/V_{PP}	OE	OE
A_{10}	A_{10}	A_{10}	A_{10}
CE	CE	CE	CE
O_7	O_7	O_7	O_7
O_6	O_6	O_6	O_6
O_5	O_5	O_5	O_5
O_4	O_4	O_4	O_4
O_3	O_3	O_3	O_3

NOTE: Intel "Universal Site"-Compatible EPROM Pin Configurations are Shown in the Blocks Adjacent to the 27128A Pins

Figure 2. Pin Configurations

Mode Selection

Mode \ Pins	CE (20)	OE (22)	PGM (27)	A_9 (24)	V_{PP} (1)	V_{CC} (28)	Outputs (11-13, 15-19)
Read	V_{IL}	V_{IL}	V_{IH}	X	V_{CC}	V_{CC}	D_{OUT}
Output Disable	V_{IL}	V_{IH}	V_{IH}	X	V_{CC}	V_{CC}	High Z
Standby	V_{IH}	X	X	X	V_{CC}	V_{CC}	High Z
Verify	V_{IL}	V_{IL}	V_{IH}	X	V_{PP}	V_{CC}	D_{OUT}
Program Inhibit	V_{IH}	X	X	X	V_{PP}	V_{CC}	High Z
Int$_e$ligent Identifier	V_{IL}	V_{IL}	V_{IH}	V_H	V_{CC}	V_{CC}	Code
Int$_e$ligent Programming	V_{IL}	V_{IH}	V_{IL}	X	V_{PP}	V_{CC}	D_{IN}

1. X can be V_{IH} or V_{IL}
2. $V_H = 12.0V \pm 0.5V$

Pin Names

A_0-A_{13}	ADDRESSES
CE	CHIP ENABLE
OE	OUTPUT ENABLE
O_0-O_7	OUTPUTS
PGM	PROGRAM

April 1984
Order Number: 230849-002

VT27C256 PRELIMINARY

32,768 x 8 STATIC CMOS EPROM

FEATURES

- 32,768 x 8-bit organization
- Current—Operating: 80 mA max
  Standby: 200 μA max
- Total static operation
- Automatic power down ($\overline{CE}$)
- Complete TTL compatibility
- 3-state outputs for wired-OR expansion
- 28-pin JEDEC approved pinout
- Programming—Voltage: +12V
  Current: 30 mA

DESCRIPTION

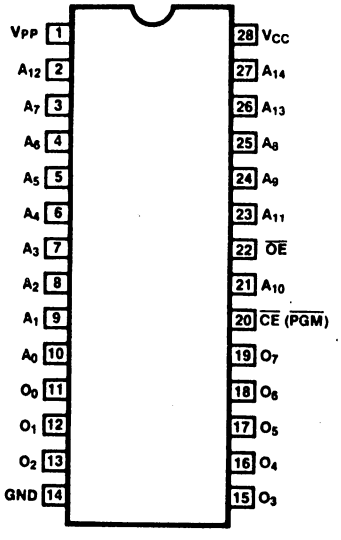
The VT27C256 high-performance CMOS Erasable Programmable Read Only Memory is organized as 32,768 bytes and has an access time of 200 ns. It is compatible with all microprocessors and similar high-performance applications where high-density storage reprogrammability and simple interfacing are important design considerations.

The VT27C256 has automatic power-down which is controlled by the Chip Enable ($\overline{CE}$) input. When $\overline{CE}$ goes HIGH, the device automatically powers down and remains in a low-power standby mode. This unique feature provides substantial systems-level power savings. Another function of the VT27C256 is the Output Enable ($\overline{OE}$) which eliminates bus contention.

