A **COMPUTE! Books** Publication

$14.95

MACHINE LANGUAGE FOR BEGINNERS

Personal Computer Machine Language Programming For The Atari®, VIC™, Apple®, Commodore 64™, And PET/CBM™ Computers

By Richard Mansfield

MACHINE LANGUAGE FOR BEGINNERS

Machine Language Programming For BASIC Language Programmers

Richard Mansfield

COMPUTE! Publications, Inc. abc
A Subsidiary Of American Broadcasting Companies, Inc
Greensboro, North Carolina

"A few entry points, original/upgrade ROM" and "Plotting With the CBM 2022 Printer" were originally published in *COMPUTE!* Magazine, January/February 1980, copyright 1980, Small System Services, Inc. "BASIC 4.0 Memory Map" and "PET 4.0 ROM Routines" were originally published in *COMPUTE!* Magazine, November/December 1980, copyright 1980, Small System Services, Inc. "More VIC Maps" was originally published in *COMPUTE!* Magazine, March 1982, copyright 1982, Small System Services, Inc. "Commodore 64 Memory Map" was originally published in *COMPUTE!* Magazine, October 1982, copyright 1982, Small System Services, Inc. "Shoot" was originally published in *COMPUTE!* Magazine, September 1981, copyright 1981, Small System Services, Inc. "SUPERMON: A Primary Tool For Machine Language Programming" was originally published in *COMPUTE!* Magazine, December 1981, copyright 1981, Small System Services, Inc. "MICROMON: An Enhanced Machine Language Monitor" was originally published in *COMPUTE!* Magazine, January 1982, copyright 1982, Small System Services, Inc. "VIC Micromon" was originally published in *COMPUTE!* Magazine, November 1982, copyright 1982, Small System Services, Inc. "Supermon 64" was originally published in *COMPUTE!* Magazine, January 1983, copyright 1983, Small System Services, Inc.

Printed in the United States of America

ISBN 0-942386-11-6

10 9 8 7 6 5 4

Table of Contents

Preface

Something amazing lies beneath BASIC.

Several years ago I decided to learn to program in machine language, the computer's own language. I understood BASIC fairly well and I realized that it was simply not possible to accomplish all that I wanted to do with my computer using BASIC alone. BASIC is sometimes just too slow.

I faced the daunting (and exhilarating) prospect of learning to go below the surface of my computer, of finding out how to talk directly to a computer in *its* language, not the imitation-English of BASIC. I bought four books on 6502 machine language programming and spent several months practicing with them and puzzling out opcodes and hexadecimal arithmetic, and putting together small machine language programs.

Few events in learning to use a personal computer have had more impact on me than the moment that I could instantly fill the TV screen with any picture I wanted because of a machine language program I had written. I was amazed at its speed, but more than that, I realized that any time large amounts of information were needed on screen in the future — it could be done via machine language. I had, in effect, created a new BASIC "command" which could be added to any of my programs. This command — using a SYS or USR instruction to send the computer to my custom-designed machine language routine — allowed me to have previously impossible control over the computer.

BASIC might be compared to a reliable, comfortable car. It will get you where you want to go. Machine language is like a sleek racing car — you get there with lots of time to spare. When programming involves large amounts of data, music, graphics, or games — speed can become the single most important factor.

After becoming accustomed to machine language, I decided to write an arcade game entirely without benefit of

BASIC. It was to be in machine language from start to finish. I predicted that it would take about twenty to thirty hours. It was a space invaders game with mother ships, rows of aliens, sound . . . the works. It took closer to 80 hours, but I am probably more proud of that program than of any other I've written.

After I'd finished it, I realized that the next games would be easier and could be programmed more quickly. The modules handling scoring, sound, screen framing, delay, and player/enemy shapes were all written. I only had to write new sound effects, change details about the scoring, create new shapes. The essential routines were, for the most part, already written for a variety of new arcade-type games. When creating machine language programs you build up a collection of reusable subroutines. For example, once you find out how to make sounds on your machine, you change the details, but not the underlying procedures, for any new songs.

The great majority of books about machine language assume a considerable familiarity with both the details of microprocessor chips and with programming technique. This book only assumes a working knowledge of BASIC. It was designed to speak directly to the amateur programmer, the part-time computerist. It should help you make the transition from BASIC to machine language with relative ease.

This book is dedicated to Florence, Jim, and Larry. I would also like to express my gratitude to Lou Cargile for his many helpful suggestions; to Tom R. Halfhill and Charles Brannon of the *COMPUTE!* Magazine editorial staff for their contributions — both direct and indirect — to this book; and to Robert Lock and Kathleen Martinek for their encouragement, comments, and moral support. And special thanks to Jim Butterfield for his maps, programs, and constant encouragement to everyone who decides to learn 6502 machine language programming.

Introduction

Why Machine Language?

Sooner or later, many programmers find that they want to learn machine language. BASIC is a fine general-purpose tool, but it has its limitations. Machine language (often called *assembly language)* performs much faster. BASIC is fairly easy to learn, but most beginners do not realize that machine language can also be easy. And, just as learning Italian goes faster if you already know Spanish, if a programmer already knows BASIC, much of this knowledge will make learning machine language easier. There are many similarities.

This book is designed to teach machine language to those who have a working knowledge of BASIC. For example, Chapter 9 is a list of BASIC statements. Following each is a machine language routine which accomplishes the same task. In this way, if you know what you want to do in BASIC, you can find out how to do it in machine language.

To make it easier to write programs in machine language (called "ML" from here on), most programmers use a special program called an *assembler*. This is where the term "assembly language" comes from. ML and assembly language programs are both essentially the same thing. Using an assembler to create ML programs is far easier than being forced to look up and then POKE each byte into RAM memory. That's the way it used to be done, when there was too little memory in computers to hold *languages* (like BASIC or Assemblers) at the same time as *programs* created by those languages. That old style hand-programming was very laborious.

There is an assembler (in BASIC) at the end of this book which will work on most computers which use Microsoft BASIC, including the Apple, PET/CBM, VIC, and the Commodore 64. There is also a separate version for the Atari. It will let you type in ML instructions (like INC 2) and will translate them into the right numbers and POKE them for you wherever in memory you decide you want your ML program. *Instructions* are like BASIC commands; you build an ML program using the ML "instruction set." A complete table of all the 6502 ML instructions can be found in Appendix A.

It's a little premature, but if you're curious, INC 2 will increase the number in your computer's second memory cell by one. If the number in cell 2 is 15, it will become a 16 after INC 2. Think of it as "increment address two."

Throughout the book we'll be learning how to handle a variety of ML instructions, and the "Simple Assembler" program will be of great help. You might want to familiarize yourself with it. Knowing what it does (and using it to try the examples in this book), you will gradually build your understanding of ML, hexadecimal numbers, and the new possibilities open to the computerist who knows ML.

Seeing It Work

Chapters 2 through 8 each examine a major aspect of ML where it differs from the way BASIC works. In each chapter, examples and exercises lead the programmer to a greater understanding of the methods of ML programming. By the end of the book, you should be able to write, in ML, most of the programs and subroutines you will want or need.

Let's examine some advantages of ML, starting with the main one — ML runs extremely fast.

Here are two programs which accomplish the same thing. The first is in ML, and the second is in BASIC. They get results at very different speeds indeed, as you'll see:

Machine Language

```
   169 1 160 0 153 0 128 153 0 129 153 0
130 153 0 131 200 208 241 96
```

BASIC

```
5 FOR I=1 TO 1000: PRINT "A";: NEXT I
```

These two programs both print the letter "A" 1000 times on the screen. The ML version takes up 28 bytes of Random Access Memory (RAM). The BASIC version takes up 45 bytes and takes about 30 times as long to finish the job. If you want to see how quickly the ML works, you can POKE those numbers somewhere into RAM and run the ML program with a SYS (Commodore computers) or USR (Atari) or CALL (Apple). In both BASIC and ML, many instructions are followed by an *argument.* The instructions SYS and CALL have numbers as their arguments. In these cases, the instruction is going to turn control of the computer over to the address given as the argument. There would be an ML program waiting there. To make it easy to see this ML program's speed, we'll load it into memory without yet knowing much about it.

A *disassembly* is like a BASIC program's LISTing. You can give the starting address of an ML program to a *disassembler* and it will translate the numbers in the computer's memory into a readable series of ML instructions. See Appendix D for a disassembler that you can use to examine and study ML programs.

Here's what the PET/CBM version looks like when it has been translated by a disassembler:

A Disassembly

Program I-1. Disassembly.

```
.,  0360  A9  01       LDA  #$01
.,  0362  A0  00       LDY  #$00
.,  0364  99  00  80   STA  $8000,Y
.,  0367  99  00  81   STA  $8100,Y
.,  036A  99  00  82   STA  $8200,Y
.,  036D  99  00  83   STA  $8300,Y
.,  0370  C8           INY
.,  0371  D0  F1       BNE  $0364
.,  0373  60           RTS
.
```

The following BASIC programs (called *loaders*) POKE the ML instructions (and their arguments) into memory for you:

Program I-2. PET Version.

```
1 REM PET VERSION
800 FOR AD=864TO883:READ DA:POKE AD
    ,DA:NEXT AD
810 PRINT"SYS 864 TO ACTIVATE"
820 DATA169,01,160,0,153,0
830 DATA128,153,0,129,153,0
840 DATA130,153,0,131,200,208
850 DATA241,96
```

Program I-3. VIC Version.

```
1 REM VIC VERSION
800 FOR AD=864TO885:READDA:POKEAD,D
    A:NEXTAD
805 PRINT"SYS 864 TO ACTIVATE"
810 DATA 160, 0, 169, 1, 153, 0
820 DATA 30, 153, 0, 31, 169, 6
830 DATA 153, 0, 150, 153, 0, 151
840 DATA 200, 208, 237, 96
```

Program I-4. 64 Version.

Newer model 64's need to have the color registers set before running this program to see the effect on the full screen.

```
1 REM  COMMODORE 64 VERSION
800 FOR AD=40000TO40019:READDA:POKE
    AD,DA:NEXTAD
805 PRINT"SYS 40000 TO ACTIVATE"
810 DATA169,1,160,0,153,0
820 DATA4,153,0,5,153,0
830 DATA6,153,0,7,200,208
840 DATA241,96
```

Program I-5. Apple Version.

```
100 FOR I = 770 TO 789: READ A: POKE I,A: NE
    XT
110 PRINT "CALL 770 TO ACTIVATE "
120 DATA 169,129,162,0,157,0,4,157,0,5,157,0
    ,6,157,0,7,202,208,241,96
```

Program I-6. Atari Version.

```
100 FOR I=1536 TO 1561:READ A:POKE I,A:NEXT I
110 PRINT "A=USR(1536) TO ACTIVATE "
120 DATA 165,88,133,0,165,89,133,1,169
130 DATA 33,162,4,160,0,145,0,200,208,251,230
140 DATA 1,202,208,244,104,96
```

After running this program, type the SYS or USR or CALL as instructed and the screen will instantly fill. From now on, when we mention SYS, Atari owners should mentally substitute USR and Apple owners should think CALL.

BASIC stands for Beginners All-purpose Symbolic Instruction Code. Because it is all-purpose, it cannot be the perfect code for any specific job. The fact that ML speaks directly to the machine, in the machine's language, makes it the more efficient language. This is because however cleverly a BASIC program is written, it will require extra running time to finish a job.

For example, PRINT involves BASIC in a series of operations which ML avoids. BASIC must ask and answer a series of questions. Where is the text located that is to be PRINTed? Is it a variable? Where is the variable located? How long is it? Then, it must find the proper location on the screen to place the text. However, as we will discover, ML does not need to hunt for a string variable. And the screen addresses do not require a complicated series of searches in an ML program. Each of these tasks, and others, slow BASIC down because it must serve so many general purposes. The screen fills slowly because BASIC has to make many more decisions about every action it attempts than does ML.

Inserting ML For Speed

A second benefit which you derive from learning ML is that your understanding of computing will be much greater. On the abstract level, you will be far more aware of just how computers work. On the practical level, you will be able to choose between BASIC or ML, whichever is best for the purpose at hand. This choice between two languages permits far more flexibility and allows a number of tasks to be programmed which are clumsy or even impossible in BASIC. Quite a few of your favorite BASIC programs would benefit from a small ML routine, "inserted" into BASIC with a SYS, USR, or CALL, to replace a heavily used, but slow, loop or subroutine. Large sorting tasks, smooth animation, and many arcade-type games *must* involve ML.

BASIC Vs. Machine Language

BASIC itself is made up of many ML programs stored in your computer's Read Only Memory (ROM) or sometimes loaded into RAM from disk. BASIC is a group of special words such as STOP or RUN, each of which stands for a cluster of ML instructions. One such cluster might sit in ROM (unchanging memory) just waiting for you to type LIST. If you do type in that word, the computer turns control over to the ML routine which accomplishes a program listing. The BASIC programmer understands and uses these BASIC words to build a program. You hand instructions over to the computer relying on the convenience of referring to all those pre-packaged ML routines by their BASIC names. The computer, however, always follows a series of ML instructions. You cannot honestly say that you truly understand computing until you understand the computer's language: machine language.

Another reason to learn ML is that custom programming is then possible. Computers come with a disk operating system (DOS) and BASIC (or other "higher-level" languages). After a while, you will likely find that you are limited by the rules or the commands available in these languages. You will want to add to them, to customize them. An understanding of ML is necessary if you want to add new words to BASIC, to modify a word processor (which was written in ML), or to personalize your computer — to make it behave precisely as you want it to.

BASIC's Strong Points

Of course, BASIC has its advantages and, in many cases, is to be preferred over ML. BASIC is easier to analyze, particularly because it often includes REM statements which reveal the functions of the program's parts. REMs also make BASIC easier to modify. This could make it the language of choice if the program must frequently be partially rewritten or updated to conform to changing conditions. For example, a program which calculates a payroll might well have at the beginning a series of data statements which contain the tax rates. BASIC DATA statements can be easily altered so that the program will reflect the current rates. If the payroll program runs fast enough in BASIC, there is no advantage to translating it into ML.

BASIC is also simpler to *debug* (to get all the problems ironed out so that it works as it should). In Chapter 3 we will examine some ML debugging techniques which work quite well, but BASIC is the easier of the two languages to correct. For one thing, BASIC often just comes out and tells you your programming mistakes by printing out error messages on the screen.

Contrary to popular opinion, ML is not necessarily a memory-saving process. ML can use up about as much memory as BASIC does when accomplishing the same task. Short programs can be somewhat more compact in ML, but longer programs generally use up bytes fast in both languages. However, worrying about using up computer memory is quickly becoming less and less important. In a few years, programmers will probably have more memory space available than they will ever need. In any event, a talent for conserving bytes, like skill at trapping wild game, will likely become a victim of technology. It will always be a skill, but it seems as if it will not be an everyday necessity.

So, which language is best? They are both best — but for different purposes. Many programmers, after learning ML, find that they continue to construct programs in BASIC, and then add ML modules where speed is important. But perhaps the best reason of all for learning ML is that it is fascinating and fun.

1

How To Use This Book

Although anyone wishing to learn 6502 machine language (ML) will likely find this book instructive and worthwhile, the specific example programs are written to work on five popular personal computers: Apple, Atari, VIC, Commodore 64, and the PET/CBMs. If your computer uses the 6502 microprocessor, but is not one of these machines, you will need to find a ''memory map'' for your particular machine. These maps — widely available in books and magazines, and from user groups — will allow you to follow and practice with the examples of 6502 machine language throughout this book.

In particular, there are several memory addresses which are used in many of the examples presented in this book. Their addresses are given for the five computers mentioned above, but if you have a different computer, you should look them up in a map of your machine:

1. *''Which key is pressed?''* This is an address, usually somewhere in the first 256 addresses, which is always holding the value of the most recently pressed key on the keyboard.

2. *Starting Address of RAM Screen Memory.* This is the address in your computer where, if you POKEd something into it from BASIC, you would see the effect in the upper left-hand corner of your screen.

3. *Print a Character.* This address is within your BASIC ROM memory itself. It is part of the BASIC language, but written in ML. It is the starting address of a routine which will put a character on the screen.

4. *Get a Character.* Also part of BASIC in ROM memory, this ML routine accepts a character from the keyboard and stores it.

5. *A safe place.* You must know where, in your computer, you can construct ML programs without interfering with a BASIC program or anything else essential to the computer's normal operations. The best bet is often that memory space designed to serve the cassette player called the *cassette buffer.* While practicing, you won't be using the cassette player and that space will be left alone by the computer itself.

Here are the answers to give the Simple Assembler (Appendix C) when it asks for ''Starting Address.'' These are hexadecimal numbers about which we'll have more to say in the next chapter. For now, if you've got an Atari, type in 0600. If you use a PET/CBM, answer 0360. For VIC or Commodore 64, type: 0340. If you have an

Apple, use 0300. For other computers, you'll need to know where there are about 100 RAM memory addresses that are safe.

All through this book, the examples will start at various arbitrary addresses (1000, 2000, 5000, for example). You should substitute the addresses which are safe in your computer. Just as it doesn't matter whether you start a BASIC program at line number 10 or line 100, it makes no difference whether a ML program starts at address 1000 or 0340, as long as you are putting it in a safe memory zone.

So, start all of the examples you assemble for practice in the same convenient, safe memory location for your machine. In fact, the Simple Assembler (SA) was designed to be modified and customized. See the introduction to Appendix C for more detailed instructions on customizing. Because you can make the SA conform to your needs, you might want to replace the line with the INPUT that requests the starting address (variable SA) with a specific address. In this way, you can work with the examples in the book without having to specify the safe address each time.

The First Step: Assembling

Throughout this book there are many short example ML programs. They vary in length, but most are quite brief and are intended to illustrate a ML concept or technique. The best way to learn something new is most often to just jump in and do it. Machine language programming is no different. Machine language programs are written using a program called an *assembler,* just as BASIC programs are written using a program called "BASIC."

In Appendix C there is a program called the "Simple Assembler." Your first step in using this book should be to type in the Microsoft version; it will work correctly on all personal computers using Microsoft BASIC. (If you have an Atari, type in the Atari version.)

Once you've typed this program into your computer, you can save it to tape or disk and use it whenever you want to construct a ML program. The example ML routines in this book should be entered into your computer using the Simple Assembler and then modified, examined, and played with.

Frequently, the examples are designed to do something to the screen. The reason for this is that you can tell at once if things are working as planned. If you are trying to send the message "TEST STRING" and it comes out "test string" or "TEST STRIN" or "TEST STRING@" — you can go back and reassemble it with the SA until you get it right. More importantly, you'll discover what you did wrong.

What you see on the screen when you POKE a particular number to the screen will differ from computer to computer. In fact, it can vary on different models of the same computer. For this reason, the examples in the book are usually given in standard ASCII codes (explained later).

Chances are that your computer uses a particular code for the alphabet which is not ASCII. The Commodores use what's called ''PET ASCII'' and the Atari uses ATASCII, for *ATari ASCII*. It's not that bad, however, since once you've found the correct number to show the letter ''A'' on screen, the letter ''B'' will be the next higher number. If you don't have a chart of the character codes for your computer's screen POKEs, just use this BASIC program and jot down the number which is used to POKE the uppercase and lowercase ''A.''

10 FOR I = 0 TO 255: POKE (your computer's start-of-screen-RAM address), **I: NEXT**

With that knowledge, you can easily achieve the exact, predicted results for the examples in the book by substituting your computer's code.

A Sample Example

The following illustrations will show you how to go about entering and testing the practice examples in the book. At this point, of course, you won't recognize the ML instructions involved. The following samples are only intended to serve as a guide to working with the examples you will come upon later in the text.

After you've typed in and saved the SA, you can RUN it (it's a BASIC program which helps you to write ML). The first thing it does is ask you where you want to start your ML program — where you want it stored in memory. This is why you need to know of a safe place to put ML programs in your computer.

Of course you use line numbers when creating a BASIC program. Line numbers are not used in ML programming. Instead, you can think of memory addresses as ''line numbers.'' So, if you are using the Atari, you will tell the SA that you are going to start your ML program at 0600. It will then print 0600 on the screen as if it were a line number, and you enter a ML program instruction, one per line, like this:

0600	**PLA**		(This PLA is always required in the Atari when you use USR.)
0601	**LDY**	#00	(Stay in the hexadecimal mode for this example.)
0603	**LDA**	#21	
0605	**STA**	(58)Y	
0608	**RTS**		
0609	**END**		

The SA will automatically print each "line number" address when you are programming. You just type in those now mysterious ML instructions. This program will put the letter "A" on screen. After you are finished with an example, you type the word "END" and the SA will tell you the starting address of your ML program in RAM memory.

The next step is to try out the ML program you've written to see that it will work as planned. On the Atari, you could type:

X=USR(1536) (and hit RETURN)

and this will "RUN" your ML program. You will have sent control of the computer from BASIC to your new ML program via the USR command. Be sure to remember that the Atari requires the PLA as the first instruction of each ML program that you plan to go to from BASIC by using the USR command. *In all the examples in this book, type in a PLA as the first instruction before continuing with the rest of the example if you use an Atari.*

Most personal computers use Microsoft BASIC, and the PLA is not necessary. Here's how the same example would look on a PET/CBM after you answered 0360 as the starting address when the SA asked for it:

0360	**LDY**	**#01**	
0362	**LDA**	**#41**	
0364	**STA**	**8000**	
0367	**RTS**		
0368	**END**	(The word "END" isn't a 6502 ML instruction; it's a special signal to the SA to stop constructing a program and exit the SA program. Such special words are called *pseudo-ops.)*	

Then you could test it in direct mode (just typing in the instruction onto the screen with no line number and not as part of a BASIC program) by typing:

SYS 864 and you should see the "A" on the screen.

Notice that the Atari and PET versions are similar, but not identical. All 6502 based computers will work with the same "instruction set" of commands which the 6502 chip can understand. The major differences occur when you need to specify something which is particular to the design of your computer brand. An example would be the location in memory of your computer's screen. The instructions at 0605 in the Atari example and 0364 in the PET example send the code for the letter "A" to the different screen locations for these two computer brands. Also, the letter "A" itself is signified by the number 41 on a PET and by the number 21 on an Atari.

But we'll go into these things further on. The main thing to learn here is how to use the SA to practice the examples. If you type in 0600 as the starting address as in the Atari example above, the SA will print the number 0600 on screen and wait for you to type in a 6502 instruction (PLA in this case) and hit RETURN. Then it will print the next memory address just as if you were using an automatic line numbering routine when programming in BASIC. After you hit RETURN, the SA will print 0601 and wait for you to type in LDY #00.

2

The Fundamentals

The difficulty of learning ML has sometimes been exaggerated. There are some new rules to learn and some new habits to acquire. But most ML programmers would probably agree that ML is not inherently more difficult to understand than BASIC. More of a challenge to debug in many cases, but it's not worlds beyond BASIC in complexity. In fact, many of the first home computerists in the 1970's learned ML before they learned BASIC. This is because an average version of the BASIC language used in microcomputers takes up around 12,000 bytes of memory, and early personal computers (KIM, AIM, etc.) were severely restricted by containing only a small amount of available memory. These early machines were unable to offer BASIC, so everyone programmed in ML.

Interestingly, some of these pioneers reportedly found BASIC to be just as difficult to grasp as ML. In both cases, the problem seems to be that the rules of a new language simply are "obscure" until you know them. In general, though, learning either language probably requires roughly the same amount of effort.

The first thing to learn about ML is that it reflects the construction of computers. It most often uses a number system (hexadecimal) which is not based on ten. You will find a table in Appendix E which makes it easy to look up hex, decimal, or binary numbers.

We count by tens because it is a familiar (though arbitrary) grouping for us. Humans have ten fingers. If we had eleven fingers, the odds are that we would be counting by elevens.

What's a Natural Number?

Computers count in groups of twos. It is a fact of electronics that the easiest way to store and manipulate information is by ON-OFF states. A light bulb is either on or off. This is a two-group, it's *binary,* and so the powers of two become the natural groupings for electronic counters. 2, 4, 8, 16, 32, 64, 128, 256. Finger counters (us) have been using tens so long that we have come to think of ten as *natural,* like thunder in April. Tens isn't natural at all. What's more, twos is a more efficient way to count.

To see how the powers of two relate to computers, we can run a short BASIC program which will give us some of these powers. *Powers* of a number are the number multiplied by itself. Two to the power of two (2^2) means 2 times 2 (4). Two to the power of three (2^3) means 2 times 2 times 2 (8).

```
10 FOR I=0 to 16
20 PRINT 2 ^ I
30 NEXT I
```

ML programming *can* be done in decimal (based on ten-groupings), but usually is not. Most ML programming involves *hex* numbers. This means groups of 16 rather than 10.

Why not just program in the familiar decimal numbers (as BASIC does)? Because 16 is one of the powers of two. It is a convenient grouping (or *base*) for ML because it organizes numbers the way the computer does. For example, all computers work, at the most elementary level, with *bits.* A bit is the smallest piece of information possible: something is either on or off, yes or no, plus or minus, true or false. This two-state condition (binary) can be remembered by a computer's smallest single memory cell. This single cell is called a *bit.* The computer can turn each bit "on" or "off" as if it were a light bulb or a flag raised or lowered.

It's interesting that the word *bit* is frequently explained as a shortening of the phrase BInary digiT. In fact, the word *bit* goes back several centuries. There was a coin which was soft enough to be cut with a knife into eight pieces. Hence, *pieces of eight.* A single piece of this coin was called a *bit* and, as with computer memories, it meant that you couldn't slice it any further. We still use the word *bit* today as in the phrase *two bits,* meaning 25 cents.

Whatever it's called, the bit is a small, essential aspect of computing. Imagine that we wanted to remember the result of a subtraction. When two numbers are subtracted, they are actually being compared with each other. The result of the subtraction tells us which number is the larger or if they are equal. ML has an instruction, like a command in BASIC, which compares two numbers by subtraction. It is called CMP (for *compare*). This instruction sets "flags" in the CPU (Central Processing Unit), and one of the flags always remembers whether or not the result of the most recent action taken by the computer was a zero. We'll go into this again later. What we need to realize now is that each flag — like the flag on a mailbox — has two possible conditions: up or down. In other words, this information (zero result or not-zero) is *binary* and can be stored within a single bit. Each of the flags is a bit. Together they make up one byte. That byte is called the Status Register.

Byte Assignments

Our computers group these bits into units of eight, called *bytes.* This relationship between bits and bytes is easy to remember if you think of a bit as one of the "pieces of eight." Eight is a power of two also (two, to the third power). Eight is a convenient number of bits to work with as a group because we can count from zero to 255 using only eight bits.

This gives us enough room to assign all 26 letters of the alphabet (and the uppercase letters and punctuation marks, etc.) so that each printed character will have its particular number. The letter ''A'' (uppercase) has been assigned the number 65. ''B'' is 66, and so on. Throughout this book, examples will follow the ASCII code for letters of the alphabet. Most microcomputers, however, do not adhere strictly to the ASCII code. If you get unexpected results when trying the example programs, check your BASIC manual to see if POKEing to the screen RAM uses a different code than ASCII. If that is the case, substitute your screen POKE code for the values given in the examples.

These ''assignments'' form the convention called the ASCII code by which computers worldwide can communicate with each other. Text can be sent via modems and telephone lines and arrive meaning the same thing to a different computer. It's important to visualize each byte, then, as being eight bits ganged together and able to represent 256 different things. As you might have guessed, 256 is a power of two also (two, to the power of eight).

So, these groupings of eight, these bytes, are a key aspect of computing. But we also want to simplify our counting from 0 to 255. We want the numbers to line up in a column on the screen or on paper. Obviously, the *decimal* number five takes up one space and the number 230 takes up three spaces.

Also, hex is easier to think about in terms of *binary* numbers — the on-off, single-bit way that the computer handles numbers:

Decimal		Hex	Binary
1		01	00000001
2		02	00000010
3		03	00000011 (1 and 2)
4		04	00000100
5		05	00000101 (4 and 1)
6		06	00000110 (4 and 2)
7		07	00000111 (4+2+1)
8		08	00001000
9		09	00001001
10	(note new digits)——>	0A	00001010
11		0B	00001011
12		0C	00001100
13		0D	00001101
14		0E	00001110
15		0F	00001111
16	(note new column——>	10	00010000
17	in the hex)	11	00010001

See how hex $10 (hex numbers are usually preceded by a dollar sign to show that they are not decimal) *looks like* binary? If you split a hex number into two parts, 1 and 0, and the binary (it's an eight-bit group, a *byte*) into two parts, 0001 and 0000 — you can see the relationship.

The Rationale For Hex Numbers

ML programmers often handle numbers as *hexadecimal* digits, meaning groups of sixteen instead of ten. It is usually just called *hex*. You should read over the instructions to the Simple Assembler and remember that you can choose between working in hex or decimal with that assembler. You can know right from the start if you're working with hex or decimal, so the dollar sign isn't used with the Simple Assembler.

DECIMAL 0 1 2 3 4 5 6 7 8 9 then you start over with 10

HEX 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F then you start over with 10

Program 2-1. Microsoft Hex-Decimal Converter.

```
1 HE$="Ø123456789ABCDEF"
2 PRINT"{CLEAR}{Ø3 DOWN}PLEASE CHOOSE:
4 PRINT"{Ø3 DOWN}{Ø3 RIGHT}1-INPUT HEX &
     GET DECIMAL BACK.
5 REM NEW LINE HERE
6 PRINT"{Ø2 DOWN}    2-INPUT DECIMAL TO G
    ET HEX BACK.
7 GETK:IFK=ØTHEN7
9 PRINT"{CLEAR}":ON KGOTO2ØØ,4ØØ
1ØØ H$="":FORM=3TOØSTEP-1:N%=DE/(16^M):DE=
     DE-N%*16^M:H$=H$+MID$(HE$,N%+1,1)
     :NEXT
1Ø1 RETURN
1Ø2 D=Ø:Q=3:FORM=1TO4:FORW=ØTO15:IFMID$(H$
     ,M,1)=MID$(HE$,W+1,1)THEN1Ø4
1Ø3 NEXTW
1Ø4 D1=W*(16^(Q)):D=D+D1:Q=Q-1:NEXTM
1Ø5 DE=INT(D):RETURN
2ØØ INPUT"{Ø2 DOWN}HEX";H$:GOSUB1Ø2:PRINTS
     PC(11)"{UP}= {REV}"DE"{LEFT} "
21Ø GOTO2ØØ
4ØØ INPUT"{Ø2 DOWN}DECIMAL";DE:GOSUB1ØØ:PR
```

```
    INTSPC(14)"{UP}= {REV} "H$" "
410 GOTO400
```

Program 2-2. Atari Hex-Decimal Converter.

```
100 DIM H$(23),N$(9):OPEN#1,4,0,"K:"
130 GRAPHICS 0
140 PRINT" PLEASE CHOOSE:"
150 PRINT"1- INPUT HEX AND GET DECIMAL BAC
    K."
160 PRINT"2- INPUT DECIMAL AND GET HEX BAC
    K."
170 PRINT:PRINT"==>";:GET#1,K
180 IFK<49OR>50THEN170
190 PRINTCHR$(K):ONK-48 GOTO 300,400
300 H$="@ABCDEFGHI!!!!!!!JKLMNO"
310 PRINT"HEX";:INPUT N$:N=0
320 FORI=1TOLEN(N$)
330 N=N*16+ASC(H$(ASC(N$(I))-47))-64:NEXTI

340 PRINT"$";N$;"=";N:PRINT:PRINT:GOTO140
400 H$="0123456789ABCDEF"
410 PRINT"DECIMAL";:INPUTN:M=4096
420 PRINTN;"=$";
430 FORI=1TO4:J=INT(N/M)
440 PRINTH$(J+1,J+1);:N=N-M*J:M=M/16
450 NEXTI:PRINT:PRINT:GOTO140
```

The first thing to notice is that instead of the familiar decimal symbol 10, hex uses the letter ''A'' because this is where decimal numbers run out of symbols and start over again with a one and a zero. Zero always reappears at the start of each new grouping in any number system: 0, 10, 20, etc. The same thing happens with the groupings in hex: 0, 10, 20, 30, etc. The difference is that, in hex, the 1 in the ''tens'' column equals a decimal 16. *The second column is now a ''sixteens'' column.* 11 means 17, and 21 means 33 (2 times 16 plus one). Learning hex is probably the single biggest hurdle to get over when getting to know ML. Don't be discouraged if it's not immediately clear what's going on. (It probably never will be totally clear — it is, after all, unnatural.) You might want to practice the exercises at the end of this chapter. As you work with ML, hex will gradually seem less and less alien.

To figure out a hex number, multiply the second column by 16 and add the other number to it. So, 1A would be one times 16 plus 10 (recall that A stands for ten).

Hex does seem impossibly confusing when you come upon it for the first time. It will never become second nature, but it should be at least generally understood. What is more, you can program in ML quite easily by looking up the hex numbers in the table at the end of this book. You need not memorize them beyond learning to count from 1 to 16 — learning the symbols. Be able to count from 00 up to 0F. (By convention, even the smallest hex number is listed as two digits as in 03 or 0B. The other distinguishing characteristic is that dollar sign that is usually placed in front of them: $05 or $0E.) It is enough to know what they look like and be able to find them when you need them.

The First 255

Also, most ML programming involves working with hex numbers only between 0 and 255. This is because a single byte (eight bits) can hold no number larger than 255. Manipulating numbers larger than 255 is of no real importance in ML programming until you are ready to work with more advanced ML programs. This comes later in the book. For example, all 6502 ML instructions are coded into one byte, all the "flags" are held in one byte, and many "addressing modes" use one byte to hold their argument.

To learn all we need know about hex for now, we can try some problems and look at some ML code to see how hex is used in the majority of ML work. But first, let's take an imaginary flight over computer memory. Let's get a visual sense of what bits and bytes and the inner workings of the computer's RAM look like.

The City Of Bytes

Imagine a city with a single long row of houses. It's night. Each house has a peculiar Christmas display: on the roof is a line of eight lights. The houses represent bytes; each light is a single bit. (See Figure 2-1.) If we fly over the city of bytes, at first we see only darkness. Each byte contains nothing (zero), so all eight of its bulbs are off. (On the horizon we can see a glow, however, because the computer has memory up there, called ROM memory, which is very active and contains built-in programs.) But we are down in RAM, our free user-memory, and there are no programs now in RAM, so every house is dark. Let's observe what happens to an individual byte when different numbers are stored there; we can randomly choose byte 1504. We hover over that house to see what information is "contained" in the light display. (See Figure 2-2.)

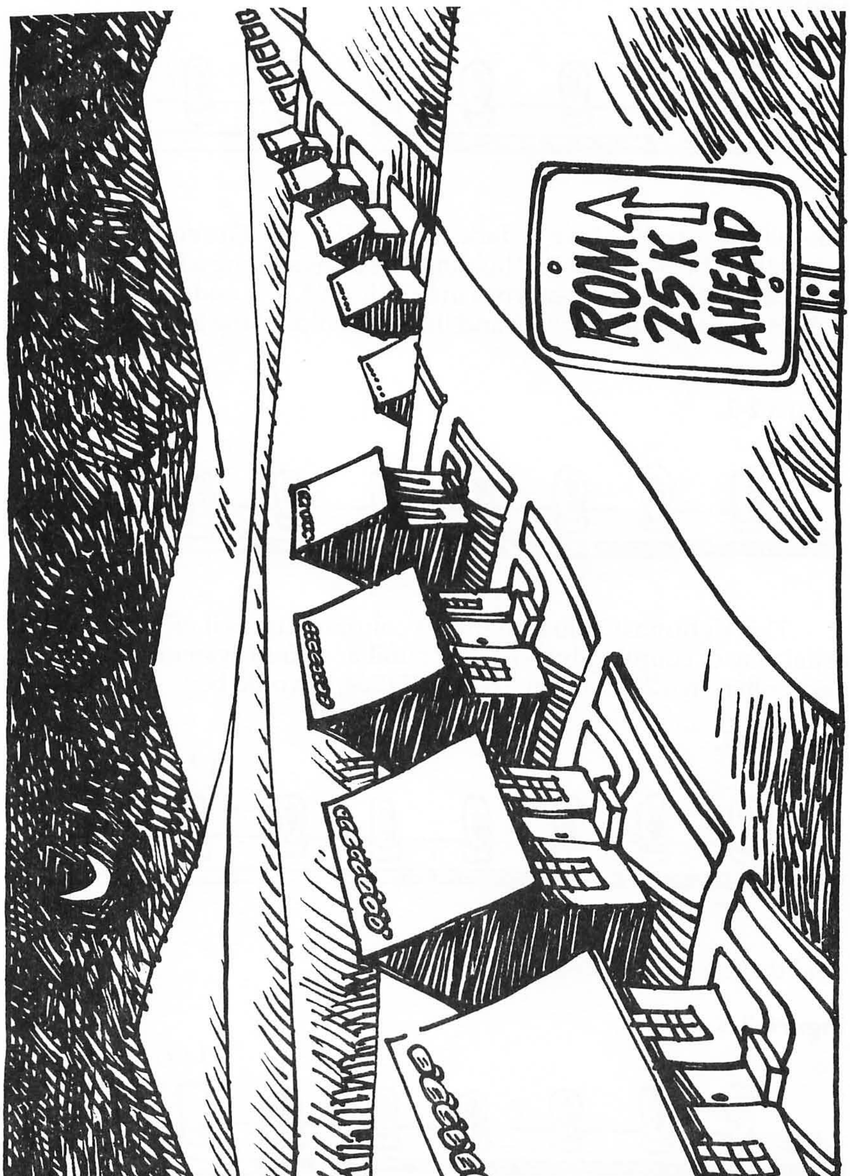
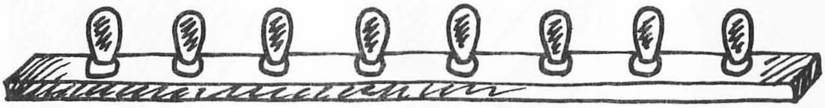


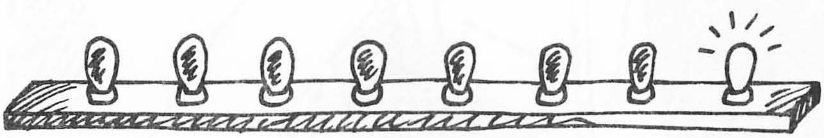
Figure 2-1. Night In The City Of Bytes.

Figure 2-2.



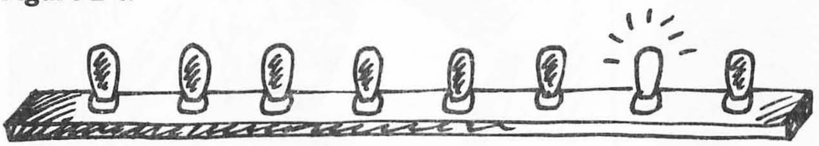
Like all the rest, this byte is dark. Each bulb is off. Observing this, we know that the byte here is "holding" or representing a zero. If someone at the computer types in POKE 1504, 1 — suddenly the rightmost light bulb goes on and the byte holds a one instead of a zero:

Figure 2-3.



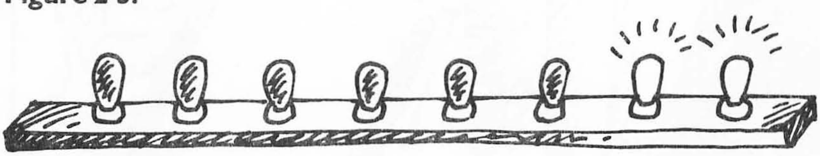
This rightmost bulb is in the 1's column (just as it would be in our usual way of counting by tens, our familiar *decimal* system). But the next bulb is in a 2's column, so POKE 1504, 2 would be:

Figure 2-4.



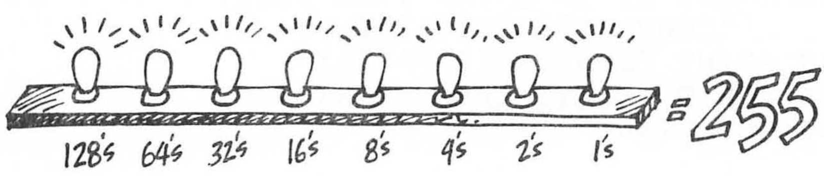
And three would be one and two:

Figure 2-5.



In this way — by checking which bits are turned on and then adding them together — the computer can look at a byte and know what number is there. Each light bulb, each *bit,* is in its own special position in the row of eight and has a value twice the value of the one just before it:

Figure 2-6.



Eight bits together make a byte. A byte can ''hold'' a number from 0 through 255 decimal. We can think of bytes, though, in any number system we wish — in hex, decimal, or binary. The computer uses binary, so it's useful to be able to visualize it. Hex has its uses in ML programming. And decimal is familiar. But a number is still a number, no matter what we call it. After all, five trees are going to be five trees whether you symbolize them by 5, $05, or 00000101.

A Binary Quiz

BASIC doesn't understand numbers expressed in hex or binary. The Simple Assembler contains two subroutines to translate a number from decimal to hex or vice versa. You might want to take a look at how it's done as a way of getting a better feel for these different numbers systems. The subroutines are located at lines 4000 and 5000. Binary, for humans, is very *visual.* It forms patterns out of zeros and ones. The following program will let you quiz yourself on these patterns.

Here is a game, for all computers, which will show you a byte as it looks in binary. You then try to give the number in decimal:

Program 2-3. Binary Quiz for All Computers.

```
100 REM BINARY QUIZ
110 C1=20:C0=111: REM FOR ATARI ONLY
120 C1=88:C0=79:  REM FOR APPLE ONLY
130 C1=209:C0=215:REM FOR COMMODORE ONLY
140 X=INT(256*RND(1)): D = X: P = 128
150 PRINT CHR$(125);: REM ATARI ONLY
160 PRINT CHR$(147);: REM COMMODORE ONLY
170 HOME: REM APPLE ONLY
180 FOR I = 1 TO 8
190 IF INT(D/P) = 1 THEN PRINT CHR$(C1);:
      D = D-P: GOTO 210
```

```
200 PRINT CHR$(C0);
210 P = P/2: NEXT I: PRINT
220 PRINT " WHAT IS THIS IN DECIMAL?"
230 INPUT Q: IF Q = X THEN PRINT
      "CORRECT": GOTO 250
240 PRINT "SORRY, IT WAS";X
250 FOR T = 1 TO 1000: NEXT T
260 GOTO 140
```

This program will print out the entire table of binary numbers from 0 to 255:

Program 2-4.

```
100 REM COMPLETE BINARY TABLE
110 L=8:B=2:C=1
120 FORX=0TO255:PRINTX;
140 IFXAND1THENK(C)=49:GOTO160
150 K(C)=48
160 C=C+1:IFBANDXTHENK(C)=49:GOTO180
170 K(C)=48
180 B=B*2:IFC>8THEN200
190 GOTO160
200 FORI=0TO7:PRINTSTR$(K(L)-48);:L=L-1
210 NEXT
220 C=0:PRINT
260 L=8:B=2:C=1:NEXTX
```

Examples And Practice

Here are several ordinary decimal numbers. Try to work out the hex equivalent:

1. 10 ________
2. 15 ________
3. 5 ________
4. 16 ________
5. 17 ________
6. 32 ________
7. 128 ________
8. 129 ________

9. 255 ________
10. 254 ________

We are not making an issue of learning hex or binary. If you needed to look up the answers in the table at the end of the book, fine. As you work with ML, you will familiarize yourself with some of the common hex numbers. You can write most ML programs without needing to worry about binary. For now, we only want to be able to recognize what hex is. There are even some pocket "programmer" calculators which change decimal to hex for you and vice versa. Another way to go about "hexing" is to use a BASIC program which does the translation. A problem with BASIC is that you will be working in ML and your computer will be tied up. It is often inconvenient to crank up a BASIC program each time you need to work out a hex number. However, the Simple Assembler will do the translations for you any time you need them.

One other reason that we are not stressing hex too much is that ML is generally not programmed without the help of an assembler. The Simple Assembler provided in this book will handle most of your input automatically. It allows you to choose whether you prefer to program in hex or decimal. You make this decision by changing line 10 before starting to assemble. After that, you can put in hex or decimal without worrying that there will be any confusion about your intentions.

This little BASIC program is good for practicing hex, but also shows how to change a small part and make it work for two-byte hex numbers. It will take decimal in and give back the correct hex. It is designed for Microsoft BASIC computers, so it will not work on the Atari.

```
10 H$="0123456789ABCDEF"
20 PRINT "ENTER DECIMAL NUMBER";:INPUT X
30 IF X > 255 GOTO 20: REM NO NUMBERS BIGGER
THAN 255 ALLOWED
40 FOR I=1 TO 0 STEP-1
50 N%=X/(16↑I): X=X-N% * 16↑I
60 HE$=HE$+MID$(H$,N%+1,1)
70 NEXT
80 PRINT HE$
90 GOTO 20
```

For larger hex numbers (up to two, to the power of 16 — which is 65536), we can just change the above program. Eliminate line 30 and change line 40 to: FOR I=3 TO 0 STEP -1. This will give us four-place hex numbers (used only as addresses) but which will also become recognizable after some ML practice.

65535 is an interesting number because it represents the limit of our computers' memories. In special cases, with additional hardware, memory *can* be expanded beyond this. But this is the normal upper limit because the 6502 chip is designed to be able to *address* (put bytes in or take them out of memory cells) up to $FFFF.

Ganging Two Bytes Together To Form An Address

The 6502 often sets up an address by attaching two bytes together and looking at them as if they formed a unit. An address is most commonly a two-byte number. $FFFF (65535) is the largest number that two bytes can represent, and $FF (255) is the most that *one* byte can hold. Three-byte addressing is not possible for the 6502 chip. "Machine language" means programming which is understood directly by the 6502 chip itself. There are other CPU (Central Processing Unit) chips, but the 6502 is the CPU for VIC, Apple, 64, PET/CBM, and Atari. It's the one covered in this book.

Reading A Machine Language Program

Before getting into an in-depth look at "monitors," those bridges between you and your machine's language — we should first learn how to read ML program listings. You've probably seen them often enough in magazines. Usually, these commented, labeled, but very strange-looking programs are called *source code.* They can be examined and translated by an *assembler program* into an ML program. When you have an assembler program run through source code, it looks at the key words and numbers and then POKEs a series of numbers into the computer. This series is then called the *object code.*

Source programs contain a great deal of information which is of interest to the programmer. The computer only needs a list of numbers which it can execute in order. But for most people, lists of numbers are only slightly more understandable than Morse code. The solution is to replace numbers with words. The primary job of an assembler is to recognize an ML instruction. These instructions are called *mnemonics,* which means "memory aids." They are like BASIC words, except that they are always three letters long.

If you type the mnemonic JMP, the assembler POKEs a 76 into RAM memory. It's easier to remember JMP than 76. The 76 is the number that clues the computer that it's supposed to perform a JMP. The 76 is called an *opcode,* for "operation code." The three-letter words we use in ML programming, the mnemonics, were designed to sound like what they do. JMP does a JUMP (like a GOTO in BASIC). Some deluxe assemblers also let you use labels instead of numbers — as long as you define your labels at the start of the source code. These labels can refer to individual memory locations, special values like the score in a game, or entire subroutines.

Four Ways To List A Program

Labeled, commented source code listings are the most elaborate kind of ML program representation. There are also three other kinds of ML listings. We can use a simple addition example program to show how it looks when represented in each of the four ML program listing styles. The first two styles are simply ways for you to type a program into the computer. The last two styles show you what to type in, but also illustrate what is going on in the ML program. First let's look at the most elementary kind of ML found in books and magazines: the BASIC loader.

Program 2-6. BASIC Loader.

```
10 FOR ADDRESS = 4096 TO 4103
20 READ BYTE
30 POKE ADDRESS, BYTE
40 NEXT ADDRESS
50 DATA 169,2,105,5,141,160,15,96
```

This is a series of decimal numbers in DATA statements which is POKEd into memory starting at decimal address 4096. When these numbers arrive in RAM, they form a little routine which puts the number 2 into the *accumulator* — a special location in the computer that we'll get to later — and then adds 5. The result of the addition is then moved from the accumulator to decimal address 4000. If you try this program out, you can SYS 4096 to execute ML program and then ? PEEK (4000) and you'll see the answer: seven. BASIC loaders are convenient because the user doesn't need to know how to enter ML programs. The loader POKEs them in and all the user has to do is SYS or USR or CALL to the right address and the ML transfers control back to BASIC when its job is done.

Getting even closer to the machine level is the second way you might see ML printed in books or magazines: the hex dump. On some computers (PET, Apple) there is a special "monitor" program in ROM which lets you list memory addresses and their contents as hex numbers. More than that, you can usually type over the existing values on the screen and change them. That's what a hex dump listing is for. You copy it into your computer's RAM by using your computer's monitor. How you enter the monitor mode differs on each computer and we'll get to monitors in the next chapter.

The hex dump, like the BASIC loader, tells you nothing about the functions or strategies employed within an ML program. Here's the hex dump version of the same 2+5 addition program:

Program 2-7.

```
1000 A9 02 69 05 8D A0 0F 60
```

The third type of listing is called a *disassembly*. It's the opposite of an assembly because another program called a *disassembler* takes machine language (the series of numbers, the opcodes in the computer's memory) and translates it into the words, the mnemonics, which ML programmers use. The instruction you use when you want to load the accumulator is called LDA, and you can store what's in the accumulator by using an STA. We'll get to them later. In this version of our example addition routine, it's a bit clearer what's going on and how the program works. Notice that on the left we have the hex numbers and, on the right, the translation into ML instructions. ADC means ADd with Carry and RTS means ReTurn from Subroutine.

Program 2-8.

```
1000 A9 02       LDA #$02
1002 69 05       ADC #$05
1004 8D A0 0F    STA $0FA0
1007 60          RTS
```

The Deluxe Version

Finally we come to that full, luxurious, commented, labeled, deluxe source code we spoke of earlier. It includes the hex dump and the disassembly, but it also has labels and comments and line numbers added, to further clarify the purposes of things. Note that the numbers are all in hex. On the far left are the memory addresses where this routine is located. Next to them are the hex numbers of the instructions. (So far, it resembles the traditional hex dump.) Then come line numbers which can be used the way BASIC line numbers are: deleted, inserted, and so on. Next are the disassembled translations of the hex, but you can replace numbers with labels (see Program 2-10). You could still use numbers, but if you've defined the labels early on, they can serve as a useful reminder of what the numbers represent. Last, following the semicolons, are the comments. They are the same as REM statements. (See Programs 2-9 and 2-10.)

Program 2-9. A Full Assembly Listing.

```
                                      ------------- Source Code -------------
Memory   Object      Line       Disassembly                Comments
Address  Code        Number

                     0005       .BA $1000           ; START ADDR $1000 (4096)
1000-  A9 02         0100       LDA #$02            ; LOAD A WITH 2
1002-  69 05         0110       ADC #$05            ; ADD 5
1004-  8D A0 0F      0120       STA $0FA0           ; STORE AT DECIMAL 4000
1007-  60            0130       RTS                 ; RETURN
                     0140       .EN                 ; END OF ASSEMBLY
```

Program 2-10. Labelled Assembly.

```
                     0005                .BA $1000     ; START ADDR $1000 (4096)
                     0010 TWO            .DE 2         ;DEFINE LABEL "TWO" AS 2.
                     0020 ADDER          .DE 5         ;DEFINE "ADDER" AS A 5.
                     0030 STORAGE        .DE $0FA0     ;DEFINE STORAGE ADDR.
                     0040 ;
1000-  A9 02         0100 START          LDA #TWO      ; LOAD A WITH 2
1002-  69 05         0110                ADC #ADDER    ; ADD 5
1004-  8D A0 0F      0120                STA STORAGE   ; STORE AT DECIMAL 4000
1007-  60            0130                RTS           ; RETURN
                     0140                .EN           ; END OF ASSEMBLY

---  LABEL FILE:  ---

ADDER =0005           START =1000           STORAGE =0FA0
TWO =0002
```

Program 2-11. The Source Code By Itself.

```
            .BA $1000       ; START ADDR $1000 (4096)
TWO         .DE 2           ;DEFINE LABLE "TWO" AS 2.
ADDER       .DE 5           ;DEFINE "ADDER" AS A 5.
STORAGE     .DE $0FA0       ;DEFINE STORAGE ADDR.
;
START       LDA #TWO        ; LOAD A WITH 2
            ADC #ADDER      ; ADD 5
            STA STORAGE     ; STORE AT DECIMAL 4000
            RTS             ; RETURN
            .EN             ; END OF ASSEMBLY
```

Program 2-11 illustrates just the *source code* part. The object code has not yet been generated from this source code. The code has not been *assembled* yet. You can save or load source code via an assembler in the same way that you can save or load programs via BASIC. When 2-11 is in the computer, you could type ''ASSEMBLE'' and the assembler would translate the instructions, print them on the screen, and POKE them into memory.

The Simple Assembler operates differently. It translates, prints, and POKEs after you hit RETURN on each line of code. You can save and load the object, but not the source code.

Before we get into the heart of ML programming, a study of the instruction mnemonics and the various ways of moving information around (called *addressing*), we should look at a major ML programming aid: the monitor. It deserves its own chapter.

ANSWERS to quiz: 0A, 0F, 05, 10, 11, 20, 80, 81, FF, FE

3

The Monitor

A monitor is a program which allows you to work directly with your computer's memory cells. When the computer ''falls below'' BASIC into the monitor mode, BASIC is no longer active. If you type RUN, it will not execute anything. BASIC commands are not recognized. The computer waits, as usual, for you to type in some instructions. There are only a few instructions to give to a monitor. When you're working with it, you're pretty close to talking directly to the machine in machine language.

The PET and Apple II have monitors in ROM. This means that you do not need to load the monitor program into the computer; it's always available to you. (PETs with Original ROM sets do not have a ROM monitor; you must load in the monitor from a tape or disk.) Atari and VIC computers have a monitor as part of a larger ''Assembler Editor'' plug-in cartridge. The monitor on the Atari cartridge is called the ''Debugger.'' That's a good name for it: debugging is the main purpose of a monitor. You use it to check your ML code, to find errors.

The various computers have different sets of instructions which their monitors recognize. However, the main functions are similar, so it is worth reading through all of the following descriptions, even if the discussion is not specifically about the monitor for your computer. On the PET/CBM, VIC, and 64 you can add many of these functions with a monitor ''extension'' program called *Micromon* or *Supermon* (about which more later). These monitors are included in Appendix F. The monitors on the Apple II and available in the Atari Assembler Editor Cartridge do not need ''extending.'' They contain most of the significant features required of a monitor. However, the special extensions in Appendix F for the Commodore computers add considerably to the Commodore ML programmer's repertoire.

The Apple II

You enter the Apple monitor by typing CALL -151. You will see the ''*'' monitor prompt and the cursor immediately after it. Here are the monitor instructions:

1. Typing an address (in hex) will show you the number contained in that memory cell. *2000 (hit RETURN) will show 2000 — FF (if, in fact, 255 decimal ($FF, hex) is in that location).

2. You can examine a larger amount of memory in hex (this is

called a *memory dump* or a *hex dump*). The Apple monitor remembers the address of the last number displayed. This can be used as a starting address for the dump. If you type the instruction in number one above, and then type *.2010, you will see a dump of memory between 2001 and 2010. The only difference between this and instruction one is the period (.) before the requested address.

3. You can directly cause a dump by putting the period between two addresses: *2000.2010 combines the actions of instructions one and two above.

4. Hitting RETURN will continue a dump, one line at a time.

5. The last displayed memory location can be *changed* by using the colon (:). This is the equivalent of BASIC's POKE. If *2000 results in FF on the screen, you can change this FF to zero by typing *:00. To see the change, type *2000 again. Or you could type *2000:00 and make the change directly.

The Apple II reference manual contains excellent descriptions of the monitor instructions. We will list the rest of them only briefly here:

6. Change a series of locations at once: *2000: 00 69 15 65 12.

7. Move (transfer) a section of memory: *4000<2000.2010M will copy what's between 2000 and 2010 up to address 4000. (All these addresses are hex.)

8. Compare two sections of memory: *4000<2000.2010V. This looks like Move, but its job is to see if there are any differences between the numbers in the memory cells from 2000-2010 and those from 4000-4010. If differences are found, the address where the difference occurs appears on screen. If the two memory ranges are identical, nothing is printed on the screen.

9. Saving (writing) a section of ML to tape: *2000.2010W. This is how you would save an ML program. You specify the addresses of the start and end of your program.