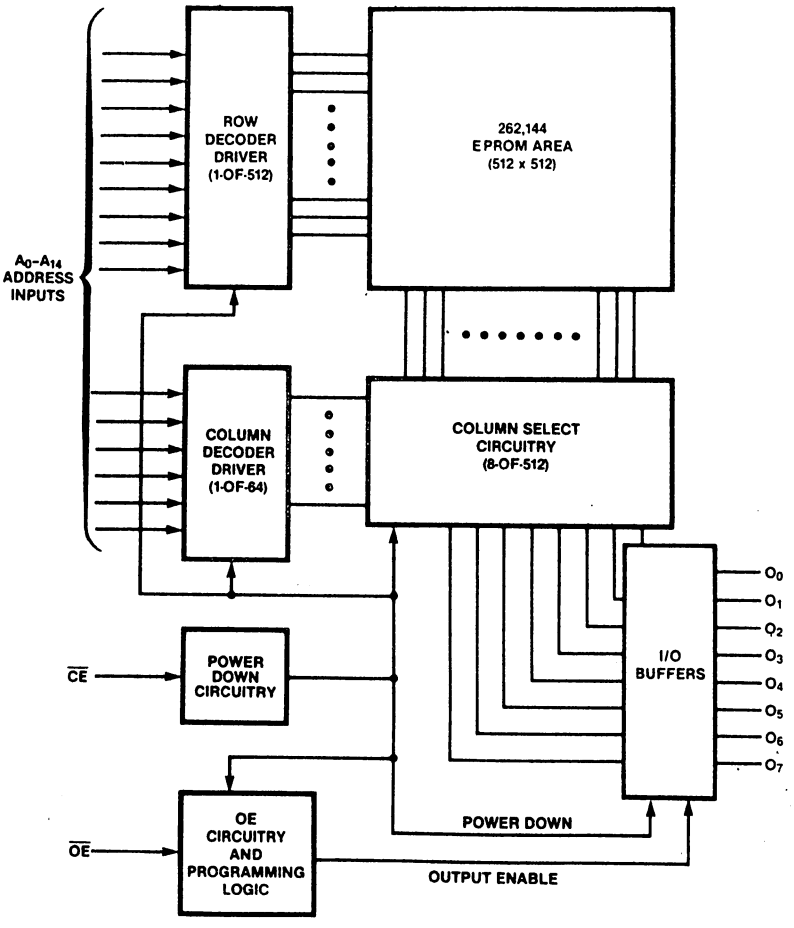
The VT27C256 is manufactured with VTI's advanced high-performance HCMOS process and is available in a JEDEC-standard 28-pin dual in-line package.

PIN CONFIGURATIONS

VT27C256



BLOCK DIAGRAM



VLSI Technology, Inc. • 1109 McKay Drive • San Jose, CA 94131 • 408-942-1810

27512
512K (64K x 8) UV ERASABLE PROM

- **Software Carrier Capability**
- **250 ns Standard Access Time**
- **Two-Line Control**
- **int$_e$ligent Identifier™ Mode**
  — Automated Programming Operations
- **TTL Compatible**
- **Industry Standard Pinout . . . JEDEC Approved**
- **Low Power**
  — 125 mA max. Active
  — 40 mA max. Standby
- **int$_e$ligent Programming™ Algorithm**
  — Fastest EPROM Programming

The Intel 27512 is a 5V only, 524, 288 bit ultraviolet Erasable and Electrically Programmable Read Only Memory (EPROM). Organized as 64K words by 8 bits, individual bytes are accessed in under 250ns. This ensures compatibility with high performance microprocessors, such as the Intel 8MHz iAPX 186, allowing full speed operation without the addition of performance-degrading WAIT states. The 27512 is also directly compatible with Intel's 8051 family of microcontrollers.

The 27512 enables implementation of new, advanced systems with firmware intensive architectures. The combination of the 27512's high density, cost effective EPROM storage, and new advanced microprocessors having megabyte addressing capability provides designers with opportunities to engineer user friendly, high reliability, high-performance systems.

The 27512's large storage capability of 64K bytes enables it to function as a high density software carrier. Entire operating systems, diagnostics, high-level language programs and specialized application software can reside in a 27512 EPROM directly on a system's memory bus. This permits immediate microprocessor access and execution of software and eliminates the need for time consuming disk accesses and downloads.

Two advanced features have been designed into the 27512 that allow for fast and reliable programming — the int$_e$ligent identifier™ mode and the int$_e$ligent Programming™ Algorithm. Programming equipment that takes advantage of these innovations will electronically identify the 27512 and then rapidly program it using an efficient programming method.

Two-line control and JEDEC-approved, 28-pin packaging are standard features of all Intel high-density EPROMs. This assures easy microprocessor interfacing and minimum design efforts when upgrading, adding, or choosing between nonvolatile memory alternatives.

The 27512 is manufactured using Intel's advanced HMOS *II-E technology.

*HMOS is a patented process of Intel Corporation.