10. Loading (reading) a section of memory (or an ML program) back into the computer from tape: *2000.2010R will put the bytes saved, in instruction nine, above, back where they were when you saved them.

An interesting additional feature is that you could send the bytes to *any* address in the computer. To put them at 4000, you would just type *4000.4010R. This gives you another way to relocate subroutines or entire ML programs (in addition to the Move instruction, number seven above). If you move an ML program to reside at a different address from the one it was originally intended during assembly, any JMP or JSR (Jump To Subroutine, like BASIC's GOSUB) instructions which point to within your program must be adjusted to point to the new addresses. If your subroutine contained an instruction such as 2000 JSR 2005, and you loaded at 4000, it would still say 4000 JSR 2005. You would have to change it to read 4000 JSR 4005. All the BNE, BPL, BEQ, *branching* instructions, though, will make the move without damage. They are *relative* addresses (as opposed to the *absolute* addressing of JSR 2005). They will not need any adjusting. We'll go into this in detail later.

11. Run (go): *2000G will start executing the ML program which begins at address 2000. There had better be a program there or the machine is likely to lock up, performing some nonsense, an endless loop, until you turn off the power or press a RESET key. The program or subroutine will finish and return control of the computer to the monitor when it encounters an RTS. This is like BASIC's SYS command, except the computer returns to the monitor mode.

12. Disassemble (list): *2000L will list 20 lines of ML on the screen. It will contain three *fields* (a field is a ''zone'' of information). The first field will contain the address of an instruction (in hex). The address field is somewhat comparable to BASIC's line numbers. It defines the order in which instructions will normally be carried out.

Here's a brief review of *disassembly* listings. The second field shows the hex numbers for the instruction, and the third field is where a disassembly differs from a ''memory'' or ''hex'' dump (see numbers one and two, above). This third field translates the hex numbers of the second field back into a mnemonic and its argument. Here's an example of a disassembly:

```
2000    A9 41        LDA    #$41
2002    8D 23 32     STA    $3223
2005    A4 99        LDY    $99
```

Recall that a dollar sign ($) shows that a number is in hexadecimal. The pound sign (#) means ''immediate'' addressing (put the *number itself* into the A register at 2000 above). Confusing these two symbols is a major source of errors for beginning ML programmers. You should pay careful attention to the distinction between LDA #$41 and LDA $41. The second instruction (without the pound sign) means to load A with whatever number is found in *address* $41 hex. LDA #$41 means put the *actual number 41 itself* into the accumulator. If you are debugging a routine, check to see that you've got these two types of numbers straight, that you've loaded from addresses where you meant to (and, vice versa, you've loaded *immediately* where you intended).

13. Mini-assembler. This is an assembler program, though it is not part of the monitor ROM. It is in the Integer BASIC ROM, so systems using firmware Applesoft II cannot use it although the Apple II Plus can, in the INT mode. Like the Simple Assembler, this mini-assembler cannot use labels or calculate forward branches. (The Simple Assembler can be used for forward branches, however, as we'll see later.) You enter the Apple mini-assembler by typing the

address, mnemonic, and argument of your first instruction. The ! is printed by the computer:

!2000:LDA #15

This will be disassembled, and then you type in the next line, using spaces between each field:

! LDY #01

14. Step and Trace. These are very useful ways to isolate and fix errors. Remember that ML does not have much in the way of error messages. In fact, unless you are using a very complex assembler program, the only error that an assembler can usually detect is an impossible mnemonic. If you mistyped LDA as LDDA, your assembler would print ??? or, in the Apple, sound a beep and put a circumflex (^) near the error. In any case, you are not going to get elaborate SYNTAX ERROR messages. The Simple Assembler will type the word ERROR on the screen. Try it.

We'll examine step and trace debugging methods under numbers 10 and 11 of the discussion of the Atari cartridge below. The Atari Assembler Cartridge and the Commodore Monitor Extension programs both allow step and trace, too.

15. Changing registers. *(CONTROL) E will display the contents of the Accumulator, the X and Y registers, the status register (P) and the stack pointer (S). You can then change the contents of these registers by typing them in on screen, following a colon. Note that to change the Y register, you must type in the A and X registers as well:

*** (CONTROL) E**

You'll see: A=01 X=05 Y=FF P=30 S=FE (whatever's in the registers at the time).

To change the Y register to 00, you type in the A, X, and then the new version of Y:

***:01 05 00 (and hit RETURN)**

16. Going back to BASIC. You can use * (CONTROL) B to go to BASIC (but it will wipe out any BASIC program that might have been there). Or you can use * (CONTROL) C to go back to BASIC, non-destructively.

The Atari Monitor

To enter the monitor on the Atari, you put the assembler cartridge into the left slot. The Atari does not have a monitor in ROM; you need the cartridge. As mentioned at the start of this chapter, the monitor mode in Atari is called DEBUG and is a part of the larger program within the assembler cartridge. There are three parts (or modes) within the cartridge: EDIT, ASM (assembler), and DEBUG. Before looking at the commands available in the DEBUG mode, let's briefly explore how an ML program is created using the EDIT mode followed by ASM. The cartridge provides the Atari with a more advanced assembler than the Simple Assembler or the mini-assemblers available within the Apple II monitor or the Commodore monitor extension programs. The cartridge allows labels, comments, and line numbers.

Until now, we've discussed ML programming which uses three *fields* (zones). Here's an example program which shows these three simple fields. We will print ten "A's" on the screen (the numbers are decimal):

Address Field	Instruction Field	Argument (Operand) Field
2000	LDY	#10
2002	LDA	#33
2004	STA	(88),Y
		(The screen location is remembered by the Atari in addresses 88 and 89.)
2007	DEY	
2008	BNE	2004
2010	RTS (or BRK)	

When you are in Atari's EDIT mode, you construct a program somewhat differently than you do with the Simple Assembler (or with mini-assemblers). Here's the same program using the Atari's additional fields:

Line #	Label	Instruction	Argument	Comments
100	START	LDY	#10	Set up counter for loop
110		LDA	#33	"A" in ATASCII
120	LOOP	STA	(88),Y	
130		DEY		
140		BNE	LOOP	Loop until zero

Notice that labels allow us to use the word *LOOP* instead of the specific address we want to loop back to. In addition to all this, there are *pseudo-ops* which are instructions to the assembler to perform some task. A pseudo-op does not become part of the ML program (it's not a 6502 instruction), but it affects the assembly process in some way. We would need two pseudo-ops in the above program to allow it to be assembled properly. Add these lines:

10 *=$0600 (tells the assembler that this program should be assembled starting at address $0600. The $ means hexadecimal.)
160 .END (tells the assembler that it should stop assembling here.)

The example above with line numbers and labels is called *source code* because it is the source from which the assembler gets its information when it assembles *object code* (object code is an actual ML program which could be run, or executed). You cannot run the program above as is. It must first be assembled into 6502 ML. For one thing, the label *LOOP* has to be replaced with the correct branch back to line 120. Source code does not put bytes into memory as you write it (as a more elementary assembler like the Simple Assembler does).

More Than A Monitor

To make this into *object code* which you can actually execute, you type ASM (for assemble), and the computer will put the program together and POKE the bytes into memory, showing you on screen what it looks like.

To test the program, type BUG to enter the DEBUG mode, clear the screen, and RUN it by typing G600 (for GO $0600). You'll see AAAAAAAAAA on screen. It works!

All this isn't, strictly speaking, a monitor. It's a full assembler. The part of the assembler cartridge program which is equivalent to the monitor programs on Apple II and PET is the DEBUG mode. There are a number of commands in DEBUG with which you can examine, test, and correct ML code. As on the other computers, the DEBUG (monitor) mode allows you to work closely with single bytes at a time, to see the registers, to trace program flow. All numbers you see on screen (or use to enter into the computer) are in hex. You enter the DEBUG mode by typing BUG when the Assembler Cartridge is in the Atari. (To go back to EDIT mode, type X.) Here are the commands of DEBUG:

1. Display the registers: type DR (RETURN) and you will see whatever is in the various registers.

A=01 X=05 Y=0F P=30 S=FE (P is the status register and S is the stack pointer.)

2. Change the registers: type CR<6,2 (RETURN) and you will have put a six into the accumulator and a two into the X register. To put a five into the status register, you must show how far to go by using commas: CR<,,,5 would do it. CR<5 would put five into the accumulator.

3. Dump memory: type D2000 and you will see the eight hex numbers which start at address 2000 in memory.

D2000
2000 FF 02 60 20 FF D2 00 00
D2000,2020 (would dump out memory between these two addresses)

4. Change memory: type C2000<00,00 to put zeros into the first two bytes following address 2000.

5. Transfer (move) memory: type M1000<2000,2010 and you will non-destructively copy what's between 2000-2010 down into 1000-1010.

6. Compare (verify) memory: type V1000<2000,2010 and any mismatches will be printed out.

7. Disassemble (list): type L2000 and you will see 20 lines of instructions displayed, the mnemonics and their arguments.

8. Mini-assemble: the DEBUG mode allows you to enter mnemonics and arguments one at a time, but you cannot use labels. (The pseudo-ops BYTE, DBYTE, and WORD are available, though.) This is similar to the Simple Assembler and the mini-assemblers available to Apple II and PET monitor users.

You type 2000<LDA $05 and the computer will show you the bytes as they assemble into this address. Subsequent instructions can be entered by simply using the less-than sign again: < INC $05. To return to the DEBUG mode, you can hit the RETURN key on a blank line.

9. Go (RUN a program): type G2000 and whatever program starts at address 2000 will run. Usually, you can stop the RUN by hitting the BREAK key. There are cases, though, (endless loops) which will require that you turn off the computer to regain control.

10. Trace: type T2000 and you will also RUN your program, but the registers, bytes of ML code, and the disassembled mnemonics and arguments are shown as each instruction is executed. This is especially useful since you can watch the changes taking place in the registers and discover errors. If you have an LDA $03 and you then expect to find the accumulator to have the number three in it — you'll notice that you made that very common mistake we talked about earlier. Following LDA $03, you will see that the accumulator has, perhaps, a ten in it instead of the three you thought you'd get. Why? Because you wanted to write LDA #03 (immediate). Instead, you mistakenly loaded A with the value *in address three,* whatever it is.

Seeing unexpected things like this happen during trace allows you to isolate and fix your errors. Trace will stop when it lands on a BRK instruction or when you press the BREAK key.

11. Step: type S2000 and you will "step" through your program at 2000, one instruction at a time. It will look like trace, but you move slowly and you control the rate. To see the following instruction, you type the S key again. Typing S over and over will bring you through

the program.

12. Return to EDIT mode: type X.

PET, VIC, And Commodore 64 Monitors

The resident monitor on the PET/CBM computer is the simplest of monitors. You enter it from BASIC by typing SYS 4 when no program is RUNning. This lands on a BReaK instruction; address 4 always contains a zero which is the opcode for BRK. You are then in monitor mode. Original ROM PETs, the earliest models, do not have a monitor in ROM, but one is available on tape, called TIM. Everything is done with hexadecimal numbers.

There are only six monitor commands:

1. Go (RUN) : type G 2000 and the program starts at address 2000. It will continue until it lands on a BRK instruction. There is no key you can type to stop it.

2. LOAD (from tape or disk) : type L ''0:NAME'',08 and a program called ''name'' on disk drive zero will be loaded at the address from which it was SAVEd. There is no provision to allow you to LOAD to a different address. L ''NAME'',01 will LOAD from tape.

3. SAVE (to a tape or disk): type S ''0:NAME'',08,2000,2009 and the bytes between hex 2000 and 2008 will be saved to disk drive zero and called ''name.'' *Important note:* you should always be aware that a SAVE *will not save the highest byte* listed in your SAVE instruction. You always specify *one byte more* than you want to save. In our example here, we typed 2009 as our top address, but the monitor SAVEd only up to 2008. S ''NAME'',01,2000,2009 will SAVE to tape.

An interesting trick is to save the picture on your screen. Try this from the monitor (for a disk drive) : S ''0:SCREEN'',08,8000,8400 (with a tape drive: S ''SCREEN'',01,8000,8400). Then, clear the screen and type: L ''0:SCREEN'',08 (tape: L ''SCREEN'',01). This illustrates that an ML SAVE or LOAD just takes bytes from within whatever range of memory you specify; it doesn't care what those bytes contain or if they make ML sense as a program.

4. See memory (memory dump): type M 2000 2009 and the bytes between these addresses will be displayed on screen. To change them, you use the PET cursor controls to move to one of these hex numbers and type over it. Hitting the RETURN key makes the change in the computer's memory (the same way you would change a line in BASIC).

Machine Language Registers

5. See the registers: type R and you will see something like this on screen (the particular numbers in each category will depend on what's going on in your computer whenever you type R):

PC	IRQ	SR	AC	XR	YR	SP
2000	E62E	30	00	05	FF	FE

The PC is the program counter: above, it means that the next instruction the computer would perform is found at address 2000. If you typed G (for RUN), this is where it would start executing. The IRQ is the interrupt request. The SR is the status register (the condition of the flags). The AC is the accumulator, the XR and YR are the X and Y registers. The SP is the stack pointer. We'll get into all this later.

6. Exit to BASIC: type X.

That's it. Obviously, you will want to add trace, step, transfer, disassemble, and other useful monitor aids. Fortunately, they are available. Two programs, *Supermon* and *Micromon,* can be LOADed into your Commodore computer and will automatically attach themselves to your "resident" monitor. That is, when you're in the monitor mode, you can type additional monitor commands.

Both *Micromon* and *Supermon* are widely available through user groups (they are in the public domain, available to everyone for free). If there is no user group nearby, you can type them in yourself. *Supermon* appeared in *COMPUTE!* Magazine, December 1981, Issue #19, on page 134. *Micromon* appeared in *COMPUTE!*, January 1982, Issue #20, page 160. A *Micromon* for VIC can be found in *COMPUTE!*, November 1982. Because of their value, particularly when you are debugging or analyzing ML programs, you will want to add them to your program library. Several of these monitor extensions can be found in Appendix F.

Using The Monitors

You will make mistakes. Monitors are for checking and fixing ML programs. ML is an exacting programming process, and causing bugs is as unavoidable as mistyping when writing a letter. It will happen, be sure, and the only thing for it is to go back and try to locate and fix the slip-up. It is said that every Persian rug is made with a deliberate mistake somewhere in its pattern. The purpose of this is to show that only Allah is perfect. This isn't our motivation when causing bugs in an ML program, but we'll cause them nonetheless. The best you can do is try to get rid of them when they appear.

Probably the most effective tactic, especially when you are just starting out with ML, is to write very short sub-programs (subroutines). Because they are short, you can more easily check each one to make sure that it is functioning the way it should. Let's assume that you want to write an ML subroutine to ask a question on the screen. (This is often called a *prompt* since it prompts the user to do something.)

The message can be: "press any key." First, we'll have to store the message in a data table. We'll put it at hex $1500. That's as good a place as anywhere else. Remember that your computer may be using a different screen RAM POKE code to display these letters. POKE the letter "A" into your screen RAM to see what number represents the start of your screen alphabet and use those numbers for any direct-to-screen messages in this book.

have to be converted to hex

	ASCII		ATARI	
1500	80	P	48	36 30
1501	82	R	50	38 32
1502	69	E	37	25 25
1503	83	S	51	39 33
1504	83	S	51	39 33
1505	32		0	0
1506	65	A	33	21
1507	78	N	46	2E
1508	89	Y	57	39
1509	32		0	0
150A	75	K	43	2B
150B	69	E	37	25
150C	89	Y	57	39
150D	00		255	FF

(the delimiter, the signal that the message is finished. Atari must use something beside zero which is used to represent the space character.)

We'll put the subroutine at $1000, but be warned! This subroutine will not work as printed. There are two errors in this program. See if you can spot them:

```
1000 LDY #$00
1002 LDA $1500,Y
1005 CMP $00      (is it the delimiter?)
1007 BNE $100A    (if not, continue on)
1009 RTS          (it was zero, so quit and return to whatever
                   JSRed, or called, this subroutine)
100A STA $8000,Y (for PET)
100D INY
100E JMP $1000    (always JMP back to $1000)
```

Make the following substitutions if you use one of these machines:

```
Atari: 1005 CMP $FF         (That's hex for 255.)
Atari: 100A STA ($88),Y
Apple: 100A STA $0400,Y
```

Since we haven't yet gone into addressing or instructions much, this is like learning to swim by the throw-them-in-the-water method. See if you can make out some of the meanings of these instructions anyway.

This subroutine will not work. There are two errors and they are two of the most common bugs in ML programming. Unfortunately, they are not obvious bugs. An obvious bug would be mistyping: LDS when you mean LDA. That sort of bug would be caught by your assembler, and it would print an error message to let you know that no such instruction as LDS exists in 6502 ML.

The bugs in this routine are mistakes in logic. If you disassemble this, it will also look fine to the disassembler, and no error messages will be printed there either. But, it will not work the way you wanted it to. Before reading on, see if you can spot the two errors. Also see if you can figure out how this routine would execute its instructions. Where does the computer go after the first pass through this code? When and how does it finish the job?

Two Common Errors

A very common bug, perhaps the most common ML bug, is caused by accidentally using *zero page addressing* when you mean to use *immediate addressing*. We mentioned this before, but it is the cause of so much puzzlement to the beginning ML programmer that we'll go over it several times in this book. Zero page addressing looks very similar to immediate addressing. Zero page means that you are addressing one of the cells in the first 256 addresses. A *page* of memory is 256 bytes. The lowest page is called *zero page* and is the RAM cells from number zero through 255. Page one is from 256-511 (this is the location of the "stack" which we'll get to later). Addresses 512-767 are page three and so on up to the top memory, page 255.

Immediate addressing means that the number is right within the ML code, that it's the number which follows (which is the operand or the argument of) an instruction. LDA #13 is immediate. It puts the number 13 into the accumulator. LDA $13 is zero page and puts *whatever number is in address 13* into the accumulator. It's easy and very common to mix up these two, so you might look at these instructions first when debugging a faulty program. See that all your zero page addressing is supposed to be zero page and that all your immediate addressing is supposed to be immediate.

In the prompt example above, the LDY #00 is correct — we do want to set the Y register counter to zero to begin printing the message. So we want an immediate, the *actual* number zero. Take a good look, however, at the instruction at location $1005. Here we are not asking the computer to compare the number in the accumulator to zero. Instead, we are asking the computer to compare it to whatever might be in *address* zero — with unpredictable results. To fix this bug, the instruction should be changed to the immediate addressing mode with CMP # 0.

The second bug is also a very common one. The subroutine, as written, can never leave itself. It is an endless loop. Loop structures are usually preceded by a short initialization phase. The counters have to be set up before the loop can begin. Just as in BASIC, where FOR I =1 TO 10 tells the loop to cycle ten times, in ML, we set the Y register to zero to let it act as our counter. It kills two birds with one stone in this subroutine. It is the offset (a pointer to the current position in a list or series) to load from the message in the data table and the offset to print to the screen. Without Y going up one (INY) each time through the loop, we would always print the first letter of the message, and always in the first position on the screen.

What's the problem? It's that JMP instruction at $100E. It sends us back to the LDY # 0 address at 1000. We should be looping back to address 1002. As things stand, the Y register will always be reset to zero, and there will never be any chance to pick up the delimiter and exit the subroutine. An endless cycle of loading the "P" and printing it will occur. Y will never get beyond zero because each loop jumps back to 1000 and puts a zero back into Y. To see this, here's the same bug in BASIC:

```
10 T=5
20 T=T+1
30 IF T=10 THEN 50
40 GOTO 10
```

Tracking Them Down

The monitor will let you discover these and other errors. You can replace an instruction with zero (BRK) and see what happens when you execute the program up to the BRK. Better yet, you can single step through the program and see that, for example, you are not really computing CMP #00 where you thought you were. It would also be easy to see that the Y register is being reset to zero each time through the loop. You are expecting to use it as a counter and it's not cooperating, it's not counting up each time through the loop. These and other errors are, if not obvious, at least discoverable from the monitor.

Also, the disassembler function of the monitor will permit you to study the program and look, deliberately, for correct use of #00 and $00. Since that mix-up between immediate and zero page addressing is so common an error, always check for it first.

Programming Tools

The single most significant quality of monitors which contributes to easing the ML programmer's job is that monitors, like BASIC, are *interactive.* This means that you can make changes and test them right away, right then. In BASIC, you can find an error in line 120, make the correction, and RUN a test immediately.

It's not always that easy to locate and fix bugs in ML: there are few, if any, error messages, so finding the location of a bug can be difficult. But a monitor does allow interactivity: you make changes and test them on the spot. This is one of the drawbacks of complex assemblers, especially those which have several steps between the writing of the source code and the final assembly of executable object code (ML which can be executed).

These assemblers often require several steps between writing an ML program and being able to test it. There are linkers, relocatable loaders, double-pass assembly, etc. All of these functions make it easier to rearrange ML subroutines, put them anywhere in memory without modification, etc. They make ML more modular (composed of small, self-sufficient modules or subroutines), but they also make it less interactive. You cannot easily make a change and see the effects at once.

However, using a mini-assembler or the Simple Assembler, you are right near the monitor level and fixes can easily and quickly be tested. In other words, the simpler assemblers sometimes gain in efficiency what they lose in flexibility. The simpler assemblers support a style of programming which involves less pre-planning, less forethought, less abstract analysis. If something goes awry, you can just try something else until it all works.

Plan Ahead Or Plunge In?

Some find such trial and error programming uncomfortable, even disgraceful. The more complicated assemblers discourage interactivity and expect careful preliminary planning, flowcharts, even writing out the program ahead of time on paper and debugging it there. In one sense, these large assemblers are a holdover from the early years of computing when computer time was extremely expensive. There was a clear advantage to coming to the terminal as prepared as possible. Interactivity was costly. But, like the increasingly outdated advice urging programmers to worry about saving computer memory space, it seems that strategies designed to conserve computer time are anachronistic. You can spend all the time you want on your personal computer.

Complex assemblers tend to downgrade the importance of a monitor, to reduce its function in the assembly process. Some programmers who've worked on IBM computers for 20 years do not use the word *monitor* in the sense we are using it. To them, monitors are CRT screens. The deluxe assembler on the SuperPet, for example, does have a monitor, but it has no single-step function and has no provision for SAVEing an ML program to disk or tape from the monitor.

Whether or not you are satisfied with the interactive style of simple, mini-assemblers and their greater reliance on the monitor mode and on trial and error programming is your decision. If you want to graduate to the more complicated assemblers, to move closer to high-level languages with labels and relocatable code, fine. The Atari assembler is fairly high-level already, but it does contain a full-featured monitor, the "debugger," as well. The choice is ultimately a matter of personal style.

Some programmers are uncomfortable unless they have a fairly complete plan before they even get to the computer keyboard. Others are quickly bored by elaborate flowcharting, "dry computing" on paper, and can't wait to get on the computer and see-what-happens-if. Perhaps a good analogy can be found in the various ways that people make telephone calls. When long-distance calls were extremely expensive, many people made lists of what they wanted to say and carefully planned the call before dialing. They would also watch the clock during the call. (Some still do this today.) As the costs of phoning came down, most people found that spontaneous conversation was more satisfying. It's up to you.

Computer time, though, is now extremely cheap. If your computer uses 100 watts and your electric company charges five cents per KWH, leaving the computer on continuously costs about 12 cents a day.

4

Addressing

The 6502 processor is an electronic brain. It performs a variety of manipulations with numbers to allow us to write words, draw pictures, control outside machines such as tape recorders, calculate, and do many other things. Its manipulations were designed to be logical and fast. The computer has been designed to permit everything to be accomplished accurately and efficiently.

If you could peer down into the CPU (Central Processing Unit), the heart of the computer, you would see numbers being delivered and received from memory locations all over the computer. Sometimes the numbers arrive and are sent out, unchanged, to some other address. Other times they are compared, added, or otherwise modified, before being sent back to RAM or to a peripheral.

Writing an ML program can be compared to planning the activities of this message center. It can be illustrated by thinking of computer memory as a city of bytes and the CPU as the main post office. (See Figure 4-1.) The CPU does its job using several tools: three registers, a program counter, a stack pointer, and seven little one-bit flags contained in a byte called the Status Register. We will only concern ourselves with the "C" (carry) flag and the "Z" (it equals zero) flags. The rest of them are far less frequently needed for ML programming so we'll only describe them briefly. (See Figure 4-1.)

Most monitors, after you BRK (like BASIC's STOP) out of a program, will display the present status of these tools. It looks something like this:

Program 4-1. Current Status Of The Registers.

```
PC   IRQ  SR AC XR YR SP
0005 E455 30 00 5E 04 F8
```

The PC is the Program Counter and it is two bytes long so it can refer to a location anywhere in memory. The IRQ is also two bytes and points to a ROM ML routine which handles interrupts, special-priority actions. A beginning ML programmer will not be working with interrupts and need not worry about the IRQ. You can also more or less let the computer handle the SP on the end. It's the stack

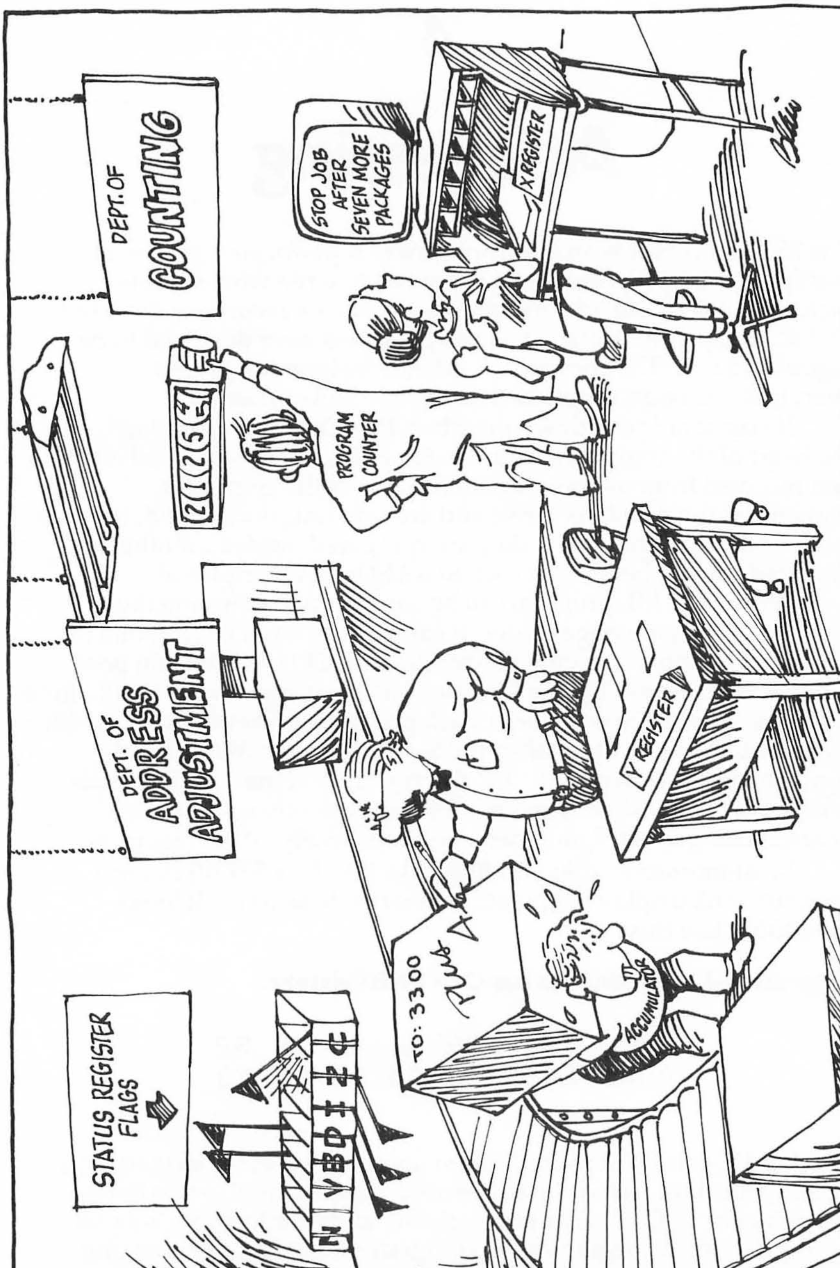


Figure 4-1. Postal Executives At Work On An Instruction: 21254 STA 3300, Y.

pointer. The SP keeps track of numbers, usually return-from-subroutine addresses which are kept together in a list called the stack.