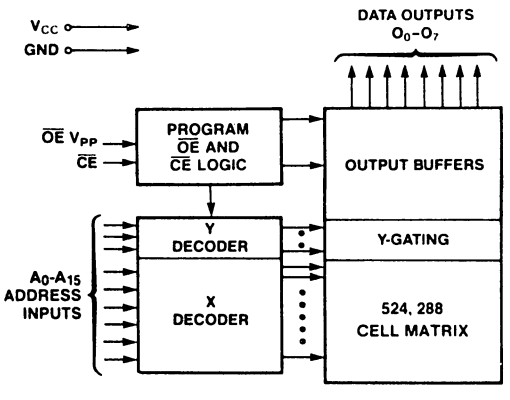
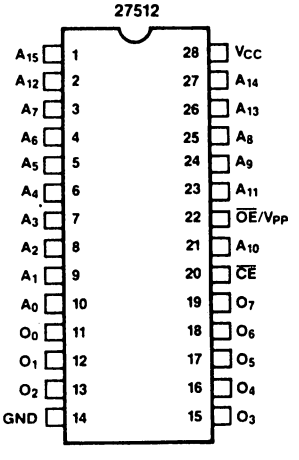


Figure 1: Block Diagram



PIN NAMES	
A_0-A_{15}	ADDRESSES
CE	CHIP ENABLE
$\overline{OE}/V_{PP}$	OUTPUT ENABLE/V_{PP}
O_0-O_7	OUTPUTS

Figure 2. Pin Configuration

MAY 1984

MOTOROLA

SEMICONDUCTORS

3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721

MCM68764

MOS

(N-CHANNEL, SILICON-GATE)

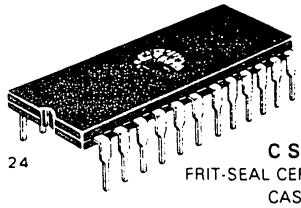
8192 × 8-BIT UV ERASABLE PROGRAMMABLE READ ONLY MEMORY

64K-BIT UV ERASABLE PROM

The MCM68764 is a 65,536-bit Erasable and Electrically Reprogrammable PROM designed for system debug usage and similar applications requiring nonvolatile memory that could be reprogrammed periodically, or for replacing 64K ROMs for fast turnaround time. The transparent window on the package allows the memory content to be erased with ultraviolet light.

For ease of use, the device operates from a single power supply and has a static power-down mode. Pin-for-pin mask programmable ROMs are available for large volume production runs of systems initially using the MCM68764.

- Single +5 V Power Supply
- Automatic Power-down Mode (Standby) with Chip Enable
- Organized as 8192 Bytes of 8 Bits
- Power Dissipation
  - 120 mA Active Maximum
  - 25 mA Standby Maximum
- Fully TTL Compatible
- Maximum Access Time = 450 ns MCM68764
  - 350 ns MCM68764-35
- Standard 24-Pin DIP for EPROM Upgradability
- Pin Compatible to MCM68A364 Mask Programmable ROM

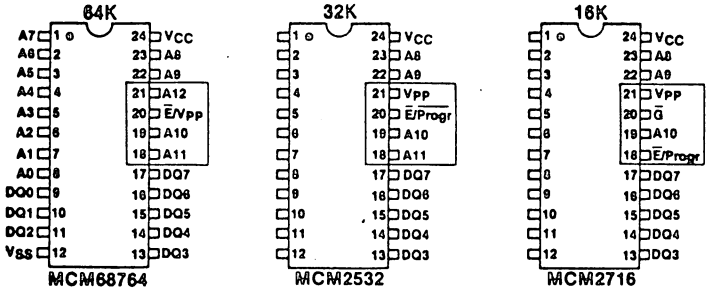


24

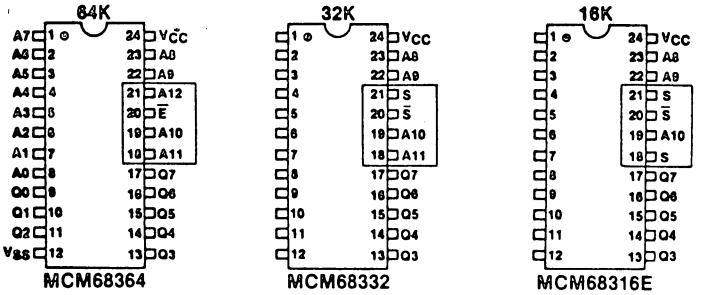
C SUFFIX
FRIT-SEAL CERAMIC PACKAGE
CASE 623A