The computer will automatically handle the stack pointer for us. It will also deal with IRQ and the program counter. For example, each ML instruction we give it could be one, two, or three bytes long. TYA has no argument and is the instruction to transfer a number from the Y register to the accumulator. Since it has no argument, the PC can locate the next instruction to be carried out by raising itself by one. If the PC held $4000, it would hold $4001 after execution of a TYA. LDA #$01 is a two-byte instruction. It takes up two bytes in memory so the next instruction to be executed after LDA #$01 will be two bytes beyond it. In this case, the PC will raise itself from $4000 to $4002. But we can just let it work merrily away without worrying about it.

The Accumulator: The Busiest Register

The SR, AC, XR, and YR, however, are our business. They are all eight bits (one byte) in size. They are not located in memory proper. You can't PEEK them since they have no address like the rest of memory. They are zones of the CPU. The AC, or A register, but most often called the *accumulator,* is the busiest place in the computer. The great bulk of the mail comes to rest here, if only briefly, before being sent to another destination.

Any logical transformations (EOR,AND) or arithmetic operations leave their results in the accumulator. Most of the bytes streaming through the computer come through the accumulator. You can compare one byte against another using the accumulator. And nearly everything that happens which involves the accumulator will have an effect on the status register (SR, the flags).

The X and Y registers are similar to each other in that one of their main purposes is to assist the accumulator. They are used as addressing indexes. There are addressing modes that we'll get to in a minute which add an index value to another number. For example, LDA $4000,X will load into A the number found in address $4005, if the X register is currently holding a five. The address is the number *plus* the index value. If X has a six, then we load from $4006. Why not just LDA $4006? It is far easier to raise or lower an index inside a loop structure than it would be to write in each specific address literally.

A second major use of X and Y is in counting and looping. We'll go into this more in the chapter on the instruction set.

We'll also have some things to learn later about the SR, the Status Register which holds some flags showing current conditions. The SR can tell a program or the CPU if there has been a zero, a carry, or a negative number as the result of some operation, among other things. Knowing about carry and zero flags is especially significant in ML.

For now, the task at hand is to explore the various ''classes'' of mail delivery, the 6502 addressing modes.

Aside from comparing things and so forth, the computer must have a logical way to pick up and send information. Rather like a postal service in a dream — everything should be picked up and delivered rapidly, and nothing should be lost, damaged, or delivered to the wrong address.

The 6502 accomplishes its important function of getting and sending bytes (GET and PRINT would be examples of this same thing in BASIC) by using several "addressing modes." There are 13 different ways that a byte might be "mailed" either to or from the central processor.

When programming, in addition to picking an instruction (of the 56 available to you) to accomplish the job you are working on, you must also make one other decision. You must decide how you want to *address* the instruction — how, in other words, you want the mail sent or delivered. There is some room for maneuvering. You will probably not care if you accidentally choose a slower delivery method than you could have. Nevertheless, it is necessary to know what choices you have: most addressing modes are designed to aid a common programming activity.

Absolute And Zero

Let's picture a postman's dream city, a city so well planned from a postal-delivery point of view that no byte is ever lost, damaged, or sent to the wrong address. It's the City of Bytes we first toured in Chapter 2. It has 65536 houses all lined up on one side of a street (a long street). Each house is clearly labeled with its number, starting with house zero and ending with house number 65535. When you want to get a byte from, or send a byte to, a house (each house holds one byte) — you must "address" the package. (See Figure 4-2.)

Here's an example of one mode of addressing. It's quite popular and could be thought of as "First Class." Called *absolute* addressing, it can send a number to, or receive one from, any house in the city. It's what we normally think of first when the idea of "addressing" something comes up. You just put the number on the package and send it off. No indexing or special instructions. If it says 2500, then it means house 2500.

1000 STA $2500

or

1000 LDA $2500

These two, STore A and LoaD A, STA and LDA, are the instructions which get a byte from, or send it to, the accumulator. The *address,* though, is found in the numbers following the instruction. The items following an instruction are called the instruction's *argument.* You could have written the address several ways. Writing it as $2500 tells your assembler to get it from, or send it directly to, hex $2500. This kind of addressing uses just a simple $ and a four-digit

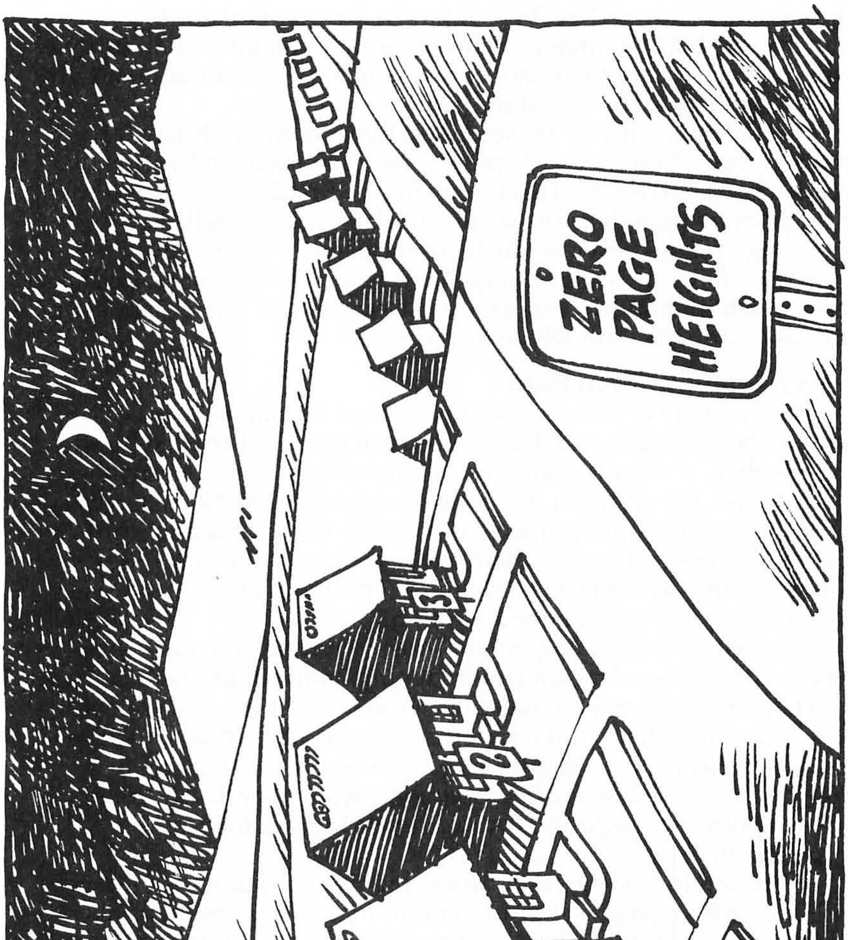


Figure 4-2. The First Few Addresses On A Street With 65536 Houses.

number. You can send the byte sitting in the accumulator to anywhere in RAM memory by this method. Remember that the byte value, although sent to memory, also remains in the accumulator. It's more a copying than a literal sending.

To save time, if you are sending a byte down to address 0 through 255 (called the "zero page"), you can leave off the first two numbers: 1000 STA $07. This is only for the first 256 addresses, but they get more than their share of mail. Your machine's BASIC and operating system (OS) use much of zero page for their own temporary flags and other things. Zero page is a busy place, and there is not much room down there for you to store your own ML pointers or flags (not to mention whole routines).

Heavy Traffic In Zero Page

This second way to address, using only two hex digits, any hex number between $00 and $FF or a decimal number between 0 and 255, is called, naturally enough, *zero page addressing*. It's pretty fast mail service: the deliverer has to decide among only 256 instead of 65536 houses, and the computer is specially wired to service these special addresses. Think of them as being close to the post office. Things get in and out fast at zero page. This is why your BASIC and operating system tend to use it so often.

These two addressing modes — absolute and zero page — are very common ones. In your programming, you will probably not use zero page as much as you might like. You will notice, on a map of your computer's flags and temporary storage areas, that zero page is heavily trafficked. You might cause a problem storing things in zero page in places used by the OS (operating system) or BASIC. Several maps of both zero page and BASIC in ROM can be found in Appendix B.

You can find safe areas to store your own programs' pointers and flags in zero page. A buffer (temporary holding area) for the cassette drive or for BASIC's floating point numbers might be used only during cassette loads and saves or during BASIC RUNs to calculate numbers. So, if your flags and pointers were stored in these addresses, things would be fine unless you involved cassette operations. In any case, zero page is a popular, busy neighborhood. Don't put any ML programs in there. Your main use of zero page is for the very efficient "indirect Y" addressing we'll get to in a minute. But you've always got to check your computer's memory map for zero page to make sure that you aren't using bytes which the computer itself uses.

By the way, don't locate your ML programs in page one (256-511 decimal) either. That's for the "stack," about which more later. We'll identify where you can safely store your ML programs in the various computers. It's always OK to use RAM as long as you keep BASIC programs from putting their variables on top of ML, and keep ML from writing over your BASIC assembler program (such as the Simple Assembler).

Immediate

Another very common addressing mode is called *immediate* addressing — it deals directly with a number. Instead of sending out for a number, we can just shove it immediately into the accumulator by putting it right in the place where other addressing modes have an address. Let's illustrate this:

```
1000 LDA $2500        (Absolute mode)
1000 LDA #$09         (Immediate mode)
```

The first example will load the accumulator with whatever number it finds at address $2500. In the second example, we simply wanted to put a 9 into the accumulator. We know that we want the number 9. So, instead of sending off for the 9, we just type a 9 in where we would normally type a memory address. And we tack on a # symbol to show that the 9 is the number we're after. Without the #, the computer will load the accumulator with whatever it finds at *address* number 9 (LDA $09). That would be zero page addressing, instead of immediate addressing.

In any case, immediate addressing is very frequently used, since you often know already what number you are after and do not need to send for it at all. So, you just put it right in with a #. This is similar to BASIC where you define a variable (10 VARIABLE =9). In this case, we have a variable being given a known value. LDA #9 is the same idea. In other words, immediate addressing is used when you know what number you want to deal with; you're not sending off for it. It's put right into the ML code *as a number, not as an address.*

To illustrate *immediate* and *absolute* addressing modes working together, let's imagine that we want to copy a 15 into address $4000. (See Program 4-2.)

Implied

Here's an easy one. You don't use *any* address or argument with this one.

This is among the more obvious modes. It's called *implied,* since the mnemonic itself implies what is being sent where: TXA means transfer X register's contents to the Accumulator. Implied addressing means that you do not put an address after the instruction (mnemonic) the way you would with most other forms of addressing.

It's like a self-addressed, stamped envelope. TYA and others are similar short-haul moves from one register to another. Included in this implied group are the SEC, CLC, SED, CLD instructions as well. They merely clear or set the flags in the status register, letting you

Program 4-2. Putting An Immediate 15 Into Absolute Address 4000.

```
                0010            .BA $2000           ; STARTING ADDRESS OF THIS
                0020 ; ML PROGRAM IS $2000 ("BA" = "BEGINNING ADDRESS").
                0030 ;
2000- A9 0F     0040            LDA #15             ; LOAD A WITH LITERALLY 15
2002- 8D 00 40  0050            STA $4000           ; STORE IT IN ADDRESS 4000
                0060 ;
                0070 ; NOTE THAT IN SOME ASSEMBLERS YOU CAN
                0080 ; SWITCH BETWEEN HEX AND DECIMAL.  THE
                0090 ; 15 IS DECIMAL, THE 4000 IS HEX.  A
                0100 ; LITERAL HEX 15 WOULD BE WRITTEN #$15.
                0110            .EN
```

and the computer keep track of whether an action resulted in a zero, if a ''carry'' has occurred during addition or subtraction, etc.

Also ''implied'' are such instructions as RTS (ReTurn from Subroutine), BRK (BReaK), PLP, PHP, PLA, PHA (which ''push'' or ''pull'' the processor status register or accumulator onto or off the stack). Such actions, and increasing by one (incrementing) the X or Y register's number (INX, INY) or decreasing it (DEX, DEY), are also called ''implied.'' What all of these implied addresses have in common is the fact that you do not need to actually give any address. By comparison, an LDA $2500 mode (the absolute mode) must have that $2500 address to know where to pick up the package. TXA already says, in the instruction itself, that the address is the X register and that the destination will be the accumulator. Likewise, you do not put an address after RTS since the computer always memorizes its jump-off address when it does a JSR (Jump to SubRoutine). NOP (No OPeration) is, of course, implied mode too.

Relative

One particular addressing mode, the *relative* mode, used to be a real headache for programmers. Not so long ago, in the days when ML programming was done ''by hand,'' this was a frequent source of errors. Hand computing — entering each byte by flipping eight switches up or down and then pressing an ENTER key — meant that the programmer had to write his program out on paper, translate the mnemonics into their number equivalents, and then ''key'' the whole thing into the machine. It was a big advance when computers would accept hexadecimal numbers which permitted entering 0F instead of eight switches: 00001111. This reduced errors and fatigue.

An even greater advance was when the machines began having enough free memory to allow an assembler program to be in the computer while the ML program was being written. An assembler not only takes care of translating LDA $2500 into its three (eight-switch binary) numbers: 10101101 00000000 00100101, but it also does relative addressing. So, for the same reason that you can program in ML without knowing how to deal with binary numbers — you can also forget about relative addressing. The assembler will do it for you.

Relative addressing is used with eight instructions only: BVS, BVC, BCS, BCC, BEQ, BMI, BNE, BPL. They are all ''branching'' instructions. Branch on: overflow flag set (or cleared), carry flag set (or cleared), equal, minus, not-equal, or plus. Branch if Not-Equal, like the rest of this group, will jump up to 128 addresses forward or backward from where it is or 127 addresses backward (if the result of the most recent activity is ''not equal''). Note that these jumps can be a distance of only 128, or 127 back, and they can go in either direction.

You specify *where* the jump should go by giving an address within these boundaries. Here's an example:

```
1000 LDX #$00
1002 INX
1003 BNE 1002
1005 BRK
```

(The X register will count up by ones until it hits 255 decimal and then it resets itself to zero.)

This is what you type in to create a ML FOR-NEXT loop. You are branching, relative to address 1003, which means that the assembler will calculate what address to place into the computer that will get you to 1002. You might wonder what's wrong with the computer just accepting the number 1002 as the address to which you want to branch. Absolute addressing *does* give the computer the actual address, but the branching instructions all need addresses which are ''offsets'' of the starting address. The assembler puts the following into the computer:

```
1000 A2 00
1002 E8
1003 D0 FD
1005 00
```

The odd thing about this piece of code is that ''FD'' at 1004. How does FD tell the computer to Branch back to 1002? (Remember that X will increment up to 255, then reset to zero on the final increment.) $FD means 253 decimal. Now it begins to be clear why relative addressing is so messy. If you are curious, numbers larger than 127, when found as arguments of relative addressing instructions, tell the computer to go *back down* to lower addresses. What's worse, the larger the number, the *less* far down it goes. It counts the address 1005 as zero and counts backwards thus:

```
1005=0
1004=255
1003=254
1002=253
```

Not a very pretty counting method! Luckily, all that we fortunate assembler users need do is to give the address (as if it were an *absolute* address), and the assembler will do the hard part. This strange counting method is the way that the computer can handle negative numbers. The reason it can only count to 128 is that the leftmost bit is no longer used as a 128th's column. Instead, this bit is on or off to signify a positive or negative number.

When you are using one of the branch instructions, you sometimes branch forward. Let's say that you want to have a different kind of FOR-NEXT loop:

```
1000 LDX #0
1002 INX
1003 BEQ 100A
1005 JMP 1002
1008 BRK
1009 BRK
100A BRK
```

When jumping forward, you often do not yet know the precise address you want to branch to. In the example above, we really wanted to go to 1008 when the loop was finished (when X was equal to zero), but we just entered an approximate address (100A) and made a note of the place where this guess appeared (1004). Then, using the POKE function on the assembler, we can POKE the correct offset when we know what it should be. Forward counting is easy. When we finally saw that we wanted to go to 1008, we would POKE 1004, 3. (The assembler would have written a five because that's the correct offset to branch to 100A, our original guess.)

Remember that the zero address for these relative branches is the address immediately following the branch instructions. For example, a jump to 1008 is three because you count: 1005 a zero, 1006=1, 1007=2, 1008=3. All this confusion disappears after writing a few programs and practicing with estimated branch addresses. Luckily, the assembler does all the backwards branches. That's lucky because they are much harder to calculate.

Unknown Forward Branches

Also, the Simple Assembler will do one forward (''not-yet-known'') branch calculation for you. If you look at the BASIC program listing of the Simple Assembler, you will see that the pseudo-ops (fake operations) are located from line 241 up. You could add additional forward-resolving pseudo-ops if you just give them new names like F1 resolved later by R1. Alternatively, you can type a guess in for the forward branches, as we just did in the example above. Then, when you find out the exact address, simply exit from the assembler, give 1004 as your starting address for assembly, and write in BEQ 1008 and let the assembler calculate for you. Either way, you will soon get the hang of forward branching.

We'll get into pseudo-ops later. Essentially, they are instructions to the assembler (such as ''please show me the decimal equivalent of the following hex number''), but which are not intended to be thought of as mnemonics which get translated into ML object code. Pseudo-ops are ''false'' operations, not part of the 6502 instruction set.

They are requests to the assembler program to perform some extra service for the programmer.

Absolute,X And Absolute,Y

Another important addressing mode provides you with an easy way to manipulate lists or tables. This method looks like absolute addressing, but it attaches an X or a Y to the address. The X or Y stands for the X or Y registers, which are being used in this technique as offsets. That is, if the X register contains the number 3 and you type: LDA 1000, X, you will LoaD the Accumulator with the value (the number) which is in memory cell 1003. *The register value is added to the absolute address.*

Another method called Zero Page,X works the same way: LDA 05,X. This means that you can easily transfer or search through messages, lists, or tables. Error messages can be sent to the screen using such a method. Assume that the words SYNTAX ERROR are held in some part of memory because you sometimes need to send them to the screen from your program. You might have a whole *table* of such messages. But we'll say that the words SYNTAX ERROR are stored at address 3000. Assuming that your screen memory address is 32768 (8000 hex), here's how you would send the message:

```
1000  LDX  #$00     (set the counter register to zero)
1002  LDA  $3000,X  (get a letter at 3000+X)
1005  BEQ  $100E    (if the character is a zero, we've
                     reached the end of message,
                     so we end the routine)
1007  STA  $8000,X  (store a letter on the screen)
100A  INX           (increment the counter so the next
                     letter in the message, as well as the
                     next screen position, are pointed
                     to)
100B  JMP  $1002    (jump to the load instruction to
                     fetch the next character)
100E  BRK           (task completed, message
                     transferred)
```

This sort of indexed looping is an extremely common ML programming device. It can be used to create delays (FOR T=1 TO 5000: NEXT T), to transfer any kind of memory to another place, to check the conditions of memory (to see, for example, if a particular word appears somewhere on the screen), and to perform many other applications. It is a fundamental, all-purpose machine language technique.

Here's a fast way to fill your screen or any other area of memory. This example uses the Commodore 64 Screen RAM starting address. Just substitute your computer's screen-start address. This is a full source code for the demonstration screen-fill we tried in Chapter 1. See if you can follow how this indexed addressing works. What bytes are filled in, and when? At ML speeds, it isn't necessary to fill them in order — nobody would see an irregular filling pattern because it all happens too fast for the eye to see it, like magic. (See Program 4-3.)

Compare this to Program 1-2 to see the effects of using a different screen starting address and how source code is an expansion of a disassembly.

Indirect Y

This one is a real workhorse; you'll use it often. Several of the examples in this book refer to it and explain it in context. It isn't so much an *address in itself* as it is a method of *creating* an address. It looks like this:

```
$4000 STA ($80),Y
```

Seems innocent enough. But watch out for the parentheses. They mean that $80 is *not* the real address we are trying to store A into. Instead, addresses $80 and $81 are *holding* the address we are really sending our byte in A to. We are not dealing directly with $0080 here; hence the name for this addressing mode: *indirect* Y.

If $80,81 have these numbers in them:

```
$0080 01
$0081 20
```

and Y is holding a five, then the byte in A will end up in address $2006! How did we get $2006?

First, we've got to mentally switch the numbers in $80,81. The 6502 requires that such ''address pointers'' be held in backwards order. So visualize $80,81 as forming $2001, a pointer. Then add the value in Y, which is five, and you get $2006.

This is a valuable tool and you should familiarize yourself with it. It lets you have easy access to many memory locations very quickly by just changing the Y register or the pointer. To go up a page, add one to the number in $0081. To go down four pages, subtract four from it. Combine this with the indexing that Y is doing for you and you've got great efficiency. The pointers for this addressing mode *must be stored in zero page locations.*

When an address is put into a pointer, you can see that it was split in half. The address $2001 was split in the example above. It's a two-byte number and ML terminology distinguishes between the bytes by saying that one is the LSB (least significant byte) and the other is the MSB (most significant byte). The $01 is the least significant. To grasp what is meant by ''significant,'' imagine chopping a decimal number such as 5015 in half. Since the left half, 50, stands for fifty 100's and the right half stands for 15 ones,

Program 4-3.

```
9C40- A0 00      0010         .BA 40000      ;(NOTICE IT'S DECIMAL)
                 0020 ;
                 0030 CHAR.A  .DE $41        ; CHARACTER "A"
                 0040 ;
9C40- A0 00      0050         LDY #$00       ; SET COUNTER TO ZERO.
9C42- A9 41      0060         LDA #CHAR.A
9C44- 99 00 04   0070 LOOP    STA $0400,Y
9C47- 99 00 05   0080         STA $0500,Y
9C4A- 99 00 06   0090         STA $0600,Y
9C4D- 99 00 07   0100         STA $0700,Y
9C50- C8         0110         INY            ; RAISE Y BY 1.
9C51- D0 F1      0120         BNE LOOP       ; IF NOT ZERO, KEEP GOING.
9C53- 60         0130         RTS
                 0140         .EN
```

obviously the leftmost half, the 100's, is more significant. Likewise, the left half of a two-byte hex number like $2001 is the most significant byte. The $20 stands for 32 times 256 (in decimal terms). It's easy to multiply double-byte numbers by decimal 256 by just adding one to the MSB. This would be a quick way of moving through the ''pages'' in memory.

The other thing to remember about MSB,LSB is that they are reversed when broken up and used as an address pointer: LSB,MSB.

Indirect X

Not often used, this mode makes it possible to set up a *group* of pointers (a table) in page zero. It's like Indirect Y except the X register value is not added to the address pointer to form the ultimate address desired. Rather, it points to which of the pointers to use. Nothing is added to the address found in the pointer.

It looks like this:

$5000 STA ($90,X)

To see it in action, let's assume that part of zero page has been set up to point to various parts of memory. A table of pointers, not just one:

$0090	**$00**	**Pointer #1**
$0091	**$04**	(it points to $0400)
$0092	**$05**	**Pointer #2**
$0093	**$70**	($7005)
$0094	**$EA**	**Pointer #3**
$0095	**$80**	(pointing to $80EA)

If X holds a two when we STA $(90,X), then the byte in A will be sent to $7005. If X holds a four, the byte will go to $80EA.

All in all, this has relatively little merit. It would be useful in rare situations, but at least it's there if you should find you need it.

Accumulator Mode

ASL, LSR, ROL, and ROR shift or manipulate the *bits* in the byte in the accumulator. We'll touch on them in the chapter on the instruction set. They don't have much to do with addressing, but they are always listed as a separate addressing mode.

Zero Page,Y

This can only be used with LDX and STX. Otherwise it operates just like Zero Page, X discussed above.

There you have them, thirteen addressing modes to choose from. The six you should focus on and practice are: Immediate, Absolute (plus Absolute,Y and ,X), Zero Page, and Indirect Y. The rest are either automatic (implied) or not really worth bothering with until you have full command of the six common and useful ones.

Now that we've surveyed the ways you can move numbers around, it's time to see how to do arithmetic in ML.

5

Arithmetic

There'll be many things you'll want to do in ML, but complicated math is not one of them. Mathematics beyond simple addition and subtraction (and a very easy form of elementary division and multiplication) will not be covered in this book. For most games and other ML for personal computing, you will rarely need to program with any complex math. In this chapter we will cover what you are likely to want to know. BASIC is well-suited to mathematical programming and is far easier to program for such tasks.

Before we look at ML arithmetic, it is worth reviewing an important concept: how the computer tells the difference between addresses, numbers as such, and instructions. It is valuable to be able to visualize what the computer is going to do as it comes upon each byte in your ML routine.

Even when the computer is working with words, letters of the alphabet, graphics symbols and the like — *it is still working with numbers.* A computer works *only* with numbers. The ASCII code is a convention by which the computer understands that when the context is alphabetic, the number 65 means the letter A. At first this is confusing. How does it know when 65 is A and when it is just 65? The third possibility is that the 65 could represent the 65th cell in the computer's memory.

It is important to remember that, like us, the computer means different things at different times when it uses a symbol (like 65). We can mean a street address by it, a temperature, the cost of a milk shake, or even a secret code. We could agree that whenever we used the symbol ''65'' we were ready to leave a party. The point is that symbols aren't anything in themselves. They *stand* for other things, and what they stand for must be agreed upon in advance. There must be rules. A code is an agreement in advance that one thing symbolizes another.

The Computer's Rules

Inside your machine, at the most basic level, there is a stream of input. The stream flows continually past a ''gate'' like a river through a canal. For 99 percent of the time, this input is zeros. (BASICs differ; some see continuous 255's, but the idea is the same.) You turn it on and the computer sits there. What's it doing? It might be updating a clock, if you have one, and it's holding things coherent on the TV screen — but it mainly waits in an endless loop for you to press a key on your keyboard to let it know what it's supposed to do. There is a memory cell inside (this, too, varies in its location) which the computer constantly checks. On some computers, this cell always has a 255 in it unless a key is pressed. If you press the RETURN key, a 13 will replace the 255. At last, after centuries (the computer's sense of time differs from ours) here is something to work with! Something has come up to the gate at long last.

You notice the effect at once — everything on the screen moves up one line because 13 (in the ASCII code) stands for carriage return. How did it know that you were not intending to type the *number* 13 when it saw 13 in the keyboard sampling cell? Simple. The number 13, and any other keyboard input, is *always* read as an ASCII number.

In ASCII, the digits from 0 through 9 are the only number symbols. There is no single symbol for 13. So, when you type in a 1 followed immediately by a 3, the computer's input-from-the-keyboard routine scans the line on the screen and notices that you have *not* pressed the "instant action" keys (the STOP, BREAK, ESC, TAB, cursor-control keys, etc.). Rather, you typed 1 and 3 and the keyboard sampling cell (the "which key pressed" address in zero page) received the ASCII value for one and then for three. ASCII digits are easy to remember in hex: zero is 30, 1 is 31, and up to 39 for nine. In decimal, they are 48 through 57.

The computer decides the "meaning" of the numbers which flow into and through it by the numbers' *context*. If it is in "alphabetic" mode, the computer will see the number 65 as "a"; or if it has just received an "a," it might see a subsequent number 65 as an address to store the "a". It all depends on the events that surround a given number. We can illustrate this with a simple example:

```
2000   LDA #65     A9 (169)  41 (65)
2000   STA $65     85 (133)  41 (65)
```

This short ML program (the numbers in parentheses are the decimal values) shows how the computer can "expect" different meanings from the number 65 (or 41 hex). When it receives an *instruction* to perform an action, it is then prepared to act *upon* a number. The instruction comes first and, since it is the first thing the computer sees when it starts a job, it *knows that the A9 (169) is not a number.* It has to be one of the ML instructions from its set of instructions (see Appendix A).

Instructions And Their Arguments

The computer would no more think of this first 169 as the *number* 169 than you would seal an envelope before the letter was inside. If you are sending out a pile of Christmas cards, you perform instruction-argument just the way the computer does: you (1) fill the envelope (instruction) (2) with a card (argument or operand). All actions do something *to* something. A computer's action is called an instruction (or, in its numeric form inside the computer's memory it's called an *opcode* for *operation code).* The target of the action is called the instruction's argument (operand). In our program above, the computer must LoaD Accumulator with 65. The # symbol means ''immediate''; the target is right there in the next memory cell following the mnemonic LDA, so it isn't supposed to be fetched from a distant memory cell.

Then the action is complete, and the next number (the 133 which means STore Accumulator in zero page, the first 256 cells) is seen as the start of another complete action. The action of storing always signals that the number following the store instruction must be an address of a cell in memory to store to.

Think of the computer as completing each action and then looking for another instruction. Recall from the last chapter that the target can be ''implied'' in the sense that INX simply increases the X register by one. That ''one'' is ''implied'' by the instruction itself, so there is no target argument in these cases. The next cell in this case *must* also contain an instruction for a new instruction-argument cycle.

Some instructions call for a single-byte argument. LDA #65 is of this type. You cannot LoaD Accumulator with anything greater than 255. The accumulator is only one byte large, so anything that can be loaded into it can also be only a single byte large. Recall that $FF (255 decimal) is the largest number that can be represented by a single byte. STA $65 also has a one byte argument because the target address for the STore Accumulator is, in this case, in zero page. Storing to zero page or loading from it will need only a one byte argument — the address. Zero page addressing is a special case, but an assembler program will take care of it for you. It will pick the correct opcode for this addressing mode when you type LDA $65. LDA $0065 would create ML code that performs the same operation though it would use three bytes instead of two to do it.

The program counter is like a finger that keeps track of where the computer is located in its trip up a series of ML instructions. Each instruction takes up one, two, or three bytes, depending on what type of addressing is going on.

Context Defines Meaning

TXA uses only one byte so the program counter (PC) moves ahead one byte and stops and waits until the value in the X register is moved over to the accumulator. Then the computer asks the PC, ''Where are we?'' and the PC is pointing to the address of the next instruction. It never points to an argument. It skips over them because it knows how many bytes each addressing mode uses up in a program.

Say that the next addresses contain an LDA $15. This is two bytes long (zero page addressing). The PC is raised by two. The longest possible instruction would be using three bytes, such as LDA $5000 (absolute addressing). Here the argument takes up two bytes. Add that to the one byte used by any instruction and you have a total of three bytes for the PC to count off. Zero page LDA is represented by the number A5 and Absolute LDA is AD. Since the opcodes are different, even though the mnemonics are identical, the computer can know how many bytes the instruction will use up.

Having reviewed the way that your computer makes *contextual* sense out of the mass of seemingly similar numbers of which an ML program is composed, we can move on to see how elementary arithmetic is performed in ML.

Addition

Arithmetic is performed in the accumulator. The accumulator holds the first number, the target address holds the second number (but is not affected by the activities), and the result is left in the accumulator. So:

LDA #$40 (remember, the # means immediate, the $ means hex)
ADC #$01

will result in the number 41 being left in the accumulator. We could then STA that number wherever we wanted. Simple enough. The ADC means ADd with Carry. If this addition problem resulted in a number higher than 255 (if we added, say, 250+6), then there would have to be a way to show that the number left behind in the accumulator was not the correct result. What's left behind is the *carry*. What would happen after adding 250+6 is that the accumulator would contain a 1. To show that the answer is really 256 (and not 1), the ''carry flag'' in the status register flips up. So, if that flag is up, we know that the real answer is 255 plus the 1 left in the accumulator.

To make sure that things never get confused, always put in a CLC (CLear Carry) before any addition problems. Then the flag will go down before any addition and, if it is up afterward, we'll know that we need to add 256 to whatever is in the accumulator. We'll know that the accumulator holds the carry, not the total result.

One other point about the status register: there is another flag, the ''decimal'' flag. If you ever set this flag up (SED), all addition and subtraction is performed in a decimal mode in which the carry flag is set when addition exceeds 99. In this book, we are not going into the decimal mode at all, so it's a good precaution to put a CLear Decimal mode (CLD) instruction as the first instruction of any ML program you write. After you type CLD, the flag will be put down and the assembler will move on to ask for your next instruction, but all the arithmetic from then on will be as we are describing it.

Adding Numbers Larger Than 255

We have already discussed the idea of setting aside some memory cells as a table for data. All we do is make a note to ourselves that, say, $80 and $81 are declared a zone for our personal use as a storage area. Using a familiar example, let's think of this zone as the address that holds the address of a ball-like character for a game. As long as the addresses are not in ROM, or used by our program elsewhere, or used by the computer (see your computer's memory map), it's fine to declare any area a data zone. It is a good idea (especially with longer programs) to make notes on a piece of paper to show where you intend to have your subroutines, your main loop, your initialization, and all the miscellaneous data — names, messages for the screen, input from the keyboard, etc. This is one of those things that BASIC does for you automatically, but which you must do for yourself in ML.

When BASIC creates a string variable, it sets aside an area to store variables. This is what DIM does. In ML, you set aside your own areas by simply finding a safe and unused memory space and then not writing a part of your program into it. Part of your data zone can be special registers you declare to hold the results of addition or subtraction. You might make a note to yourself that $80 and $81 will hold the current address of the bouncing ball in your game. Since the ball is constantly in motion, this register will be changing all the time, depending on whether the ball hit a wall, a paddle, etc. Notice that you need *two* bytes for this register. That is because one byte could hold only a number from 0 to 255. Two bytes together, though, can hold a number up to 65535.

In fact, a two-byte register can address *any* cell in most microcomputers because most of us have machines with a total of 65536 memory cells (from zero to 65535). So if your ball is located (on your screen) at $8000 and you must move it down one, just change the ball-address register you have set up. If your screen has 40 columns, you would want to add 40 to this register.

The ball address register now looks like this: $0080 00 80 (remember that the higher, most significant byte, comes *after* the LSB, the least significant byte in the 6502's way of looking at pointers). We want it to be: $0080 28 80. (The 28 is hex for 40.) In other words, we're going to move the ball down one line on a 40-column screen.

Remember the "indirect Y" addressing mode described in the previous chapter? It lets us use an address in *zero page* as a *pointer* to another address in memory. The number in the Y register is added to whatever address sits in 80,81, so we don't STA to $80 or $81, but rather to the address that they *contain*. STA ($80),Y or, using the simplified punctuation rules of the Simple Assembler: STA (80)Y.

Moving A Ball Down

How to add $28 to the ball address register? First of all, CLC, clear the carry to be sure that flag is down. To simplify our addition, we can set aside another special register which serves only to hold the $28 as a double-byte number all through the game: $4009 28 00. This is the size of one screen line in our 40-column computer and it won't change. Since it moves the ball down one screen line, it can be used equally well for a subtraction that would move the ball up one screen line as well. Now to add them together:

```
1000 CLC            (1000 is our "add 40 to ball address"
                     subroutine)
1001 LDA  $80       (we fetch the LSB of ball address)
1003 ADC  $4009     (LSB of our permanent screen line size)
1006 STA  $80       (put the new result into the ball address)
1008 LDA  $81       (get the MSB of ball address)
100A ADC  $400A     (add with carry to the MSB of screen value)
100D STA  $81       (update the ball address MSB)
```

That's it. Any carry will automatically set the carry flag up during the ADC action on the LSB and will be added into the result when we ADC to the MSB. It's all quite similar to the way that we add ordinary decimal numbers, putting a carry onto the next column when we get more than a 10 in the first column. And this carrying is why we always CLC (clear the carry flag, putting it down) just before additions. If the carry is set, we could get the wrong answer if our problem did not result in a carry. Did the addition above cause a carry?

Note that we need not check for any carries during the MSB+MSB addition. Any carries resulting in a screen address greater than $FFFF (65535) would be impossible on our machines. The 6502 is permitted to address $FFFF tops, under normal conditions.

Subtraction

As you might expect, subtracting single-byte numbers is a snap:

```
LDA #$41
SBC #$01
```

results in a $40 being left in the accumulator. As before, though, it is good to make it a habit to deal with the carry flag before each calculation. When subtracting, however, you *set* the carry flag: SEC. Why is unimportant. Just always SEC before any subtractions, and your answers will be correct. Here's double subtracting that will move the ball up the screen one line instead of down one line:

```
$1020 SEC           ($1020 is our "take 40 from ball address"
                     subroutine)
1021  LDA $80       (get the LSB of ball address)
```

```
1023  SBC $4009 (LSB of our permanent screen line value)
1026  STA $80   (put the new result into the ball address)
1028  LDA $81   (get the MSB of ball address)
102A  SBC $400A (subtract the MSB of screen value)
102D  STA $81   (update the ball address MSB)
```

Multiplication And Division

Multiplying could be done by repeated adding. To multiply 5×4, you could just add 4+4+4+4+4. One way would be to set up two registers like the ones we used above, both containing 04, and then loop through the addition process five times. For practical purposes, though, multiplying and dividing are much more easily accomplished in BASIC. They simply are often not worth the trouble of setting up in ML, especially if you will need results involving decimal points (floating point arithmetic). Perhaps surprisingly, for the games and personal computing tasks where creating ML routines is useful, there is little use either for negative numbers or arithmetic beyond simple addition and subtraction.

If you find that you need complicated mathematical structures, create the program in BASIC, adding ML where super speeds are necessary or desirable. Such hybrid programs are efficient and, in their way, elegant. One final note: an easy way to divide the number in the accumulator by two is to LSR it. Try it. Similarly, you can multiply by two with ASL. We'll define LSR and ASL in the next chapter.

Double Comparison

One rather tricky technique is used fairly often in ML and should be learned. It is tricky because there are two branch instructions which *seem* to be worth using in this context, but they are best avoided. If you are trying to keep track of the location of a ball on the screen, it will have a two-byte address. If you need to compare those two bytes against another two-byte address, you need a "double compare" subroutine. You might have to see if the ball is out of bounds or if there has been a collision with some other item flying around on screen. Double compare is also valuable in other kinds of ML programming.

The problem is the BPL (Branch on PLus) and BMI (Branch on MInus) instructions. *Don't use them* for comparisons. In any comparisons, single- or double-byte, use BEQ to test if two numbers are equal; BNE for not equal; BCS for equal or higher; and BCC for lower. You can remember BCS because its "S" is *higher* and BCC because its "C" is *lower* in the alphabet. To see how to perform a double-compare, here's one easy way to do it. (See Program 5-1.)

Program 5-1. Double Compare.

```
                      .BA $1010
               ; ------- STORAGE AREAS -------
               TESTED   .DE $1000
               SECOND   .DE $1002
               TEMP     .DE $1008
               ;
               ; ------- LANDING PLACES ------
               LOWER    .DE $1004
               EQUAL    .DE $1005
               HIGHER   .DE $1006
               ;
1010- 38       START    SEC
1011- AD 00 10          LDA TESTED     ; COMPARE THE LOW BYTES
1014- ED 02 10          SBC SECOND
1017- 8D 08 10          STA TEMP
101A- AD 01 10          LDA TESTED+1   ; COMPARE THE HIGH BYTES
101D- ED 03 10          SBC SECOND+1
1020- 0D 08 10          ORA TEMP
1023- F0 E0             BEQ EQUAL      ; TESTED = SECOND
1025- 90 DD             BCC LOWER      ; TESTED < SECOND
1027- B0 DD             BCS HIGHER     ; TESTED > SECOND
                        .EN
```

This is a full-dress, luxurious assembler at work. With such assemblers you can use line numbers and labels, add numbers to labels (see TESTED +1 in line 150), add comments, and all the rest. To try this out, type in the hex bytes on the left, starting at address $1010, which make up the program itself. Then fill bytes $1000-100f with zeros — that's your storage area for the numbers you are comparing as well as a simulated "landing place" where your computer will branch, demonstrating that the comparison worked correctly.

Now try putting different numbers into the two-byte zones called TESTED and SECOND. TESTED, at $1000, is the first, the tested, number. It's being *tested against* the second number, called SECOND. As you can see, you've got to keep it straight in your mind which number is the primary number. There has to be a way to tag them so that it means something when you say that one is larger (or smaller) than the other.

When you've set up the numbers in their registers ($1000 to $1003), you can run this routine by starting at $1010. All that will happen is that you will land on a BRK instruction. *Where* you land tells you the result of the comparison. If the numbers are equal, you land at $1005. If the TESTED number is less than the SECOND number, you'll end up at $1004. If all you needed to find out was whether they were unequal, you could use BNE. Or you could leave out branches that you weren't interested in. Play around with this routine until you've understood the ideas involved.

In a real program, of course, you would be branching to the addresses of subroutines which do something if the numbers are equal or greater or whatever. This example sends the computer to $1004, $1005, or $1006 just to let you see the effects of the double-compare subroutine. Above all, remember that comparing in ML is done with BCS and BCC (*not* BPL or BMI).

6

The Instruction Set

There are 56 instructions (commands) available in 6502 machine language. Most versions of BASIC have about 50 commands. Some BASIC instructions are rarely used by the majority of programmers: USR, END, SGN, TAN, etc. Some, such as END and LET, contribute nothing to a program and seem to have remained in the language for nostalgic reasons. Others, like TAN, have uses that are highly specialized. There are surplus commands in computer languages just as there are surplus words in English. People don't often say *culpability.* They usually say *guilt.* The message gets across without using the entire dictionary. The simple, common words can do the job.

Machine language is the same as any other language in this respect. There are around 20 heavily used instructions. The 36 remaining ones are far less often used. Load the disassembler program in Appendix D and enter the starting address of your computer's BASIC in ROM. You can then read the machine language routines which comprise it. You will quickly discover that the accumulator is heavily trafficked (LDA and STA appear frequently), but you will have to hunt to find an ROR, SED, CLV, RTI, or BVC.

ML, like BASIC, offers you many ways to accomplish a given job. Some programming solutions, of course, are better than others, but the main thing is to get the job done. An influence still lingers from the early days of computing when memory space was rare and expensive. This influence — that you should try to write programs using up as little memory as possible — is usually safely ignored. Efficient memory use will often be low on your list of objectives. It could hardly matter if you used up 25 instead of 15 bytes to print a message to your screen when your computer has space for programs which exceeds 30,000 bytes.

Rather than memorize each instruction individually, we will concentrate on the workhorses. Bizarre or arcane instructions will get only passing mention. Unless you are planning to work with ML for interfacing or complex mathematics and such, you will be able to write excellent machine language programs for nearly any application with the instructions we'll focus on here.

For each instruction group, we will describe three things before getting down to the details about programming with them. 1. What the instructions accomplish. 2. The addressing modes you can use with them. 3. What they do, if anything, to the flags in the Status Register. All of this information is also found in Appendix A.

The Six Instruction Groups

The best way to approach the ''instruction set'' might be to break it down into the following six categories which group the instructions according to their functions: 1. The Transporters 2. The Arithmetic Group 3. The Decision-makers 4. The Loop Group 5. The Subroutine and Jump Group and 6. The Debuggers. We will deal with each group in order, pointing out similarities to BASIC and describing the major uses for each.

As always, the best way to learn is by doing. Move bytes around. Use each instruction, typing a BRK as the final instruction to see the effects. If you LDA #65, look in the A register to see what happened. Then STA $12 and check to see what was copied into address $12. If you send the byte in the accumulator (STA), what's left behind in the accumulator? Is it better to think of bytes as being *copied* rather than being *sent?*

Play with each instruction to get a feel for it. Discover the effects, qualities, and limitations of these ML commands.

1. The Transporters: LDA, LDX, LDY STA, STX, STY TAX, TAY TXA, TYA

These instructions move a byte from one place in memory to another. To be more precise, they *copy* what is in a source location into a target location. The source location still contains the byte, but after a ''transporter'' instruction, a copy of the byte is also in the target. This does replace whatever was in the target.

All of them affect the N and Z flags, except STA, STX, and STY which do nothing to any flag.

There are a variety of addressing modes available to different instructions in this group. Check the chart in Appendix A for specifics.

Remember that the computer does things *one at a time.* Unlike the human brain which can carry out up to 1000 different instructions simultaneously (walk, talk, and smile, all at once) — the computer goes from one tiny job to the next. It works through a series of instructions, raising the *program counter* (PC) each time it handles an instruction.

If you do a TYA, the PC goes up by one to the next address and the computer looks at that next instruction. STA $80 is a two-byte long instruction, it's zero page addressing, so the PC=PC+2. STA $8500 is a three-byte long absolute addressing mode and PC=PC+3.

Recall that there's nothing larger than a three-byte increment of the PC. However, in each case, the PC is cranked up the right amount to make it point to the address for the next instruction. Things would get quickly out of control if the PC pointed to some argument, thinking it was an instruction. It would be incorrect (and soon disastrous) if the PC landed on the $15 in LDA $15.

If you type SYS 1024 (or USR or CALL), the program counter is loaded with $0400 and the computer ''transfers control'' to the ML instructions which are (we hope!) waiting there. It will then look at the byte in $0400, expecting it to be an ML instruction. It will do that job and then look for the next instruction. Since it does this very fast, it can seem to be keeping score, bouncing the ball, moving the paddle, and everything else — simultaneously. It's not, though. It's flashing from one task to another and doing it so fast that it creates the illusion of simultaneity much the way that 24 still pictures per second look like motion in movies.

The Programmer's Time Warp

Movies are, of course, lots of still pictures flipping by in rapid succession. Computer programs are composed of lots of individual instructions performed in rapid succession.

Grasping this sequential, step-by-step activity makes our programming job easier: we can think of large programs as single steps, coordinated into meaningful, harmonious actions. Now the computer will put a blank over the ball at its current address, then add 40 to the ball's address, then print a ball at the new address. The main single-step action is moving information, as single-byte numbers, from here to there, in memory. We are always creating, updating, modifying, moving and destroying single-byte variables. The moving is generally done from one double-byte address to another. But it all looks smooth to the player during a game.

Programming in ML can pull you into an eerie time warp. You might spend several hours constructing a program which executes in seconds. You are putting together instructions which will later be read and acted upon by coordinated electrons, moving at electron speeds. It's as if you spent an afternoon slowly and carefully drawing up pathways and patterns which would later be a single bolt of lightning.

Registers

In ML there are three primary places where variables rest briefly on their way to memory cells: the X, the Y, and the A registers. And the A register (the *accumulator*) is the most frequently used. X and Y are used for looping and indexing. Each of these registers can grab a byte from anywhere in memory or can load the byte right after its own opcode (immediate addressing):

LDX $8000 (puts the number at hex address 8000 into X, without destroying it at $8000)
LDX #65 (puts *the number 65* into X)
LDA and **LDY** work the same.

Be sure you understand what is happening here. LDX $1500 does not copy the ''byte in the X register into address $1500.'' It's just the opposite. The number (or ''value'' as it's sometimes called) in $1500 is copied into the X register.

To copy a byte from X, Y, or A, use STX, STY, or STA. For these ''store-bytes'' instructions, however, there is no immediate addressing mode. No STA #15. It would make no sense to have STA #15. That would be disruptive, for it would *modify the ML program itself.* It would put the number 15 into the next cell beyond the STA instruction *within* the ML program itself.

Another type of transporter moves bytes *between* registers — TAY, TAX, TYA, TXA. See the effect of writing the following. Look at the registers after executing this:

```
1000 LDA #65
     TAY
     TAX
```

The number 65 is placed into the accumulator, then transferred to the Y register, then sent from the accumulator to X. All the while, however, the A register (accumulator) is *not* being emptied. Sending bytes is not a ''transfer'' in the usual sense of the term ''sending.'' It's more as if a Xerox copy were made of the number and then the copy is sent. The original stays behind after the copy is sent.

LDA #15 followed by TAY would leave the 15 in the accumulator, sending a copy of 15 into the Y register.

Notice that you cannot directly move a byte from the X to the Y register, or vice versa. There is no TXY or TYX.

Flags Up And Down

Another effect of moving bytes around is that it sometimes throws a flag up or down in the Status Register. LDA (or LDX or LDY) will affect the N and Z, negative and zero, flags.

We will ignore the N flag. It changes when you use ''signed numbers,'' a special technique to allow for negative numbers. For our purposes, the N flag will fly up and down all the time and we won't care. If you're curious, signed numbers are manipulated by allowing the seven bits on the right to hold the number and the leftmost bit stands for positive or negative. We normally use a byte to hold values from 0 through 255. If we were working with "signed" numbers, anything higher than 127 would be considered a negative number since the leftmost bit would be "on" — and an LDA #255 would be thought of as -1. This is another example of how the same things (the number 255 in this case) could signify several different things, depending on the context in which it is being interpreted.

The Z flag, on the other hand, is quite important. It shows whether or not some action during a program run resulted in a zero. The branching instructions and looping depend on this flag, and we'll deal with the important zero-result effects below with the BNE, INX, etc., instructions.

No flags are affected by the STA, STX, or STY instructions.

The Stack Can Take Care Of Itself

There are some instructions which move bytes to and from the stack. These are for advanced ML programmers. PHA and PLA copy a byte from A to the stack, and vice versa. PHP and PLP move the status register to and from the stack. TSX and TXS move the stack pointer to or from the X register. Forget them. Unless you know precisely what you are doing, you can cause havoc with your program by fooling with the stack. The main job for the stack is to keep the return addresses pushed into it when you JSR (Jump To Subroutine). Then, when you come back from a subroutine (RTS), the computer pulls the addresses off the stack to find out where to go back to.

The one major exception to this warning about fiddling with the stack is Atari's USR instruction. It is a worthwhile technique to master. Atari owners can move between BASIC and ML programs fairly easily, passing numbers to ML via the stack. The parameters (the passed numbers) must be pulled off the stack when the ML program first takes control of the computer.

For most ML programming, on the other hand, avoid stack manipulation until you are an advanced programmer. If you manipulate the stack without great care, you'll give an RTS the wrong address and the computer will travel far, far beyond your control. If you are lucky, it sometimes lands on a BRK instruction and you fall into the monitor mode. The odds are that you would get lucky roughly once every 256 times. Don't count on it. Since BRK is rare in your BASIC ROM, the chances are pretty low. If your monitor has a FILL instruction which lets you put a single number into large amounts of RAM memory, you might want to fill the RAM with "snow." FILL 1000 8000 00 would put zeros into every address from 1000 to 8000. This greatly improves the odds that a crash *will* hit a BRK.

As an aside, there is another use for a blanket of "zero page snow." Many Atari programs rely on the fact that the computer leaves page six ($0600-06FF) pretty much alone. The PET doesn't make much use of the second cassette buffer. So, you can safely put an ML subroutine in these places to, for example, add a routine which customizes an ML word processor. Does your Atari's ML word-processing program use any memory space in page six? Probably. What locations does it use? Fill page six with 00's, put the word-processor through its paces, then look at the tracks, the non-zeros, in the snow.

2. The Arithmetic Group: ADC, SBC, SEC, CLC

Here are the commands which add, subtract, and set or clear the carry flag. ADC and SBC affect the N, Z, C, and V (overflow) flags. CLC and SEC, needless to say, affect the C flag and their only addressing mode is Implied.

ADC and SBC can be used in eight addressing modes: Immediate, Absolute, Zero Page, (Indirect,X), (Indirect),Y, Zero Page,X, and Absolute,X and Y.

Arithmetic was covered in the previous chapter. To review, before any addition, the carry flag must be cleared with CLC. Before any subtraction, it must be set with SEC. The decimal mode should be cleared at the start of any program (the initialization): CLD. You can multiply by two with ASL and divide by two with LSR. Note that you can divide by four with LSR LSR or by eight with LSR LSR LSR. You could multiply a number by eight with ASL ASL ASL. What would this do to a number: ASL ASL ASL ASL? To multiply by numbers which aren't powers of two, use addition plus multiplication. To multiply by ten, for example: copy the original number temporarily to a vacant area of memory. Then ASL ASL ASL to multiply it by eight. Then multiply the stored original by two with a single ASL. Then add them together.

If you're wondering about the V flag, it is rarely used for anything. You can forget about the branch which depends on it, BVC, too. Only five instructions affect it and it relates to "twos complement" arithmetic which we have not touched on in this book. Like decimal mode or negative numbers, you will be able to construct your ML programs very effectively if you remain in complete ignorance of this mode. We have largely avoided discussion of most of the flags in the status register: N, V, B, D, and I. This avoidance has also removed several branch instructions from our consideration: BMI, BPL, BVC, and BVS. These flags and instructions are not usually found in standard ML programs and their use is confined to specialized mathematical or interfacing applications. They will not be of use or interest to the majority of ML programmers.

The two flags of interest to most ML programmers are the Carry flag and the Zero flag. That is why, in the following section, we will examine only the four branch instructions which test the C and Z flags. They are likely to be the only branching instructions that you'll ever find occasion to use.

3. The Decision-Makers: CMP, BNE, BEQ, BCC, BCS

The four "branchers" here — they all begin with a "B" — have only one addressing mode. In fact, it's an interesting mode unique to the "B" instructions and created especially for them: *relative* addressing. They do not address a memory location as an *absolute* thing; rather, they address a location which is a certain distance from their position in the ML code. Put another way, the argument of the "B" instructions is an offset which is *relative* to their position. You never have to worry about relocating "B" instructions to another part of memory. You can copy them and they will work just as well in the new location. That's because their argument just says "add five to the present address" or "subtract twenty-seven," or whatever argument you give them. But they can't branch further back than 127 or further forward than 128 bytes.

None of the brancher instructions have any effect whatsoever on any flags; instead, they are the instructions which *look at* the flags. They are the only instructions that base their activity on the condition of the status register and its flags. They are why the flags exist at all.

CMP is an exception. Many times it is the instruction that comes just before the branchers and sets flags for them to look at and make decisions about. Lots of instructions — LDA is one — will set or "clear" (put down) flags — but sometimes you need to use CMP to find out what's going on with the flags. CMP affects the N, Z, and C flags. CMP has many addressing modes available to it: Immediate, Absolute, Zero Page, (Indirect,X), (Indirect),Y, Zero Page,X, and Absolute,X and Y.

The Foundations Of Computer Power

This decision-maker group and the following group (loops) are the basis of our computers' enormous strength. The decision-makers allow the computer to decide among two or more possible courses of action. This decision is based on comparisons. *If* the ball hits a wall, *then* reverse its direction. In BASIC, we use IF-THEN and ON-GOTO structures to make decisions and to make appropriate responses to conditions as they arise during a program run.

Recall that most micros use *memory mapped video,* which means that you can treat the screen like an area of RAM memory. You can PEEK and POKE into it and create animation, text, or other visual events. In ML, you PEEK by LDA $VIDEO MEMORY and examine what you've PEEKed with CMP. You POKE via STA $VIDEO MEMORY.

CMP does comparisons. This tests the value at an address against what is in the accumulator. Less common are CPX and CPY. Assume that we have just added 40 to a register we set aside to hold the current address-location of a ball on our screen during a game. Before the ball can be POKEd into that address, we'd better make sure that something else (a wall, a paddle, etc.) is not sitting there. Otherwise the ball would pass right through walls.

Since we just increased the location register (this register, we said, was to be at $80,81), we can use it to find out if there is blank space (32) or something else (like a wall). Recall that the very useful "indirect Y" addressing mode allows us to use an address in zero page as a *pointer* to another address in memory. The number in the Y register is added to whatever address sits in 80,81; so we don't LDA from 80 or 81, but rather from the address that they *contain,* plus Y's value.

To see what's in our potential ball location, we can do the following:

LDY #0 (we want to fetch from the ball address itself, so we don't want to add anything to it. Y is set to zero.)

LDA (80),Y (fetch whatever is sitting where we plan to next send the ball. To review Indirect, Y addressing once more: say that the address we are fetching from here is $1077. Address $80 would hold the LSB ($77) and address $81 would hold the MSB ($10). Notice that the argument of an Indirect,Y instruction only mentions the lower address of the two-byte pointer, the $80. The computer knows that it has to combine $80 and $81 to get the full address — and does this automatically.)

At this point in your game, there might be a 32 (ASCII for the space or blank character) or some other number which we would know indicated a wall, another player, a paddle, etc. Now that this questionable number sits in the accumulator, we will CMP it against a space. We could compare it with the number which means wall or the other possibilities — it doesn't matter. The main thing is to compare it:

2000 CMP #32	(is it a space?)
2002 BNE 200A	(Branch if Not Equal [if not 32] to address 200A, which contains the first of a series of comparisons to see if it's a wall, a paddle, etc. On the other hand, if the comparison *worked*, if it was a 32 (so we didn't Branch Not Equal), then the next thing that happens is the instruction in address 2004. We ''fall through'' the BNE to an instruction which jumps to the subroutine (JSR), which moves the ball into this space and then returns to address 2007, which jumps over the series of comparisons for wall, paddle, etc.)
2004 JSR 3000	(the ball printing subroutine)
2007 JMP 2020	(jump over the rest of the comparisons)
200A CMP #128	(is it our paddle symbol?)
200C BNE 2014	(if not, continue to next comparison)
200E JSR 3050	(do the paddle-handling subroutine and . . .)
2011 JMP 2020	(jump over the rest, as before in 2007)
2014 CMP #144	(is it a wall . . . and so forth with as many comparisons as needed)

This structure is to ML what ON-GOTO or ON-GOSUB is to BASIC. It allows you to take multiple actions based on a single LDA. Doing the CMP only once would be comparable to BASIC's IF-THEN.