L SUFFIX CERAMIC PACKAGE
ALSO AVAILABLE — CASE 716

MOTOROLA'S PIN-COMPATIBLE EPROM FAMILY



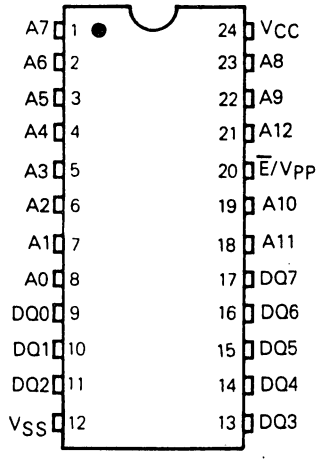
MOTOROLA'S PIN-COMPATIBLE ROM FAMILY



INDUSTRY STANDARD PINOUTS

AA00090-2

PIN ASSIGNMENT



Pin Names	
A	Address
DQ	Data Input/Output
$\overline{E}/V_{PP}$	Chip Enable/Program

MOTOROLA
SEMICONDUCTORS
3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721

MCM68766

Advance Information

8192 × 8-BIT UV ERASABLE PROM

The MCM68766 is a 65,536-bit Erasable and Electrically Reprogrammable PROM designed for system debug usage and similar applications requiring nonvolatile memory that could be reprogrammed periodically, or for replacing 64K ROMs for fast turnaround time. The transparent window on the package allows the memory content to be erased with ultraviolet light.

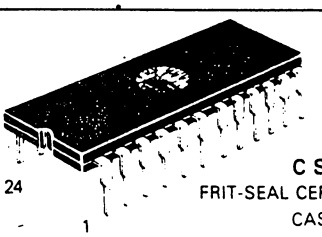
For ease of use, the device operates from a single power supply that has an output enable control and is pin-for-pin compatible with the MCM68366 mask programmable ROMs, which are available for large volume production runs of systems initially using the MCM68766.

- Single +5 V Power Supply
- Organized as 8192 Bytes of 8 Bits
- Fully TTL Compatible
- Maximum Access Time = 450 ns MCM68766
  350 ns MCM68766-35
- Standard 24-Pin DIP for EPROM Upgradability
- Pin Compatible to MCM68366 Mask Programmable ROM
- Power Dissipation — 160 mA Maximum

MOS

(N-CHANNEL, SILICON-GATE)

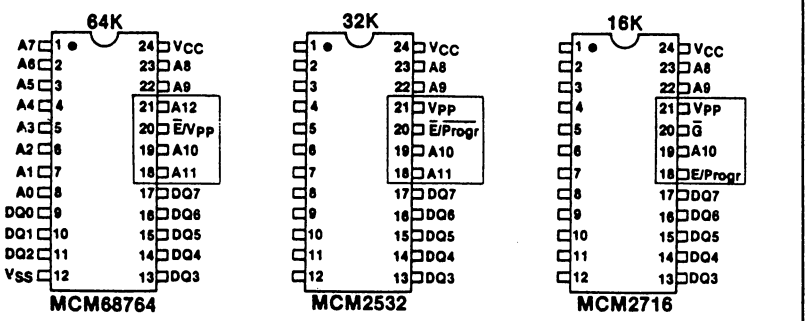
8192 × 8-BIT
UV ERASABLE
PROGRAMMABLE READ ONLY
MEMORY



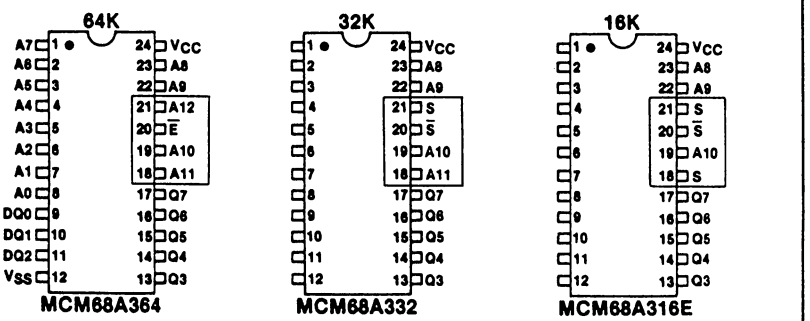
C SUFFIX
FRIT-SEAL CERAMIC PACKAGE
CASE 623A