Other Branching Instructions

In addition to the BNE we just looked at, there are BCC, BCS, BEQ, BMI, BPL, BVC, and BVS. Learn BCC, BCS, BEQ, and BNE and you can safely ignore the others.

All of them are branching, IF-THEN, instructions. They work in the same way that BNE does. You write BEQ followed by the address you want to go to. If the result of the comparison is ''yes, equal-to-zero is true,'' then the ML program will jump to the address which is the argument of the BEQ. ''True'' here means that something EQuals zero. One example that would send up the Z flag (thereby triggering the BEQ) is: LDA #00. The action of loading a zero into A sets the Z flag up.

You are allowed to ''branch'' either forward or backward from the address that holds the ''B—'' instruction. However, you cannot branch any further than 128 bytes in either direction. If you want to go further, you must JMP (JuMP) or JSR (Jump to SubRoutine). For all practical purposes, you will usually be branching to instructions located within 30 bytes of your ''B'' instruction in either direction. You will be taking care of most things right near where a CoMPare, or other flag-setting event, takes place.

If you need to use an elaborate subroutine, simply JSR to it at the target address of your branch:

```
2000 LDA 65
2002 CMP 85   (is what was in address 65 equal to what was in
               address 85?)
2004 BNE 2009 (if Not Equal, branch over the next three bytes
               which perform some elaborate job)
2006 JSR 4000 (at 4000 sits an elaborate subroutine to take care
               of cases where addresses 65 and 85 turn out to
               be equal)
2009          (continue with the program here)
```

If you are branching backwards, you've written that part of your program, so you know the address to type in after a BNE or one of the other branches. But, if you are branching forward, to an address in part of the program not yet written — how do you know what to give as the address to branch to? In complicated *two-pass assemblers,* you can just use a word like "BRANCHTARGET", and the assembler will "pass" twice through your program when it assembles it. The first "pass" simply notes that your BNE is supposed to branch to "BRANCHTARGET," but it doesn't yet know where that is.

When it finally finds the actual address of "BRANCHTARGET," it makes a note of the correct address in a special *label table.* Then, it makes a second "pass" through the program and fills in (as the next byte after your BNE or whatever) the correct address of "BRANCHTARGET". All of this is automatic, and the labels make the program you write (called the *source code*) look almost like English. In fact, complicated assemblers can contain so many special features that they can get close to the *higher-level* languages, such as BASIC:

```
(These initial definitions of labels   TESTBYTE =80
 are sometimes called "equates.")      NEWBYTE=99
                           2004 LDA TESTBYTE
                           2006 CMP NEWBYTE
                           2008 BNE BRANCHTARGET
                           200A JR  SPECIALSUBROUTINE
          BRANCHTARGET 200D . . . etc.
```

Instead of using lots of numbers (as we do when using the Simple Assembler) for the target/argument of each instruction, these assemblers allow you to *define* ("equate") the meanings of words like "TESTBYTE" and from then on you can use the word instead of the number. And they do somewhat simplify the problem of forward branching since you just give (as above) address 200D a name, "BRANCHTARGET," and the word at address 2009 is later replaced with 200D when the assembler does its passes.

This is how the example above looks as the source code listing from a two-pass, deluxe assembler:

Program 6-1.

```
                    0010                   .BA $2004
                    0020 TESTBYTE          .DE $80
                    0030 NEWBYTE           .DE $99
                    0040 ;
2004- A9 80         0050 START             LDA #TESTBYTE        ; (IMMEDIATE ADDRE
2006- C5 99         0060                   CMP *NEWBYTE  ; (ZERO PAGE ADDRESSING)
2008- D0 03         0070                   BNE BRANCHTARGET     ; (RELATIVE ADDRES
200A- 20 10 20      0080                   JSR SPECIALSUBROUTINE
200D- AD 00 04      0090 BRANCHTARGET      LDA $400       ; YOU CAN FREELY MIX
                    0100 ; LABLES AND SUBROUTINES.  ALSO, COMMENTS
                    0110 ; WILL BE IGNORED BY THE ASSEMBLER AND CAN
                    0120 ; BE STUCK ANYWHERE, AS YOU SEE.
                    0130 ;
2010- AD 21 00      0140 SPECIALSUBROUTINE LDA 33
                    0150 ; ETC. ETC.
                    0160                   .EN
```

Actually, we should note in passing that a 200D will not be the number which finally appears at address 2009 to replace "BRANCHTARGET". To save space, all branches are indicated as an "offset" from the address of the branch. The number which will finally replace "BRANCHTARGET" at 2009 above will be three. This is similar to the way that the value of the Y register is *added* to an address in zero page during indirect Y addressing (also called "indirect indexed"). The number given as an argument of a branch instruction is *added* to the address of the next *instruction*. So, 200A + 3 = 200D. Our Simple Assembler will take care of all this for you. All you need do is give it the 200D and it will compute and put the 3 in place for you.

Forward Branch Solutions

There is one responsibility that you do have, though. When you are writing 2008 BNE 200D, how do you know to write in 200D? You can't yet know to exactly which address up ahead you want to branch. There are two ways to deal with this. Perhaps easiest is to just put in BNE 2008 (have it branch to itself). This will result in a FE being temporarily left as the target of your BNE. Then, you can make a note on paper to later change the byte at 2009 to point to the correct address, 200D. You've got to remember to "resolve" that FE to POKE in the number to the target address, or you will leave a little bomb in your program — an endless loop. The Simple Assembler has a POKE function. When you type POKE, you will be asked for the address and value you want POKEd. So, by the time you have finished coding 200D, you could just type POKE and then POKE 2009,3.

The other, even simpler, way to deal with forward branch addresses will come after you are familiar with which instructions use one, two, or three bytes. This BNE-JSR-TARGET construction is common and will always be six away from the present address, an *offset* of 6. If the branch instruction is at 2008, you just count off three: 200A, 200B, 200C and write BNE 200D. Other, more complex branches such as ON-GOTO constructions will also become easy to count off when you're familiar with the instruction byte-lengths. In any case, it's simple enough to make a note of any unsolved branches and correct them before running the program.

Alternatively, you can use a single "unresolved" forward branch in the Simple Assembler; see its instructions. You just type BNE FORWARD.

Recall our previous warning about staying away from the infamous BPL and BMI instructions? BPL (Branch on PLus) and BMI (Branch on MInus) sound good, but should be avoided. To test for less-than or more-than situations, use BCC and BCS respectively. (Recall that BCC is alphabetically *less-than* BCS — an easy way to remember which to use.) The reasons for this are exotic. We don't need to go into them. Just be warned that BPL and BMI, which sound so logical and useful, are not. They can fail you and neither one lives up to its name. Stick with the always trustworthy BCC, BCS.

Also remember that BNE and the other three main "B" group branching instructions often don't need to have a CMP come in front of them to set a flag they can test. Many actions of many opcodes will automatically set flags during their operations. For example, LDA $80 will affect the Z flag so you can tell if the number in address $80 was or wasn't zero by that flag. LDA $80 followed by BNE would branch away if there were anything besides a zero in address $80. If in doubt, check the chart of instructions in Appendix A to see which flags are set by which instructions. You'll soon get to know the common ones. If you are really in doubt, go ahead and use CMP.

4. The Loop Group: DEY, DEX, INY, INX, INC, DEC

INY and INX raise the Y and X register values *by one* each time they are used. If Y is a 17 and you INY, Y becomes an 18. Likewise, DEY and DEX decrease the value in these registers by one. There is no such increment or decrement instruction for the accumulator.

Similarly, INC and DEC will raise or lower a memory address by one. You can give arguments to them in four addressing modes: Absolute, Zero Page, Zero Page,X and Absolute,X. These instructions affect the N and Z flags.

The Loop Group are usually used to set up FOR-NEXT structures. The X register is used most often as a counter to allow a certain number of events to take place. In the structure FOR I = 1 TO 10: NEXT I, the value of the variable *I* goes up by one each time the loop cycles around. The same effect is created by:

```
2000 LDX #10
2002 DEX        ("DEcrement" or "DEcrease X" by 1)
2003 BNE 2002   (Branch if Not Equal [to zero] back up to
                 address 2002)
```

Notice that DEX is tested by BNE (which sees if the Z flag, the zero flag, is up). DEX sets the Z flag up when X finally gets down to zero after ten cycles of this loop. (The only other flag affected by this loop group is the N [negative] flag for signed arithmetic.)

Why didn't we use INX, INcrease X by 1? This would parallel exactly the FOR I = 1 TO 10, but it would be clumsy since our starting count which is #10 above would have to be #245. This is because X will not become a zero *going up* until it hits 255. So, for clarity and simplicity, it is customary to set the count of X and then DEX it downward to zero. The following program will accomplish the same thing as the one above, and allow us to INX, but it too is somewhat clumsy:

```
2000 LDX  #0
2002 INX
2003 CPX  #10
2005 BNE 2002
```

Here we had to use zero to start the loop because, right off the bat, the number in X is INXed to one by the instruction at 2002. In any case, it is a good idea to just memorize the simple loop structure in the first example. It is easy and obvious and works very well.

Big Loops

How would you create a loop which has to be larger than 256 cycles? When we examined the technique for adding large numbers, we simply used two-byte units instead of single-byte units to hold our information. Likewise, to do large loops, you can count down in two bytes, rather than one. In fact, this is quite similar to the idea of "nested" loops (loops within loops) in BASIC.

```
2000 LDX  #10  (start of 1st loop)
2002 LDY  #0   (start of 2nd loop)
2004 DEY
2005 BNE  2004 (if Y isn't yet zero, loop back to DEcrease Y
                again — this is the inner loop)
2007 DEX       (reduce the outer loop by one)
2008 BNE 2002  (if X isn't yet zero, go through the entire DEY
                loop again)
200A           (continue with the rest of the program . . . )
```

One thing to watch out for: be sure that a loop BNE's back up to *one address after* the start of its loop. The start of the loop sets a number into a register and, if you keep looping up to it, you'll always be putting the same number into it. The DEcrement (decrease by one) instruction would then never bring it down to zero to end the looping. You'll have created an endless loop.

The example above could be used for a "timing loop" similarly to the way that BASIC creates delays with: FOR T=1 TO 2000: NEXT T. Also, sometimes you *do* want to create an endless loop (the BEGIN . . . UNTIL in "structured programming"). A popular "endless" loop structure in BASIC waits until the user hits any key: 10 GET K$: IF K$="" THEN 10.

10 IF PEEK (764)=255 THEN 10 is the way to accomplish this on the Atari; it will cycle endlessly unless a key is pressed. The simplest way to accomplish this in ML is to look on the map of your computer

to find which byte holds the "last key pressed" number. On Upgrade and 4.0 CBM/PET, it's address 151. On Atari, it's 764. On Apple II, it's -16384. On VIC and Commodore 64, it's 203 with a 64 in that location if no key is pressed. In any event, when a key is pressed, it deposits its special numerical value into this cell. If no key is pressed, some standard value stays there all the time. We'll use the CBM as our model here. If no key is pressed, location 151 will hold a 255:

```
2000 LDA 151
2002 CMP #255
2004 BEQ 2000
```

If the CMP is EQual, this means that the LDA pulled a 255 out of address 151 and, thus, no key is pressed. So, we keep looping until the value of address 151 is something other than 255. This setup is like GET in BASIC because not only does it wait until a key is pressed, but it also leaves the value of the key in the accumulator when it's finished.

Recall that a CMP performs a *subtraction.* It subtracts the number in its argument from whatever number sits in the accumulator at the time. LDA #12 CMP $15 would subtract a 5 from 12 if 5 is the number "held" in address 15. This is how it can leave flags set for testing by BEQ or BNE. The key difference between this "subtraction" and SBC is that neither the accumulator nor the argument is affected at all by it. They stay what they were. The result of the subtraction is "thrown away," and all that happens is that the status flags go up or down in response to the result. If the CMP subtraction causes an answer of zero, the Z flag flips up. If the answer is not zero, the Z flag flips down. Then, BNE or BEQ can do their job — checking flags.

Dealing With Strings

You've probably been wondering how ML handles strings. It's pretty straightforward. There are essentially two ways: known-length and zero-delimit. If you know how many characters there are in a message, you can store this number at the very start of the text: "5ERROR." (The number 5 will fit into one byte, at the start of the text of the message.) If this little message is stored in your "message zone" — some arbitrary area of free memory you've set aside to hold all of your messages — you would make a note of the particular address of the "ERROR" message. Say it's stored at 4070. To print it out, you have to know where you "are" on your screen (cursor position). Usually, the cursor address is held in two bytes in zero page so you can use Indirect,Y addressing.

Alternatively, you could simply set up your own zero-page pointers to the screen. For Apple II and Commodore 64, the screen memory starts at 1024; for CBM/PET it's 32768. In any case, you'll be able to set up a "cursor management" system for yourself. To simplify, we'll send our message to the beginning of the Apple's screen:

2000 LDX 4070	(remember, we put the length of the message as the first byte of the message, so we load our counter with the length)
2003 LDY #0	(Y will be our message offset)
2005 LDA 4071,Y	(gets the character at the address plus Y. Y is zero the first time through the loop, so the "e" from here lands in the accumulator. It also stays in 4071. It's just being copied into the accumulator.)
2008 STA 1024,Y	(we can make Y do double duty as the offset for both the stored message and the screen-printout. Y is still zero the first time through this loop, so the "e" goes to 1024.)
2011 INY	(prepare to add one to the message-storage location and to the screen-print location)
2012 DEX	(lower the counter by one)
2013 BNE 2005	(if X isn't used up yet, go back and get-and-print the next character, the "r")

If The Length Is Not Known

The alternative to knowing the length of a string is to put a special character (usually zero) at the end of each message to show its limit. This is called a *delimiter.* Note that Atari users cannot make zero the delimiter because zero is used to represent the space character. A zero works well for other computers because, in ASCII, the value 0 has no character or function (such as carriage return) coded to it. Consequently, any time the computer loads a zero into the accumulator (which will flip up the Z flag), it will then know that it is at the end of your message. At 4070, we might have a couple of error messages: "Ball out of range0Time nearly up!0". (These are numeric, not ASCII, zeros. ASCII zero has a value of 48.)

To print the time warning message to the top of the CBM/PET screen (this is in decimal):

2000 LDY #0	
2002 LDA 4088,Y	(get the "T")
2005 BEQ 2005	(the LDA just above will flip the zero flag up if it loads a zero, so we *forward branch* out of our message-printing loop. "BEQ 2005" is a dummy target, used until we know the actual target and can POKE it into 2006.)
2007 STA 32768,Y	(we're using the Y as a double-duty offset again)

```
2010 INY
2011 JMP 2002 (in this loop, we always jump back. Our exit
               from the loop is not here, at the end. Rather, it
               is the Branch if EQual which is within the
               loop.)
2014          (continue with another part of the program)
```

By the way, you should notice that the Simple Assembler will reject the commas in this example and, if you've forgotten to set line 10 to accept decimal, it will not accept the single zero in LDY #0. Also, if you get unpredictable results, maybe decimal 2000 is not a safe address to store your ML. You might need to use some other practice area.

Now that we know the address which follows the loop (2014), we can POKE that address into the "false forward branch" we left in address 2006. What number do we POKE into 2006? Just subtract 2007 from 2014, which is seven. Using the Simple Assembler, type POKE and you can take care of this while you remember it. The assembler will perform the POKE and then return to wait for your next instruction.

Both of these ways of handling messages are effective, but you must make a list on paper of the starting addresses of each message. In ML, you have the responsibility for some of the tasks that BASIC (at an expense of speed) does for you. Also, no message can be larger than 255 using the methods above because the offset and counter registers count only that high before starting over at zero again. Printing two strings back-to-back gives a longer, but still under 255 byte, message:

```
2000 LDY #0
2002 LDX #2         (in this example, we use X as a counter which
                     represents the number of messages we are
                     printing)
2004 LDA 4000,Y     (get the "B" from "Ball out of . . .")
2007 BEQ 2016       (go to reduce [and check] the value of X)
2009 STA 32768,Y    (we're using the Y as a double-duty offset
                     again)
2012 INY
2013 JMP 2004
2016 INY            (we need to raise Y since we skipped that step
                     when we branched out of the loop)
2017 DEX            (at the end of the first message, X will be
                     a "1"; at the end of the second message,
                     it will be zero)
2018 BNE 2004       (if X isn't down to zero yet, re-enter the loop to
                     print out the second message)
```

To fill your screen with instructions instantly (say at the start of a game), you can use the following mass-move. We'll assume that the instructions go from 5000 to 5400 in memory and you want to transfer them to the PET screen (at $8000). If your computer's screen RAM moves around (adding memory to VIC will move the screen RAM address), you will need to know and substitute the correct address for your computer in these examples which print to the screen. This is in hex:

```
2000 LDY #0
2002 LDA 5000,Y
2005 STA 8000,Y
2008 LDA 5100,Y
200B STA 8100,Y
200E LDA 5200,Y
2011 STA 8200,Y
2014 LDA 5300,Y
2017 STA 8300,Y
201A INY
201B BNE 2002 (if Y hasn't counted up to zero — which comes
              just above 255 — go back and load-store the
              next character in each quarter of the large
              message )
```

This technique is fast and easy any time you want to mass-move one area of memory to another. It makes a copy and does not disturb the original memory. To mass-clear a memory zone (to clear the screen, for example), you can use a similar loop, but instead of loading the accumulator each time with a different character, you load it at the start with the character your computer uses to blank the screen. (Commodore including VIC and Apple = decimal 32; Atari = 0):

```
2000  LDA #20 (this example, in hex, blanks the PET screen)
2002  LDY #0
2004  STA 8000,Y
2007  STA 8100,Y
200A  STA 8200,Y
200D  STA 8300,Y
2010  DEY
2011  BNE 2004
```

Of course, you could simply JSR to the routine which already exists in your BASIC to clear the screen. In Chapter 7 we will explore the techniques of using parts of BASIC as examples to learn from and also as a collection of ready-made ML subroutines. Now, though, we can look at how subroutines are handled in ML.

5. The Subroutine and Jump Group: JMP, JSR, RTS

JMP has only one useful addressing mode: Absolute. You give it a firm, two-byte argument and it goes there. The argument is put into the Program Counter and control of the computer is transferred to this new address where an instruction there is acted upon. (There is a second addressing mode, JMP Indirect, which, you will recall, has a bug and is best left unused.)

JSR can only use Absolute addressing.

RTS's addressing mode is Implied. The address is on the stack, put there during the JSR.

None of these instructions has any effect on the flags.

JSR (Jump to SubRoutine) is the same as GOSUB in BASIC, but instead of giving a line number, you give an address in memory where the subroutine sits. RTS (ReTurn from Subroutine) is the same as RETURN in BASIC, but instead of returning to the next BASIC command, you return to the address following the JSR instruction (it's a three-byte-long ML instruction containing JSR and the two-byte target address). JMP (JuMP) is GOTO. Again, you JMP to an address, not a line number. As in BASIC, there is no RETURN from a JMP.

Some Further Cautions About The Stack

The stack is like a pile of coins. The last one you put on top of the pile is the first one pulled off later. The main reason that the 6502 sets aside an entire page of memory especially for the stack is that it has to know where to go back to after GOSUBs and JSRs.

A JSR instruction pushes the correct return address onto the "stack" and, later, the next RTS "pulls" the top two numbers off the stack to use as its argument (target address) for the return. Some programmers, as we noted before, like to play with the stack and use it as a temporary register to PHA (PusH Accumulator onto the stack). This sort of thing is best avoided until you are an advanced ML programmer. Stack manipulations often result in a very confusing program. Handling the stack is one of the few things that the computer does *for you* in ML. Let it.

The main function of the stack (as far as we're concerned) is to hold return addresses. It's done automatically for us by "pushes" with the JSR and, later, "pulls" (sometimes called *pops*) with the RTS. If we don't bother the stack, it will serve us well. There are thousands upon thousands of cells where you could temporarily leave the accumulator — or any other value — without fouling up the orderly arrangement of your return addresses.

Subroutines are extremely important in ML programming. ML programs are designed around them, as we'll see. There are times when you'll be several subroutines deep (one will call another which calls another); this is not as confusing as it sounds. Your main Player-input routine might call a print-message subroutine which itself calls a wait-until-key-is-pressed subroutine. If any of these routines PHA (PusH the Accumulator onto the stack), they then disturb the addresses on the stack. If the extra number on top of the stack isn't PLA-ed off (PulL Accumulator), the next RTS will pull off the number that was PHA'ed and half of the correct address. It will then merrily return to what it thinks is the correct address: it might land somewhere in the RAM, it might go to an address at the outer reaches of your operating system — but it certainly won't go where it should.

Some programmers like to change a GOSUB into a GOTO (in the middle of the action of a program) by PLA PLA. Pulling the two top stack values off has the effect of eliminating the most recent RTS address. It does leave a clean stack, but why bother to JSR at all if you later want to change it to a GOTO? Why not use JMP in the first place?

There are cases, too, when the stack has been used to hold the current condition of the flags (the Status Register byte). This is pushed/pulled from the stack with PHP (PusH Processor status) and PLP (PulL Processor status). If you should need to "remember" the condition of the status flags, why not just PHP PLA STA $NN? ("NN" means the address is your choice.) Set aside a byte somewhere that can hold the flags (they are always changing inside the Status Register) for later and keep the stack clean. Leave stack acrobatics to FORTH programmers. The stack, except for advanced ML, should be inviolate.

FORTH, an interesting language, requires frequent stack manipulations. But in the FORTH environment, the reasons for this and its protocol make excellent sense. In ML, though, stack manipulations are a sticky business.

Saving The Current Environment

There is one exception to our leave-the-stack-alone rule. Sometimes (especially when you are "borrowing" a routine from BASIC) you will want to take up with your own program from where it left off. That is, you might not want to write a "clear the screen" subroutine because you find the address of such a routine on your map of BASIC. However, you don't know what sorts of things BASIC will do in the meantime to your registers or your flags, etc. In other words, you just want to clear the screen without disturbing the flow of your program by unpredictable effects on your X, Y, A, and status registers. In such a case, you can use the following "Save the state of things" routine:

2000 PHP (push the status register onto the stack)
2001 PHA

2002 TXA
2003 PHA
2004 TYA
2005 PHA
2006 JSR (to the clear-the-screen routine in BASIC. The RTS will remove the return address [2009], and you'll have a mirror image of the things you had pushed onto the stack. They are pulled out in reverse order, as you can see below. This is because the first pull from the stack will get the *most recently pushed* number. If you make a little stack of coins, the *first* one you pull off will be the *last* one you put onto the stack.)
2009 PLA (now we reverse the order to get them back)
2010 TAY
2011 PLA
2012 TAX
2013 PLA (this one stays in A)
2014 PLP (the status register)

Saving the current state of things before visiting an uncharted, unpredictable subroutine is probably the only valid excuse for playing with the stack as a beginner in ML. The routine above is constructed to leave the stack intact. Everything that was pushed on has been pulled back off.

The Significance Of Subroutines

Maybe the best way to approach ML program writing — especially a large program — is to think of it as a collection of subroutines. Each of these subroutines should be small. It should be listed on a piece of paper followed by a note on what it needs as input and what it gives back as *parameters*. "Parameter passing" simply means that a subroutine needs to know things from the main program (parameters) which are handed to it (passed) in some way.

The current position of the ball on the screen is a parameter which has its own "register" (we set aside a register for it at the start when we were assigning memory space on paper). So, the "send the ball down one space" subroutine is a double-adder which adds 40 or whatever to the "current position register." This value always sits in the register to be used any time any subroutine needs this information. The "send the ball down one" subroutine *sends* the current-position parameter by *passing* it to the current-position register.

This is one way that parameters are passed. Another illustration might be when you are telling a delay loop how long to delay. Ideally, your delay subroutine will be multi-purpose. That is, it can delay for anywhere from ½ second to 60 seconds or something. This means that the subroutine itself isn't locked into a particular length of delay. The main program will ''pass'' the amount of delay to the subroutine.

```
3000 LDY #0
3002 INY
3003 BNE 3002
3005 DEX
3006 BNE 3000
3008 RTS
```

Notice that X never is initialized (set up) here with any particular value. This is because the value of X is passed to this subroutine from the main program. If you want a short delay, you would:

```
2000 LDX #5 (decimal)
2002 JSR 3000
```

And for a delay which is twice as long as that:

```
2000 LDX #10
2002 JSR 3000
```

In some ways, the less a subroutine does, the better. If it's not entirely self-sufficient, and the shorter and simpler it is, the more versatile it will be. For example, our delay above could function to time responses, to hold sounds for specific durations, etc. When you make notes, write something like this: 3000 DELAY LOOP (Expects duration in X. Returns 0 in X.). The longest duration would be LDX #0. This is because the first thing that happens to X in the delay subroutine is DEX. If you DEX a zero, you get 255. If you need longer delays than the maximum value of X, simply:

```
3000 LDX #0
3002 JSR 3000
3005 JSR 3000 (notice that we don't need to set X to zero this
               second time. It returns from the subroutine
               with a zeroed X.)
```

You could even make a loop of the JSR's above for extremely long delays. The point to notice here is that it helps to document each subroutine in your library: what parameters it expects, what registers, flags, etc., it changes, and what it leaves behind as a result. This documentation — a single sheet of paper will do — helps you remember each routine's address and lets you know what effects and preconditions are involved.

JMP

Like BASIC's GOTO, JMP is easy to understand. It goes to an address: JMP 5000 leaps from wherever it is to start carrying out the instructions which start at 5000. It doesn't affect any flags. It doesn't do anything to the stack. It's clean and simple. Yet some advocates of ''structured programming'' suggest avoiding JMP (and GOTO in BASIC). Their reasoning is that JMP is a shortcut and a poor programming habit.

For one thing, they argue, using GOTO makes programs confusing. If you drew lines to show a program's ''flow'' (the order in which instructions are carried out), a program with lots of GOTO's would look like boiled spaghetti. Many programmers feel, however, that JMP has its uses. Clearly, you should not overdo it and lean heavily on JMP. In fact, you might see if there isn't a better way to accomplish something if you find yourself using it all the time and your programs are becoming impossibly awkward. But JMP is convenient, often necessary in ML.

A 6502 Bug

On the other hand, there is another, rather peculiar JMP form which is hardly ever used in ML: JMP (5000). This is an *indirect* jump which works like the *indirect* addressing we've seen before. Remember that in Indirect,Y addressing (LDA (81),Y), the number in Y is added to the *address* found in 81 and 82. This address is the *real* place we are LDAing from, sometimes called the *effective address.* If 81 holds a 00, 82 holds a 40, and Y holds a 2, the address we LDA from is going to be 4002. Similarly (but without adding Y), the effective address formed by the two bytes at the address inside the parentheses becomes the place we JMP to in JMP (5000).

There are no necessary uses for this instruction. Best avoid it the same way you avoid playing around with the stack until you're an ML expert. If you find it in your computer's BASIC ROM code, it will probably be involved in an ''indirect jump table,'' a series of registers which are dynamic. That is, they can be changed as the program progresses. Such a technique is very close to a self-altering program and would have few uses for the beginner in ML programming. Above all, there is a bug in the 6502 itself which causes indirect JMP to malfunction under certain circumstances. Put JMP ($NNNN) into the same category as BPL and BMI. Avoid all three.

If you decide you must use indirect JMP, be sure to avoid the edge of pages: JMP ($NNFF). The ''NN'' means ''any number.'' Whenever the low byte is right on the edge, if $FF is ready to reset to 00, this instruction will correctly use the low byte (LSB) found in address $NNFF, but it will not pick up the high byte (MSB) from $NNFF plus one, as it should. It gets the MSB from NN00!

Here's how the error would look if you had set up a pointer to address $5043 at location $40FF:

```
$40FF 43
$4100 50
```

Your intention would be to JMP to $5403 by bouncing off this pointer. You would write JMP ($40FF) and expect that the next instruction the computer would follow would be whatever is written at $5043. Unfortunately, you would land at $0043 instead (if address $4000 held a zero). It would get its MSB from $4000.