L SUFFIX CERAMIC PACKAGE
ALSO AVAILABLE — CASE 716

MOTOROLA'S PIN-COMPATIBLE EPROM FAMILY



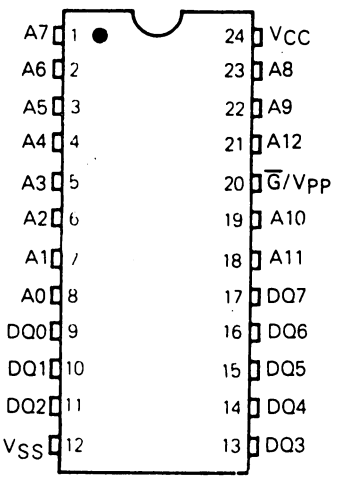
MOTOROLA'S PIN-COMPATIBLE ROM FAMILY



INDUSTRY STANDARD PINOUTS

AA00090-1

PIN ASSIGNMENT



*Pin Names	
A	Address
DQ	Data Input/Output
G/VPP	Output Enable/ Program

*New industry standard nomenclature

SPECTRONICS CORPORATION
956 Brush Hollow Road, P.O. Box 483
Westbury, New York 11590
(Tel.) 516-333-4840 • (TWX) 510 222 5877

TECHNICAL BULLETIN

ULTRAVIOLET ERASING OF EPROM'S

The earliest electronic computers used a combination of semiconductor diodes and vacuum tubes to perform logic functions, and magnetic core circuits to perform memory functions. The initial transition to integrated circuits still left main frame computers in the form of magnetic core stacks—this was the last area within the computer to adopt solid state circuitry. The main reason was economic as core memory systems could be built at a cost less than that of other methods available at the time.

Rapid advances in Large Scale Integration (LSI) techniques and the decreasing cost of semiconductors in general reversed the conditions favoring core memory in the early 1970's. Increasing labor costs for the fabrication of core memory and the widespread proliferation and availability of low cost Metal-Oxide-Semiconductor (MOS) devices reinforced the trend in the mid 1970's towards semiconductors being the primary memory technology. Also, the inherent reliability of semiconductor memories led to their use in high performance applications in which core memory had proved unsatisfactory.

Now, the new memories are no longer limited to computers and have become a universal design element that is introducing the benefits of data storage to all electronics equipment industries.

WHAT IS A SEMICONDUCTOR MEMORY?

The basic configuration of a semiconductor memory consists of a matrix of "cells" where data is stored. Various types of memory storage have evolved—from simple diodes arranged in a matrix to gate and logic arrays, stored-charge devices and amorphous semiconductors.

Erasable memories use active transistor circuits as memory cells and are generally fabricated by MOS technology. MOS devices are classified according to different fabrication techniques, i.e. p-channel (P-MOS), n-channel (N-MOS) and complementary (C-MOS).

Two major classes of semiconductor memory have developed: Random Access Memory (RAM) and Read Only Memory (ROM). RAM is employed to store and retrieve digital data that is constantly changing in computer systems; ROM serves as a source of permanent data that is not lost when power is removed from the system.

Different system applications with their specific data storage applications have spawned the development of various categories of RAM and ROM memories, each with its own peculiar advantages and disadvantages. Only one of these will be addressed in this bulletin—the Ultraviolet-Erasable Programmable ROM, often abbreviated as EPROM or UV-PROM.

ROM generally signifies a mask-programmed ROM which obtains storage ability by virtue of the semiconductor manufacturing process. Data is permanently stored when the device is manufactured and cannot be altered. If an error was made in the mask (and the permanently programmed data), it almost invariably leads to the scrapping of the device. The procedure is also time-consuming and certainly not very well matched to today's rapidly changing technological advances. PROM's employ techniques that allow the complete device to be manufactured without programmed data and provide the user with access to input data. Data is inputted by a PROM programmer which accepts data either manually by keyboard or automatically from tape. Data is thus converted into the appropriate electrical format for the PROM being used. The nature of the PROM memory cells and the techniques required for programming vary with construction technology but always, unfortunately, once the bit patterns have been programmed into the memories they are permanent. The programmed PROM results in a device that is similar to a mask-programmed ROM — with the same disadvantages.