6. Debuggers: BRK and NOP

BRK and NOP have no argument and are therefore members of that class of instructions which use only the Implied addressing mode. They also affect no flags in any way with which we would be concerned. BRK does affect the I and B flags, but since it is a rare situation which would require testing those flags, we can ignore this flag activity altogether.

After you've assembled your program and it doesn't work as expected (few do), you start *debugging*. Some studies have shown that debugging takes up more than fifty percent of programming time. Such surveys might be somewhat misleading, however, because ''making improvements and adding options'' frequently takes place after the program is allegedly finished, and would be thereby categorized as part of the debugging process.

In ML, debugging is facilitated by setting *breakpoints* with BRK and then seeing what's happening in the registers or memory. If you insert a BRK, it has the effect of halting the program and sending you into your monitor where you can examine, say, the Y register to see if it contains what you would expect it to at this point in the program. It's similar to BASIC's STOP instruction:

```
2000 LDA #15
2002 TAY
2003 BRK
```

If you run the above, it will carry out the instructions until it gets to BRK when it will put the program counter *plus two* on the stack, put the status register on the stack, and load the program counter with whatever is in addresses $FFFE, $FFFF. These are the two highest addresses in your computer and they contain the *vector* (a pointer) for an interrupt request (IRQ).

These addresses will point to a general interrupt handler and, if your computer has a monitor, its address might normally be found here. Remember, though, that when you get ready to CONT, the address on the top of the stack will be the BRK address plus two. Check the program counter (it will appear when your monitor displays the registers) to see if you need to modify it to point to the next instruction instead of pointing, as it might be, to an argument. Some monitors adjust the program counter when they are BRKed to so that you can type *g* (go) in the same way that you would type CONT in BASIC. See the instructions for your particular monitor.

Debugging Methods

In effect, you debug whenever your program runs merrily along and then does something unexpected. It might crash and lock you out. You look for a likely place where you think it is failing and just insert a BRK right over some other instruction. Remember that in the monitor mode you can display a hex dump and type over the hex numbers on screen, hitting RETURN to change them. In the example above, imagine that we put the BRK over an STY 8000. Make a note of the hex number of the instruction you covered over with the BRK so you can restore it later. After checking the registers and memory, you might find something wrong. Then you can fix the error.

If nothing seems wrong at this point, restore the original STY over the BRK, and insert a BRK in somewhere further on. By this process, you can isolate the cause of an oddity in your program. Setting breakpoints (like putting STOP into BASIC programs) is an effective way to run part of a program and then examine the variables.

If your monitor or assembler allows *single-stepping*, this can be an excellent way to debug, too. Your computer performs each instruction in your program one step at a time. This is like having BRK between each instruction in the program. You can control the speed of the stepping from the keyboard. Single-stepping automates breakpoint checking. It is the equivalent of the TRACE command sometimes used to debug BASIC programs.

Like BRK ($00), the hex number of NOP ($EA) is worth memorizing. If you're working within your monitor, it will want you to work in hex numbers. These two are particularly worth knowing. NOP means No OPeration. The computer slides over NOP's without taking any action other than increasing the program counter. There are two ways in which NOP can be effectively used.

First, it can be an eraser. If you suspect that STY 8000 is causing all the trouble, try running your program with everything else the same, but with STY 8000 erased. Simply put three EA's over the instruction and argument. (Make a note, though, of what was under the EA's so you can restore it.) Then, the program will run without this instruction and you can watch the effects.

Second, it is sometimes useful to use EA to temporarily hold open some space. If you don't know something (an address, a graphics value) during assembly, EA can mark that this space needs to be filled in later before the program is run. As an instruction, it will let the program slide by. But, remember, as an address or a number, EA will be thought of as 234. In any case, EA could become your "fill this in" alert within programs in the way that we use self-branching (leaving a zero after a BNE or other branch instruction) to show that we need to put in a forward branch's address.

When the time comes for you to "tidy up" your program, use your monitor's "find" command, if it has one. This is a search routine: you tell it where to start and end and what to look for, and it prints out the addresses of any matches it finds. It's a useful utility; if your monitor does not have a search function, you might consider writing one as your first large ML project. You can use some of the ideas in Chapter 8 as a starting point.

Less Common Instructions

The following instructions are not often necessary for beginning applications, but we can briefly touch on their main uses. There are several "logical" instructions which can manipulate or test individual bits within each byte. This is most often necessary when interfacing. If you need to test what's coming in from a disk drive, or translate on a bit-by-bit level for I/O (input/output), you might work with the "logical" group.

In general, this is handled for you by your machine's operating system and is well beyond beginning ML programming. I/O is perhaps the most difficult, or at least the most complicated, aspect of ML programming. When putting things on the screen, programming is fairly straightforward, but handling the data stream into and out of a disk is pretty involved. Timing must be precise, and the preconditions which need to be established are complex.

For example, if you need to "mask" a byte by changing some of its bits to zero, you can use the AND instruction. After an AND, *both* numbers must have contained a 1 in any particular bit position for it to result in a 1 in the answer. This lets you set up a mask: 00001111 will zero any bits within the left four positions. So, 00001111 AND 11001100 result in 00001100. The unmasked bits remained unchanged, but the four high bits were all masked and zeroed. The ORA instruction is the same, except it lets you mask to *set* bits (make them a 1). 11110000 ORA 11001100 results in 11111100. The accumulator will hold the results of these instructions.

EOR (Exclusive OR) permits you to "toggle" bits. If a bit is one it will go to zero. If it's zero, it will flip to one. EOR is sometimes useful in games. If you are heading in one direction and you want to go back when bouncing a ball off a wall, you could "toggle." Let's say that you use a register to show direction: when the ball's going up, the byte contains the number 1 (00000001), but down is zero (00000000). To toggle this least significant bit, you would EOR with 00000001. This would flip 1 to zero and zero to 1. This action results in the *complement* of a number. 11111111 EOR 11001100 results in 00110011.

To know the effects of these logical operators, we can look them up in ''truth tables'' which give the results of all possible combinations of zeros and ones:

AND	OR	EOR
0 AND 0=0	0 OR 0=0	0 EOR 0=0
0 AND 1=0	0 OR 1=1	0 EOR 1=1
1 AND 0=0	1 OR 0=1	1 EOR 0=1
1 AND 1=1	1 OR 1=1	1 EOR 1=0

BIT Tests

Another instruction, BIT, also tests (it does an AND), but, like CMP, it does not affect the number in the accumulator — it merely sets flags in the status register. The N flag is set (has a 1) if bit seven has a 1 (and vice versa). The V flag responds similarly to the value in the sixth bit. The Z flag shows if the AND resulted in zero or not. Instructions, like BIT, which do not affect the numbers being tested are called *non-destructive.*

We discussed LSR and ASL in the chapter on arithmetic: they can conveniently divide and multiply by two. ROL and ROR *rotate* the bits left or right in a byte but, unlike with the Logical Shift Right or Arithmetic Shift Left, no bits are dropped during the shift. ROL will leave the 7th (most significant) bit in the carry flag, leave the carry flag in the 0th (least significant bit), and move every other bit one space to the left:

ROL 11001100 (with the carry flag set) results in
10011001 (carry is still set, it got the leftmost 1)

If you disassemble your computer's BASIC, you may well look in vain for an example of ROL, but it and ROR are available in the 6502 instruction set if you should ever find a use for them. Should you go into advanced ML arithmetic, they can be used for multiplication and division routines.

Three other instructions remain: SEI (SEt Interrupt), RTI (ReTurn from Interrupt), and CLI (CLear Interrupt). These operations are, also, beyond the scope of a book on beginning ML programming, but we'll briefly note their effects. Your computer gets busy as soon as the power goes on. Things are always happening: timing registers are being updated; the keyboard, the video, and the peripheral connectors are being refreshed or examined for signals. To ''interrupt'' all this activity, you can SEI, perform some task, and then CLI to let things pick up where they left off.

SEI sets the interrupt flag. Following this, all *maskable* interruptions (things which can be blocked from interrupting when the interrupt status flag is up) are no longer possible. There are also *non-maskable* interrupts which, as you might guess, will jump in anytime, ignoring the status register.

The RTI instruction (ReTurn from Interrupt) restores the program counter and status register (takes them from the stack), but the X, Y, etc., registers might have been changed during the interrupt. Recall that our discussion of the BRK involved the above actions. The key difference is that BRK stores the program counter plus two on the stack and sets the B flag on the status register. CLI puts the interrupt flag down and lets all interrupts take place.

If these last instructions are confusing to you, it doesn't matter. They are essentially hardware and interface related. You can do nearly everything you will want to do in ML without them. How often have you used WAIT in BASIC?

7

Borrowing From BASIC

BASIC is a collection of ML subroutines. It is a large web of hundreds of short, ML programs. Why not use some of them by JSRing to them? At times, this is in fact the best solution to a problem.

How would this differ from BASIC itself? Doesn't BASIC just create a series of JSR's when it RUNs? Wouldn't using BASIC's ML routines in this way be just as slow as BASIC?

In practice, you will not be borrowing from BASIC all that much. One reason is that such JSRing makes your program far less *portable,* less easily RUN on other computers or other models of your computer. When you JSR to an address within your ROM set to save yourself the trouble of re-inventing the wheel, you are, unfortunately, making your program applicable only to machines which are the same model as yours. The subroutine to allocate space for a string in memory is found at $D3D2 in the earliest PET model. A later version of PET BASIC (Upgrade) used $D3CE and the current models use $C61D. With Atari, Texas Instruments, Sinclair and other computers as exceptions, Microsoft BASIC is nearly universally used in personal computers. But each computer's version of Microsoft differs in both the order and the addresses of key subroutines.

Kernals And Jump Tables

To help overcome this lack of portability, some computer manufacturers set aside a group of frequently used subroutines and create a *Jump Table,* or *kernal,* for them. The idea is that future, upgraded BASIC versions will still retain this table. It would look something like this:

```
FFCF 4C 15 F2 (INPUT one byte)
FFD2 4C 66 F2 (OUTPUT one byte)
FFD5 4C 01 F4 (LOAD something)
FFD8 4C DD F6 (SAVE something)
```

This example is part of the Commodore kernal.

There is a trick to the way this sort of table works. Notice that each member of the table begins with 4C. That's the JMP instruction and, if you land on it, the computer bounces right off to the address which follows. $FFD2 is a famous one in Commodore computers. If you load the accumulator with a number (LDA #65) and then JSR FFD2, a character will be printed on the screen. The screen location is incremented each time you use it, so it works semi-automatically. In other words, it also keeps track of the current ''cursor position'' for you.

This same ''output'' routine will work for a printer or a disk or a tape — anything that the computer sees as an output device. However, unless you open a file to one of the other devices (it's simplest to do this from BASIC in the normal way and then SYS, USR, or CALL to an ML subroutine), the computer defaults to (assumes) the screen as the output device, and FFD2 prints there.

What's curious about such a table is that you JSR to FFD2 as you would to any other subroutine. But where's the subroutine? It's not at FFD5. That's a different JMP to the LOAD code. A naked JMP (there is no RTS here in this jump table) acts like a rebound: you hit one of these JMP's in the table and just bounce off it to the true subroutine.

The real subroutine (at $F266 in one BASIC version's $FFD2's JMP) will perform what you expect. Why not just JSR to F266 directly? Because, on other models of Commodore computers — Original BASIC, for example — the output subroutine *is not located at F266.* It's somewhere else. But a JSR to FFD2 *will* rebound you to the right address in any Commodore BASIC. All Commodore machines have the correct JMP for their particular BASIC set up at FFD2. This means that you can JSR to FFD2 on any Commodore computer and get predictable results, an output of a byte.

So, if you look into your BASIC code and find a series of JMP's (4C xx xx 4C xx xx), it's a jump table. Using it should help make your programs compatible with later versions of BASIC which might be released. Though this is the purpose of such tables, there are never any guarantees that the manufacturer will consistently observe them. And, of course, the program which depends on them will certainly not work on any other computer brand.

What's Fastest?

Why, though, is a JSR into BASIC code faster than a BASIC program? When a BASIC program RUNs, it is JSRing around inside itself. The answer is that a program written entirely in ML, aside from the fact that it borrows only sparingly from BASIC prewritten routines, differs from BASIC in an important way. A finished ML program is like *compiled* code; that is, it is ready to execute without any overhead. In BASIC each command or instruction must be interpreted *as it RUNs.* This is why BASIC is called an ''interpreter.'' Each instruction must be looked up in a table to find its address in ROM. This takes time. Your ML code will contain the addresses for its JSR's. When ML runs, the instructions don't need the same degree of interpretation by the computer.

There are special programs called *compilers* which take a BASIC program and transform (''compile'') it into ML-like code which can then be executed like ML, without having to interpret each command. The JSR's are within the compiled program, just as in ML. Ordinarily, compiled programs will RUN perhaps 20 to 40 times faster than the BASIC program they grew out of. (Generally, there is a price to pay in that the compiled version is almost always larger than its BASIC equivalent.)

Compilers are interesting; they act almost like automatic ML writers. You write it in BASIC, and they translate it into an ML-like program. Even greater improvements in speed can be achieved if a program uses no floating point (decimal points) in the arithmetic. Also, there are ''optimized'' compilers which take longer during the translation phase to compile the finished program, but which try to create the fastest, most efficient program design possible. A good compiler can translate an 8K BASIC program in two or three minutes.

GET And PRINT

Two of the most common activities in a computer program are getting characters from the keyboard and printing them to the screen. To illustrate how to use BASIC from within an ML program, we'll show how both of these tasks can be accomplished from within ML.

For the Atari, $F6E2 works like BASIC's GET#. If you JSR $F6E2, the computer will wait until a key is pressed on the keyboard. Then, when one is pressed, the numerical code for that key is put into the accumulator, and control is returned to your ML program. To try this, type:

```
2000 JSR $F6E2
2003 BRK
```

Then run this program and hit a key on the keyboard. Notice that the code number for that letter appears in the accumulator.

Another location within Atari's BASIC ROM will print a character (whatever's in the accumulator) to the next available position on the screen. This is like PUT#6. Try combining the above GET# with this:

```
2000 JSR $F6E2 (get the character)
2003 JSR $F6A4 (print to the screen)
2006 BRK
```

Using $F6A4 changes the numbers in the X and Y registers (explained below).

For the Apple, there are BASIC routines to accomplish these same jobs. Apple Microsoft BASIC's GET waits for user input. (Commodore's GET doesn't wait for input.)

```
2000 JSR $FD0C (GET a byte from the keyboard)
2003 RTS       (the character is in the accumulator)
```

This address, $FD0C, will wait until the user types in a character. It will position a flashing cursor at the correct position. However, it will not print an "echo," an image of the character on the screen.

To print to the screen:

```
2000 LDA #65     (put "a" into the accumulator)
2002 JSR $FBFD   (print it)
```

For Commodore computers (VIC, 64, and PET/CBM) which also use Microsoft BASIC, the two subroutines are similar:

```
2000 JSR $FFE4   (GET whatever key is being pressed)
2003 BEQ 2000    (if no key is pressed, a zero is in the
                 accumulator, so you BEQ back and try for a
                 character again)
2005 RTS         (the character's value is in the accumulator)
```

The $FFE4 is another one of those "kernal" jump table locations common to all Commodore machines. It performs a GET.

An ML routine within your BASIC which keeps track of the current cursor position and will print things to the screen is often needed in ML programming.

The VIC, 64, and PET/CBM use the routine called by $FFD2. Apple uses $FDED. Atari uses $F6A4.

You can safely use the Y register to print out a series of letters (Y used as an index) in any BASIC except Atari's. You could print out a whole word or block of text or graphics stored at $1000 in the following way. (See Program 7-1.)

Atari's BASIC alters the X and Y registers when it executes its "print it" subroutine so you need to keep count some other way. Whenever you borrow from BASIC, be alert to the possibility that the A, X, or Y registers, as well as the flags in the status register, might well be changed by the time control is returned to your ML program. Here's one way to print out messages on the Atari. (See Program 7-2.)

If you look at Appendix B you will see that there are hundreds of freeze-dried ML modules sitting in BASIC. (The maps included in this book are for VIC, PET, Atari, and Commodore 64. Appendix B contains information on how to obtain additional maps for Apple and Atari.)

It can be intimidating at first, but disassembling some of these routines is a good way to discover new techniques and to see how professional ML programs are constructed. Study of your computer's BASIC is worth the effort, and it's something you can do for yourself. From time to time, books are published which go into great detail about each BASIC routine. They, too, are often worth studying.

Program 7-1.

```
                  ; COMMODORE & APPLE VERSION
                                .BA $2000
                                .OS               ;(OUTPUT SOURCE CODE)
                  COUNTER       .DE $55           ;(WILL HOLD INDEX)
2000- 53 55 50    STRING        .BY 'SUPERDUPER'  ; STORE THIS TEXT STRING
2003- 45 52 44
2006- 55 50 45
2009- 52
                  LENGTH        .DE 11            ; STRING IS 10 CHARS. LONG
                  ;
                  PRINTIT       .DE $FFD2         ;(COMMODORE)
                  ;
                  ; (FOR APPLE USE $FDED)
                  ;
200A- A0 00       START         LDY #$00
200C- B9 00 20    LOOP          LDA STRING,Y
200F- 20 D2 FF                  JSR PRINTIT
2012- C8                        INY
2013- C0 0B                     CPY #LENGTH       ;(NOTE LENGTH IS PLUS ONE.
2015- D0 F5                     BNE LOOP
2017- 60                        RTS
                                .EN
ENDPASS
 ---  LABEL FILE:  ---
COUNTER =0055          LENGTH =000B          LOOP =200C
PRINTIT =FFD2          START =200A           STRING =2000
```

Program 7-2.

```
               0010 ; ATARI VERSION
               0020             .BA $0600
               0030             .OS                  ;(OUTPUT SOURCE CODE)
               0040 COUNTER     .DE $55              ;(WILL HOLD INDEX)
0600- 53 55 50 0050 STRING      .BY 'SUPERDUPER' ; STORE THIS TEXT STRING
0603- 45 52 44
0606- 55 50 45
0609- 52
               0060 LENGTH      .DE 11               ; STRING IS 10 CHARS. LONG
               0070 ;
               0080 PRINTIT     .DE $F6A4            ;(ATARI)
               0090 ;
060A- A9 00    0100 START       LDA #00
060C- 85 55    0110             STA *COUNTER         ; (ANY FREE ZERO PAGE)
060E- A0 55    0120 LOOP        LDY #COUNTER
0610- B9 00 06 0130             LDA STRING,Y
0613- 20 A4 F6 0140             JSR PRINTIT
0616- E6 55    0150             INC *COUNTER
0618- A9 0B    0160             LDA #LENGTH
061A- C5 55    0170             CMP *COUNTER
061C- D0 F0    0180             BNE LOOP
061E- 60       0190             RTS
               0200             .EN
ENDPASS
 ---  LABEL FILE:  ---
COUNTER =0055           LENGTH =000B            LOOP =060E
PRINTIT =F6A4           START =060A             STRING =0600
```

8

Building A Program

Using what we've learned so far, and adding a couple of new techniques, let's build a useful program. This example will demonstrate many of the techniques we've discussed and will also show some of the thought processes involved in writing ML.

Among the computer's more impressive talents is searching. It can run through a mass of information and find something very quickly. We can write an ML routine which looks through any area of memory to find matches with anything else. If your BASIC doesn't have a FIND command or its equivalent, this could come in handy. Based on an idea by Michael Erperstorfer published in *COMPUTE!* Magazine, this ML program will report the line numbers of all the matches it finds.

Safe Havens

Before we go through some typical ML program-building methods, let's clear up the "where do I put it?" question. ML can't just be dropped anywhere in memory. When the Simple Assembler asks "Starting Address?", you can't give it any address you want to. RAM is used in many ways. There is always the possibility that a BASIC program might be residing in part of it (if you are combining ML with a BASIC program). Or BASIC might use part of RAM to store arrays or variables. During execution, these variables might write (POKE) into the area that you placed your ML program, destroying it. Also, the operating system, the disk operating system, cassette/disk loads, printers — they all use parts of RAM for their activities. There are other things going on in the computer beside your hard-won ML program.

Obviously, you can't put your ML into ROM addresses. That's impossible. Nothing can be POKEd into those addresses. The 64 is an exception to this. You *can* POKE into ROM areas because a RAM exists *beneath* the ROM. Refer to the *Programmer's Reference Guide* or see Jim Butterfield's article on 64 architecture (*COMPUTE!* Magazine, January 1983) for details.

Where to put ML? There are some fairly safe areas.

If you are using Applesoft in ROM, 768 to 1023 ($0300 to $03FF) is safe. Atari's page six, 1536 to 1791 ($0600 to $06FF) is good. The 64 and VIC's cassette buffer at 828 to 1019 ($033C to $03FB) are good if you are not LOADing or SAVEing from tape.

The PET/CBM makes provision for a second cassette unit. In theory, it would be attached to the computer to allow you to update files or make copies of programs from Cassette #1 to Cassette #2. In practice, no one has mentioned finding a use for a second cassette drive. It is just as easy to use a single cassette for anything that a second cassette could do. As a result, the buffer (temporary holding area) for bytes streaming in from the second cassette unit is very safe indeed. No bytes ever flow in from the phantom unit so it is a perfect place to put ML.

The "storage problem" can be solved by knowing the free zones, or creating space by changing the computer's understanding of the start or end of BASIC programs. When BASIC is running, it will set up arrays and strings in RAM memory. Knowing where a BASIC program ends is not enough. It will use additional RAM. Sometimes it puts the strings just after the program itself. Sometimes it builds them down from the "top of memory," the highest RAM address. Where are you going to hide your ML routine if you want to use it along with a BASIC program? How are you going to keep BASIC from overwriting the ML code?

Misleading The Computer

If the ML is a short program you can stash it into the safe areas listed above. Because these safe areas are only a couple of hundred bytes long, and because so many ML routines want to use that area, it can become crowded. Worse yet, we've been putting the word "safe" in quotes because it just isn't all that reliable. Apple uses the "safe" place for high-res work, for example. The alternative is to deceive the computer into thinking that its RAM is smaller than it really is. This is the real solution.

Your ML will be truly safe if your computer doesn't even suspect the existence of set-aside RAM. It will leave the safe area alone because you've told it that it has less RAM than it really does. Nothing can overwrite your ML program after you misdirect your computer's operating system about the size of its RAM memory. There are two bytes in zero page which tell the computer the highest RAM address. You just change those bytes to point to a lower address.

These crucial bytes are 55 and 56 ($37,38) in the 64 and VIC. They are 52,53 ($34,35) in PET/CBM Upgrade and 4.0 BASIC. In the PET with Original ROM BASIC, they are 134,135 ($86,87). The Apple uses 115,116 ($73,74), and you lower the Top-of-BASIC pointer just as you do in Commodore machines.

The Atari does something similar, but with the *bottom* of RAM. It is easier with the Atari to store ML just below BASIC than above it. Bump up the "lomem" pointer to make some space for your ML. It's convenient to start ML programs which are too long to fit into page six ($0600-06FF) at $1F00 and then put this address into lomem. The LSB and MSB are reversed, of course, as the 6502 requires its pointers to be like this:

```
$02E7 00
$02E8 1F
```

$02E7,8 is Atari's low memory pointer. You should set up this pointer (LDA $00, STA $02E7, LDA #$1F, STA $02E8) as part of your ML program. Following that pointer setup, JMP $A000 which initializes BASIC. If you are not combining ML with a BASIC program, these preliminary steps are not necessary.

Safe Atari zero page locations include $00-04, $CB-D0, $D4-D9 (if floating point numbers are not being used); $0400 (the printer and cassette buffer), $0500-057F (free), $0580-05FF (if floating point and the Editor are not being used), $0600-06FF (free) are also safe. No other RAM from $0700 (Disk Operating System) to $9FFF or $BFFF is protected from BASIC.

To repeat: address pointers such as these are stored in LSB, MSB order. That is, the more significant byte comes second (this is the reverse of normal, but the 6502 requires it of address pointers). For example, $8000, divided between two bytes in a pointer, would look like this:

```
0073 00
0074 80
```

As we mentioned earlier, this odd reversal is a peculiarity of the 6502 that you just have to get used to. Anyway, you can lower the computer's opinion of the top-of-RAM-memory, thereby making a safe place for your ML, by changing the MSB. If you need one page (256 bytes): POKE 116, PEEK (116)-1 (Apple). For four pages (1024 bytes) on the Upgrade and 4.0 PETs: POKE 53, PEEK (53) -4. Then your BA or start of assembling could begin at (Top-of-RAM-255 or Top-of-RAM-1023, respectively. You don't have to worry much about the LSB here. It's usually zero. If not, take that into account when planning where to begin storage of your object code.

Building The Code

Now we return to the subject at hand — building an ML program. Some people find it easiest to mentally break a task down into several smaller problems and then weave them into a complete program. That's how we'll look at our search program. (See Program 8-1.)