THE UV-ERASABLE PROM (EPROM)

The UV-erasable PROM has a fused silica lid positioned above the semiconductor chip, which permits ultraviolet radiation to penetrate the semiconductor. The EPROM is erased in this way rather than electrically as with RAM's, some ROM's and most recently with Electrically-Erasable-PROM's (EEPROM), also known as Electrically-Alterable-PROM's (EAPROM).

WHAT ADVANTAGES DO EPROM'S HAVE?

The attraction of the EPROM is its extreme usefulness in prototyping software codes in microprocessor designs. It typically takes several "passes" through a system before a program's code can be correct and optimized. Each "pass" requires a new program and a ROM that can be erased and reprogrammed quickly makes it much easier to optimize a program. For a complex project, UV-erasable PROM's can save months of trial-and-error programming.

An additional dimension of flexibility has been added to systems design because, even when the programmer is satisfied with the program, it can be changed at a later date to accommodate new systems features. Alternatively, the user can switch to mask-programmed ROM's or can program ROM's with identical pin assignments and power supply requirements.

In practice EPROM's are being increasingly designed as a permanent part of a system because of the severe price war that has resulted in prices that make them very competitive with the less flexible, non-erasable PROM's and ROM's. Lengthy turnaround times for custom-programmed circuits are avoided and they are now appearing in many unanticipated small volume production applications (even though they may never be programmed again), as well as for prototyping. Once the right program has been found, other UV-erasable PROM's may be programmed to duplicate the bit pattern of the prototype.

Ultimately, ROM's manufactured from specific masks to duplicate the bit pattern would be substituted for large production runs. In the latter case, the experimental EPROM would then be erased and moved to another experimental location.

HOW DO EPROM'S WORK?

The basic memory element was developed by Frohman-Bentchkowsky at Intel Corporation and was known as the Floating-Gate-Avalanche-Injection MOS (FAMOS) transistor. It was essentially a silicon gate MOS field effect transistor in which no connection was made to the gate. The gate was in fact electrically "floating" in an insulating layer of silicon dioxide. The devices have been fabricated in two structures: p-channel or n-channel. The p-channel devices were the first EPROM's available commercially, but many devices are now using n-channel technology. N-channel MOS devices have the advantage of being able to function with a single power supply.

By application of a sufficiently large potential difference between the source and drain, charge can be injected into the "floating" gate which induces a charge in the substrate. The source-to-drain impedance changes and a "p-channel" or "n-channel" is created, depending upon the type of substrate. The presence or absence of conduction is the principle of data storage. Application of short wave (254nm) ultraviolet light causes the gate charge to leak away and restores the device to its original unprogrammed state.

WHAT IS ULTRAVIOLET LIGHT?

Ultraviolet light is an invisible band of radiation beyond the violet end of the electromagnetic spectrum. It extends from a wavelength of 0.1 to about 380nm (nanometers), although the region between 180 and 380nm is the only part that is normally experienced because of the absorption of shorter wavelengths by air. The wavelength of specific interest in the UV-EPROM application is 253.7nm, which is usually rounded off to 254nm.

HOW MUCH ULTRAVIOLET LIGHT IS NEEDED FOR ERASURE?

The time required for complete erasure of the information on a cell is very consistent from device to device, provided that the fused silica window is clean and there are no pieces of dirt, grease or silicon fragments on the surface of the EPROM. In these cases the shadows reduce the amount of ultraviolet light falling on those cells obscured by the obstacle, thus increasing the time required for complete erasure. Awareness of the problems, and screening followed by careful cleaning of the fused silica window, can minimize the required erase times. For maximum irradiance at the chip, the tube or grid of the eraser should also be kept clean of grease and other ultraviolet-absorbing materials by periodically wiping it down with a solvent such as alcohol.

Most EPROM chip manufacturers have assigned **nominal erasing energies** to their devices to assist the user in determining the optimum erase time. Manufacturers usually specify an erasing energy of 15W-sec/cm², although many EPROM's require only 10 or even 6W-sec/cm² for complete erasure.

The manufacturer's recommended erasing time in seconds is obtained by multiplying the **nominal erasing energy** (W-sec/cm²) by 1,000,000 and dividing by the **ultraviolet irradiance** at the chip (in microwatts/cm²).

$$\text{Time (secs)} = \frac{\text{Nominal Erasing Energy (W-sec/cm}^2\text{) x 1,000,000}}{\text{Irradiance } (\mu\text{W/cm}^2)}$$

SIMULTANEOUS ERASING OF SEVERAL EPROM'S

The published erasure times given by some manufacturers of EPROM erasers are optimum times based on the ideal situation of one chip at the point of maximum irradiance. Because of the inadequate explanation of this fact, some confusion has arisen when simultaneously erasing several EPROM's, and incomplete erasure of one or more chips has occurred. The programmer has been uncertain as to whether the eraser was functioning incorrectly or that perhaps the chips were bad. There could be an alternative explanation and it is the purpose of this bulletin to explore the situation when erasing several EPROM's simultaneously.

The irradiance produced at the surface of the EPROM chip is dependent upon **where it is located with respect to the tube** of the eraser. For example, if it is located off to the side and near to the end of the tube it receives considerably less UV irradiance than a chip located directly below the center of the tube. Usually this latter location is that chosen by lamp manufacturers when specifying erasure times. However, these figures are usually meaningless if the eraser is loaded with more than one EPROM, because obviously only one chip can receive the maximum irradiance. The parameter is the time needed for erasing **all** of the EPROM's — including the worst placed EPROM (i.e. the chip receiving the least amount of 254nm energy per unit time).

The following table provides data concerning **estimated erasing times** for the least favorably situated chip in each of the ten Spectroline® EPROM erasers:

ERASING TIME REQUIRED, IN MINUTES

NOMINAL ERASING ENERGY	PE-14, PE-14T			PE-24T			PL-265T			PR-125T, PR-320T	PC-1100, PC-2200 PC-3300, PC-4400
	1 EPROM CHIP	3 EPROM CHIPS	9 EPROM CHIPS	2 EPROM CHIPS	4 EPROM CHIPS	12 EPROM CHIPS	2 EPROM CHIPS	5 EPROM CHIPS	30 EPROM CHIPS	1 CHIP TO FULL CAPACITY	1 CHIP TO FULL CAPACITY
6W-sec/cm²	12.5	13.8	15.3	10.4	10.9	13.6	10.4	10.9	13.6	5.9	5.4
10W-sec/cm²	20.8	22.9	25.6	17.4	18.1	22.7	17.4	18.1	22.7	9.8	9.0
15W-sec/cm²	31.2	34.5	38.4	26.0	27.2	34.1	26.0	27.2	34.1	14.7	13.5

IMPORTANT: The above erasing times are based upon typical peak irradiances of the erasers and are given only as estimates — **actual erasing times may vary**. The suggested operating procedure is to use the above table to select the estimated erasing time and then vary the exposure period until you determine the **minimum** amount of time required for complete erasure.

As with all metal vapor discharge lamps, the output of the ultraviolet tubes and grids in our EPROM erasers will gradually decrease throughout their life. Due to the general unpredictability of the rate of UV intensity decline, cumulative hour recorders in EPROM erasers cannot accurately indicate tube life. Use of a short wave UV radiometer is thus the most effective way of determining proper erasure time and monitoring the useful life of the ultraviolet light source. The Spectroline® DM-254X Meter is recommended for **measuring UV intensity**, while the DS-254E EPROMETER™ not only **measures UV intensity** but also **calculates the time needed for complete erasure**. Please contact the factory for complete information.

DOES THE EPROM WEAR OUT?

It has been noticed that repeated erasing and reprogramming of EPROM's appears to increase the required erasing time. Not much has been published regarding this phenomenon because EPROM's are usually not cycled that large a number of times—it can only be speculated that possibly some chemical changes take place within the device. The one thing that is certain is that it is dependent upon the erasing and programming techniques. Well over 100 programming and erase cycles still resulting in a usable EPROM have been reported in literature (1) using certain programming techniques and equipment, yet some reports have indicated only 30 to 50 times using other techniques.

Attempts have been made to "recondition" the devices by baking them prior to erasing—this technique has resulted in some reduction in the erase time. However, no method has gained widespread acceptance or recommendation because of its very limited need.

CAUTION

All Spectroline® EPROM erasing lamps and cabinets are provided with a safety interlock mechanism to protect against accidental exposure of eyes or skin to hazardous short wave ultraviolet light. Operation of the eraser is prevented unless the tray or drawer is fully and properly inserted into the housing. Any attempt to defeat the safety interlock may result in painful eye and/or skin burns or other harmful effects.

COPYRIGHT NOTICE

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.