For this exercise, we can follow the PET/CBM 4.0 BASIC version to see how it is constructed. All the versions (except Atari's) are essentially the same, as we will see in a minute. The only differences are in the locations in zero page where addresses are temporarily stored, the "start-of-BASIC RAM" address, the routines to print a

Program 8-1. PET Search (4.0 BASIC Version).

```
                  0010 ; SEARCH THROUGH BASIC
                  0015 ; PET 4.0 VERSION
                  0016 ;-------------------------------------------------------
                  0017 ; -O-    DEFINE VARIABLES BY GIVING THEM LABELS.
                  0018 ;
                  0020 L1L         .DE $BA                 ;STORE THESE IN
                  0030 L2L         .DE $BC                 ;UNUSED ZERO PG AREA
                  0040 FOUND       .DE $36
                  0050 BASIC       .DE $0400
                  0060 PRINT       .DE $FFD2               ; PRINT A CHAR.
                  0070 PLINE       .DE $CF7F               ; PRINT LINE#
                  0100             .BA $0360               ; 2ND CASSETTE BUFFER
                  0110             .OS
                  0120 ;-------------------------------------------------------
                  0121 ; -O- INITIALIZE POINTERS.
                  0130 ;
                  0140             ;
0360- AD 01 04    0150             LDA BASIC+1             ;GET ADDR OF NEXT
0363- 85 BA       0160             STA *L1L                ;BASIC LINE
0365- AD 02 04    0170             LDA BASIC+2
0368- 85 BB       0180             STA *L1L+1
                  0181 ;-------------------------------------------------------
                  0182 ; -O- SUBROUTINE TO CHECK FOR 2 ZEROS.  IF WE DON'T
                  0183 ; FIND THEM, WE ARE NOT AT THE END OF THE PROGRAM.
                  0184 ;
```

```
036A- A0 00     0190 READLINE   LDY #$00
036C- B1 BA     0200            LDA (L1L),Y
036E- D0 06     0210            BNE GO.ON          ; NOT END OF LINE
0370- C8        0220            INY
0371- B1 BA     0230            LDA (L1L),Y        ; 00 00 = END OF PROG.
0373- D0 01     0240            BNE GO.ON
0375- 60        0250 END        RTS                ;RETURN TO BASIC
                0251 ;-----------------------------------------------
                0252 ; -O- SUBROUTINE TO UPDATE POINTERS TO THE NEXT LINE
                0253 ;     AND STORE THE CURRENT LINE NUMBER IN CASE WE
                0254 ;     FIND A MATCH AND NEED TO PRINT THE LINE #.
                0255 ;     ALSO, WE ADD 4 TO THE CURRENT LINE POINTER SO THAT
                0256 ;     WE ARE PAST THE LINE # AND "POINTER-TO-NEXT-LINE"
                0257 ;     INFORMATION.  WE ARE THEN POINTING AT THE 1ST CHAR.
                0258 ;     IN THE CURRENT LINE AND CAN COMPARE IT TO THE SAMPLE.
                0259 ;
0376- A0 00     0260 GO.ON      LDY #$00
0378- B1 BA     0270            LDA (L1L),Y        ; GET NEXT LINE
037A- 85 BC     0280            STA *L2L           ; ADDRESS AND
037C- C8        0290            INY                ; STORE IT IN L2L
037D- B1 BA     0300            LDA (L1L),Y
037F- 85 BD     0310            STA *L2L+1
0381- C8        0320            INY
0382- B1 BA     0330            LDA (L1L),Y        ; PUT LINE #
0384- 85 36     0340            STA *FOUND         ; IN STORAGE TOO
0386- C8        0350            INY                ; IN CASE IT
0387- B1 BA     0360            LDA (L1L),Y        ; NEEDS TO BE
0389- 85 37     0370            STA *FOUND+1       ;PRINTED OUT LATER
038B- A5 BA     0380            LDA *L1L
```

```
038D- 18          0390          CLC                   ; MOVE FORWARD TO 1ST
038E- 69 04       0400          ADC #$04              ; PART OF BASIC TEXT
0390- 85 BA       0410          STA *L1L              ; (PAST LINE # AND
0392- A5 BB       0420          LDA *L1L+1            ; OF NEXT LINE)
0394- 69 00       0430          ADC #$00
0396- 85 BB       0440          STA *L1L+1
                  0441 ;-----------------------------------------------------
                  0442 ; -O- SUBROUTINE TO CHECK FOR ZERO (LINE IS FINISHED?)
                  0443 ;      AND THEN CHECK 1ST CHARACTER IN BASIC LINE AGAINST
                  0444 ;     1ST CHARACTER IN SAMPLE STRING AT LINE 0:.  IF THE
                  0445 ;     1ST CHARACTERS MATCH, WE MOVE TO A FULL STRING
                  0446 ;     COMPARISON IN THE SUBROUTINE CALLED "SAME."  IF 1ST
                  0447 ;     CHARS. DON'T MATCH, WE RAISE THE "Y" COUNTER AND
                  0448 ;     CHECK FOR A MATCH IN THE 2ND CHAR. OF THE CURRENT
                  0449 ;     BASIC LINE'S TEXT.
                  0450 ;
0398- A0 00       0451          LDY #$00
039A- B1 BA       0460 LOOP     LDA (L1L),Y
039C- F0 1C       0470          BEQ STOPLINE          ; ZERO = LINE FINISHED
039E- CD 06 04    0480          CMP BASIC+6           ; SAME AS 1ST SAMPLE CHAR?
03A1- F0 04       0490          BEQ SAME              ; YES? CHECK WHOLE STRING
03A3- C8          0500          INY                   ; NO? CONTINUE SEARCH
03A4- 4C 9A 03    0510          JMP LOOP
                  0511 ;-----------------------------------------------------
                  0512 ; -O- SUBROUTINE TO LOOK AT EACH CHARACTER IN BOTH THE
                  0513 ;     SAMPLE (LINE 0) AND THE TARGET (CURRENT LINE) TO
                  0514 ;     SEE IF THERE IS A PERFECT MATCH.  Y KEEPS TRACK OF
                  0515 ;     TARGET.  X INDEXES SAMPLE.  IF WE FIND A MISMATCH
                  0516 ;     BEFORE A LINE-END ZERO, WE FALL THROUGH TO LINE
                  0517 ;     590 AND JUMP BACK UP TO 460 WHERE WE CONTINUE ON
```

```
                0518 ;    LOOKING FOR 1ST CHAR MATCHES IN THE CURRENT LINE.
                0519 ;
03A7- A2 00     0520 SAME       LDX #$00             ; COMPARE SAMPLE TO
03A9- E8        0530 COMPARE    INX                  ; TARGET
03AA- C8        0540            INY
03AB- BD 06 04  0550            LDA BASIC+6,X
03AE- F0 07     0560            BEQ PERFECT          ; LINE ENDS SO PRINT
03B0- D1 BA     0570            CMP (L1L),Y
03B2- F0 F5     0580            BEQ COMPARE          ; CONTINUE COMPARE
03B4- 4C 9A 03  0590            JMP LOOP             ; NO MATCH
03B7- 20 C5 03  0600 PERFECT    JSR PRINTOUT
                0601 ;-----------------------------------------------------
                0602 ; -O- SUBROUTINE TO REPLACE "CURRENT LINE" POINTER
                0603 ;      WITH THE "NEXT LINE" POINTER WE SAVED IN THE SUBROUT
                0604 ;     STARTING AT LINE 260.
                0605 ;     THEN JUMP BACK TO THE START WITH THE CHECK FOR THE
                0606 ;     END-OF-PROGRAM DOUBLE ZERO.  THIS IS THE LAST SUBROUT
                0607 ;     IN THE MAIN LOOP OF THE PROGRAM.
                0608 ;
03BA- A5 BC     0610 STOPLINE   LDA *L2L             ; TRANSFER NEXT LINE
03BC- 85 BA     0620            STA *L1L             ; ADDRESS POINTER TO
03BE- A5 BD     0630            LDA *L2L+1           ; CURRENT LINE POINTER
03C0- 85 BB     0640            STA *L1L+1           ; TO GET READY TO READ
03C2- 4C 6A 03  0650            JMP READLINE         ; THE NEXT LINE.
                0651 ;-----------------------------------------------------
                0652 ; -O- SUBROUTINE TO PRINT OUT A BASIC LINE NUMBER.
                0653 ;     IN MICROSOFT IT TAKES THE NUMBER STORED IN $36,37
                0654 ;     AND THE ROM ROUTINE PRINTS THE NUMBER AT THE NEXT
                0655 ;     CURSOR POSITION ON SCREEN.  THEN WE PRINT A BLANK
```

```
                 0656 ;    SPACE AND RETURN TO LINE 610 TO CONTINUE ON WITH
                 0657 ;    THE MAIN LOOP AND FIND MORE MATCHES.
                 0658 ;
03C5- 20 7F CF   0660 PRINTOUT   JSR PLINE            ; ROM ROUTINE PRINTS
                 0661            ; A LINE NUMBER FROM THE VALUES FOUND
                 0662            ; IN "FOUND" ($36,37).
03C8- A9 20      0670            LDA #$20             ;  PRINT A BLANK
03CA- 20 D2 FF   0680            JSR PRINT            ; SPACE BETWEEN #S
03CD- 60         0690            RTS
                 0691 ;--------------------------------------------------
                 0692 ;
                 0700            .EN


 ---  LABEL FILE:  ---


BASIC =0400            COMPARE =03A9          END =0375
FOUND =0036            GO.ON =0376            L1L =00BA
L2L =00BC              LOOP =039A             PERFECT =03B7
PLINE =CF7F            PRINT =FFD2            PRINTOUT =03C5
READLINE =036A         SAME =03A7             STOPLINE =03BA
```

character and to print a line number, and the RAM where it's safe to store the ML program itself. In other words, change the defined variables between lines 20 and 100 in Program 8-1 and you can use the program on another computer.

We will build our ML program in pieces and then tie them all together at the end. The first phase, as always, is the initialization. We set up the variables and fill in the pointers. Lines 20 and 30 define two, two-byte zero page pointers. L1L is going to point at the address of the BASIC line we are currently searching through. L2L points to the starting address of the line following it.

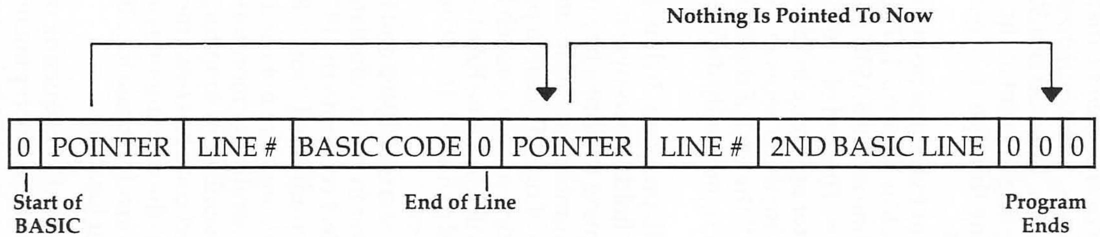
Microsoft BASIC stores four important bytes just prior to the start of the code in a BASIC line. Take a look at Figure 8-1. The first two bytes contain the address of the next line in the BASIC program. The second two bytes hold the line number. The end of a BASIC line is signaled by a zero. Zero does not stand for anything in the ASCII code or for any BASIC command. If there are three zeros in a row, this means that we have located the ''top,'' the end of the BASIC program. (The structure of Atari BASIC is significantly different. See Figure 8-2.)

But back to our examination of the ML program. In line 40 is a definition of the zero page location which holds a two-byte number that Microsoft BASIC looks at when it is going to print a line number on the screen. We will want to store line numbers in this location as we come upon them during the execution of our ML search program. Each line number will temporarily sit waiting in case a match is found. If a match is found, the program will JSR to the BASIC ROM routine we're calling ''PLINE,'' as defined in line 70. It will need the ''current line number'' to print to the screen.

Line 50 establishes that BASIC RAM starts at $0400 and line 60 gives the address of the ''print the character in the accumulator'' ROM routine. Line 100 says to put the object code into the PET's (all BASIC versions) second cassette buffer, a traditional ''safe'' RAM area to store short ML programs. These *safe areas* are not used by BASIC, the operating system (OS), or, generally, by monitors or assemblers. If you are working with an assembler or monitor, however, and keep finding that your object code has been messed up — suspect that your ML creating program (the monitor or assembler) is using part of your ''safe'' place. They consider it safe too. If this should happen, you'll have to find a better location.

Refer to Program 8-1 to follow the logic of constructing our Microsoft search program. The search is initiated by typing in line zero followed by the item we want to locate. It might be that we are interested in removing all REM statements from a program to shorten it. We would type 0:REM and hit RETURN to enter this into the BASIC program. Then we would start the search by a SYS to the

Figure 8-1. A BASIC Program's Structure.

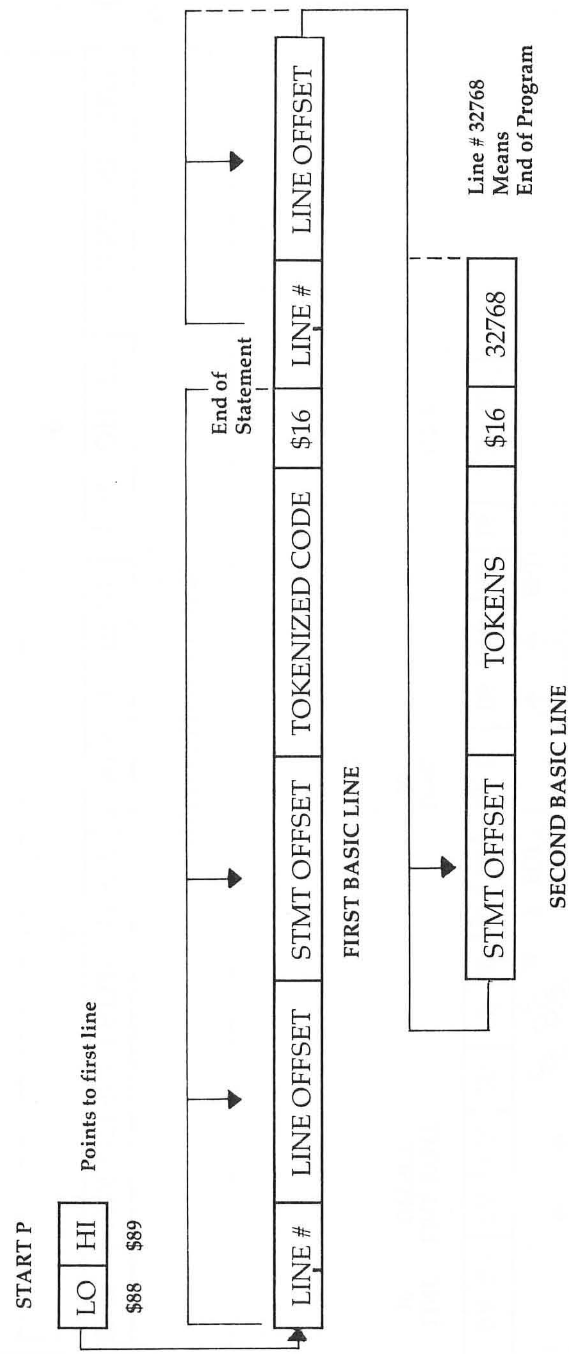


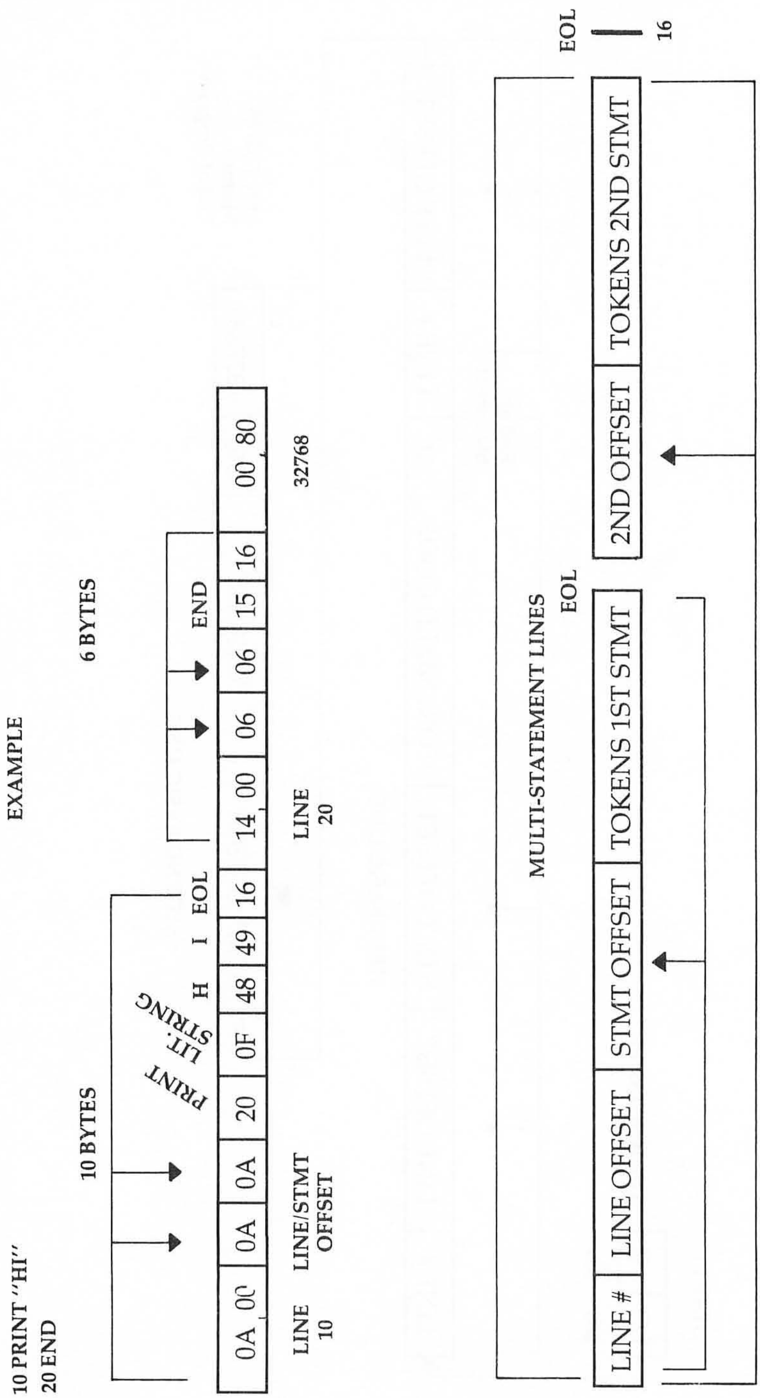

```
10 PRINT"HI"
20 END

          0400                              040B            0411
          00 0B  04  0A  00  99  22  48  49  22  00  11  04  14  00  80  00  00  00
                    LINE     ?  "    H  I  "                LINE  END
                     10                                     20
```

Figure 8-2. Atari BASIC Structure.

Single Statement Lines




10 PRINT "HI"
20 END
EXAMPLE
10 BYTES
6 BYTES
PRINT
LIT. STRING
H
I
EOL
END
0A 00
0A
0A
20
0F
48
49
16
14 00
06
06
15
16
00 80
LINE 10
LINE/STMT OFFSET
LINE 20
32768
MULTI-STATEMENT LINES
LINE #
LINE OFFSET
STMT OFFSET
TOKENS 1ST STMT
EOL
2ND OFFSET
TOKENS 2ND STMT
EOL
16

starting address of the ML program. In the PET 4.0 version of Program 8-1, it would be SYS 864 (hex $0360).

By entering the ''sample'' string or command into the BASIC program as line zero, we solve two problems. First, if it is a string, it will be stored as the ASCII code for that string, just as BASIC stores strings. If it is a keyword like REM, it will be translated into the ''tokenized,'' one-byte representation of the keyword, just as BASIC stores keywords. The second problem this solves is that our sample is located in a known area of RAM. By looking at Figure 8-1, you can tell that the sample's starting address is always the start of BASIC plus six. In Program 8-1 that means 0406 (see line 550).

Set Up The Pointers

We will have to get the address of the next line in the BASIC program we are searching. And then we need to store it while we look through the current line. The way that BASIC lines are arranged, we come upon the link to the next line's address and the line number before we see any BASIC code itself. Therefore, the first order of business is to put the address of the next line into L1L. Lines 150 through 180 take the link found in start-of-BASIC RAM (plus one) and move it to the storage pointer ''L1L.''

Next, lines 190 to 250 check to see if we have reached the end of the BASIC program. It would be the end if we had found two zeros in a row as the pointer to the next line's address. If it is the end, the RTS sends us back to BASIC mode.

The subroutine in lines 260 through 440 saves the pointer to the following line's address and also the current line number. Note the double-byte addition in lines 390-440. Recall that we CLC before any addition. If adding four to the LSB (line 400) results in a carry, we want to be sure that the MSB goes up by one during the add-with-carry in line 430. It might seem to make no sense to add a zero in that line. What's the point? The addition is *with carry;* in other words, if the carry flag has been set up by the addition of four to the LSB in line 400, then the MSB will go up by one. The carry will make this happen.

First Characters

It's better to just compare the first character in a word against each byte in the searched memory than to try to compare the entire sample word. If you are looking for MEM, you don't want to stop at each byte in memory and see if M-E-M starts there. Just look for M's. When you come upon a M, *then* go through the full string comparison. If line 490 finds a first-character match, it transfers the program to ''SAME'' (line 520) which will do the entire comparison. On the other hand, if the routine starting at line 451 comes upon a zero (line 470), it knows that the BASIC line has ended (they all end with zero). It then goes down to ''STOPLINE'' (line 610) which puts the ''next line'' address pointer into the "current line" pointer and the whole process of reading a new BASIC line begins anew.

If, however, a perfect match was found (line 560 found a zero at the end of the 0:REM line, showing that we had come to the end of the sample string) — we go to "PERFECT" and it makes a JSR to print out the line number (line 660). That subroutine bounces back (RTS) to "STOPLINE" which replaces the "current line" (L1L) pointer with the "next line" pointer (L2L). Then we JMP back to "READLINE" which, once again, pays very close attention to zeros to see if the whole BASIC program has ended with double zeros. We have returned to the start of the main loop of this ML program.

This sounds more complicated than it is. If you've followed this so far, you can see that there is enormous flexibility in constructing ML programs. If you want to put the "STOPLINE" segment earlier than the "SAME" subroutine — go ahead. It is quite common to see a structure like this:

```
INITIALIZATION
LDA #15
STA $83
MAIN LOOP
START JSR 1
      JSR 2
      JSR 3
BEQ START       (until some index runs out)
RTS             (to BASIC)
SUBROUTINES
1
2               (each ends with RTS back to the MAIN LOOP)
3
DATA
Table 1
Table 2
Table 3
```

The Atari FIND Utility

The second source listing, Program 8-2, adds a FIND command to Atari BASIC. You access it with the USR command. It is written to assemble in page six (1536 or $0600) and is an example of a full-blown assembly. You'll need the assembler/editor cartridge to type it in.

After you've entered it, enter "ASM" to assemble it into memory. After it is finished, use the SAVE command to store the object (executable ML) code on tape or disk. Use:

SAVE#C:>0600,067E for tape
SAVE#D:FIND.OBJ<0600 067E for disk

You can then put the BASIC cartridge in and enter the machine language with the BASIC loader program, or with the L command of DOS.

Using FIND from BASIC is simple. Say you want to search a master string, A$ for the substring "hello". If B$ contains "hello", the USR call would look like:

POS = USR (1536,ADR(A$),LEN(A$),ADR(B$),LEN(B$))

POS will contain the position of the match. It will be a memory location within the ADRress of A$. To get the character position within A$, just use POS-ADR(A$) + 1. If the substring (B$) is not found, POS will be zero.

It's easy to add commands like this to Atari BASIC. Also see "Getting The Most Out Of USR" in the November 1982 issue of *COMPUTE!* Magazine (p. 100).

64, Apple, & VIC Versions

Versions of the search routine for the Commodore 64 and VIC-20 and the Apple II are provided as BASIC loader programs. Remember from Chapter 2 that a loader is a BASIC program which POKEs a machine language program (stored in DATA statements) into memory. Once you have entered and run the BASIC programs, you can examine the ML programs using a disassembler. (See Appendix D.)

These versions are similar to the PET Version outlined in Program 8-1. The characters to be searched for are typed in line 0. To start the search in the 64 version (Program 8-3), type SYS 40800. Use CALL 768 to activate the Apple version (Program 8-4). The VIC version (Program 8-5) is activated with SYS 828.

As your skills improve, you will likely begin to appreciate, and finally embrace, the extraordinary freedom that ML confers on the programmer. Learning it can seem fraught with obscurity and rules. It can even look menacing. But there are flights you will soon be taking through your computer. Work at it. Try things. Learn how to find your errors. It's not circular — there will be considerable advances in your understanding. One day, you might be able to sit down and say that you can combine BASIC with ML and do pretty much anything you want to do with your machine.

Program 8-2.

```
00CB            0100 ;==========================;
                0110 ; FIND Utility             ;
                0120 ; Substring Search         ;
                0130 ; for Atari BASIC          ;
                0140 ; Completely relocatable ;
                0150 ;==========================;
                0160 ;
                0170 ;
                0180 ;Variables in zero page for speed
                0190 ;
00CB            0200 SADRL      =$CB        ;Address
00CC            0210 SADRH      =$CC        ;of search
00CD            0220 SLENL      =$CD        ;Length of
00CE            0230 SLENH      =$CE        ;search space
                0240 ;
00CF            0250 FNDL       =$CF        ;Search address
00D0            0260 FNDH       =$D0        ;and
00D1            0270 FNDLEN     =$D1        ;length
                0280 ;
00D2            0290 FIRSTCHAR  =$D2
00D3            0300 SINDEX     =$D3
00D4            0310 FR0        =$D4        ;Return
00D6            0320 FINDEX     =$D6        ;Source index
00D7            0330 TADRL      =$D7        ;Temp addr
```

```
ØØD8          Ø34Ø TADRH       =$D8
ØØD9          Ø35Ø ENDLOOP     =$D9
              Ø36Ø ;
              Ø37Ø ;Syntax documentation
              Ø38Ø ;
              Ø39Ø ;FIND:Find Text
              Ø4ØØ ;X=USR(FIND,A,B,C,D)
              Ø41Ø ;FIND:Address of utility (1536)
              Ø42Ø ;A: Where to start search
              Ø43Ø ;B: Where to quit searching
              Ø44Ø ;C: Search string address
              Ø45Ø ;D: Length of search string
              Ø46Ø ;X: Position found (=Ø if no match)
ØØØØ          Ø47Ø        *=   $Ø6ØØ
              Ø48Ø ;-----------------------------------
              Ø49Ø ;This portion sets up the parameters
              Ø5ØØ ;for the search by pulling the values
              Ø51Ø ;passed by BASIC off the stack
              Ø52Ø ;
              Ø53Ø FIND
Ø6ØØ 68       Ø54Ø        PLA            ;Count byte
Ø6Ø1 68       Ø55Ø        PLA            ;hi byte, Source start
Ø6Ø2 85CC     Ø56Ø        STA  SADRH
Ø6Ø4 68       Ø57Ø        PLA            ;lo byte, Source start
Ø6Ø5 85CB     Ø58Ø        STA  SADRL
Ø6Ø7 68       Ø59Ø        PLA            ;hi byte, Source end
```

```
0608 85CE          STA  SLENH
060A 68            PLA            ;lo byte, Source end
060B 85CD          STA  SLENL
060D 68            PLA            ;hi byte, Search string
060E 85D0          STA  FNDH
0610 68            PLA            ;lo byte, Search string
0611 85CF          STA  FNDL
0613 68            PLA            ;hi byte, Search length
              ;Ignore it
0614 68            PLA            ;lo byte, Search length
0615 85D1          STA  FNDLEN
              ;
              ;-------------------------------
              ;This is the main loop.  We
              ;search through the search space
              ;looking for the first character
              ;of the search string.  We
              ;look through entire 256-byte
              ;blocks.  If the first character
              ;is found, we exit to a full
              ;string comparison routine.
              ;
              ;If the string is never found,
              ;we just return a zero to BASIC
              ;
```

```
Ø617 AØØØ   Ø85Ø            LDY  #Ø
Ø619 B1CF   Ø86Ø            LDA  (FNDL),Y    ;Set up first
Ø61B 85D2   Ø87Ø            STA  FIRSTCHAR   ;comparison
            Ø88Ø ;
Ø61D A6CE   Ø89Ø            LDX  SLENH       ;Less than 255
Ø61F FØ18   Ø9ØØ            BEQ  SHORT       ;bytes?
            Ø91Ø NXTSRCH
Ø621 A9FF   Ø92Ø            LDA  #255        ;Select end
            Ø93Ø SEARCH2
Ø623 85D9   Ø94Ø            STA  ENDLOOP
Ø625 AØØØ   Ø95Ø            LDY  #Ø
            Ø96Ø SEARCHLOOP
Ø627 B1CB   Ø97Ø            LDA  (SADRL),Y
Ø629 C5D2   Ø98Ø            CMP  FIRSTCHAR   ;Found a match?
Ø62B FØ17   Ø99Ø            BEQ  FOUND1      ;yes
            1ØØØ NOTFOUND
Ø62D C8     1Ø1Ø            INY             ;no
Ø62E C4D9   1Ø2Ø            CPY  ENDLOOP
Ø63Ø DØF5   1Ø3Ø            BNE  SEARCHLOOP  ;continue
            1Ø4Ø ;
Ø632 E6CC   1Ø5Ø            INC  SADRH       ;Next block
Ø634 CA     1Ø6Ø            DEX             ;Done?
Ø635 3ØØ6   1Ø7Ø            BMI  EXIT        ;yes
Ø637 DØE8   1Ø8Ø            BNE  NXTSRCH     ;nope
            1Ø9Ø SHORT
Ø639 A5CD   11ØØ            LDA  SLENL       ;Set up last
```

```
Ø63B DØE6   111Ø         BNE  SEARCH2      ;scan
            112Ø EXIT
Ø63D A9ØØ   113Ø         LDA  #Ø           ;return
Ø63F 85D4   114Ø         STA  FRØ          ;=Ø
Ø641 85D5   115Ø         STA  FRØ+1        ;no string
Ø643 6Ø     116Ø         RTS               ;found
            117Ø ;
            118Ø ;-------------------------------
            119Ø ;Here is where we check for a
            12ØØ ;full match, starting with the
            121Ø ;second character of the search string
            122Ø ;We have to use two "pseudo" registers
            123Ø ;in memory, since the same Y register
            124Ø ;is needed to access both areas of memory
            125Ø ;(search space and search string)
            126Ø ;
            127Ø FOUND1
Ø644 84D4   128Ø         STY  FRØ          ;Save Y
Ø646 84D3   129Ø         STY  SINDEX       ;Source index
Ø648 AØØ1   13ØØ         LDY  #1
Ø64A 84D6   131Ø         STY  FINDEX       ;Find index
            132Ø ;
            133Ø ;We use a temporary address, since we don't want
            134Ø ;to change the address in SADR (so we can continue the
            135Ø ;search if no match found)
            136Ø ;
```

```
Ø64C A5CB   137Ø          LDA  SADRL        ;Copy to
Ø64E 85D7   138Ø          STA  TADRL        ;temp addr
Ø65Ø A5CC   139Ø          LDA  SADRH
Ø652 85D8   14ØØ          STA  TADRH
            141Ø ;
            142Ø CONTSRCH
            143Ø ;
            144Ø ;As long as each character matches, we
            145Ø ;continue to compare until we get a failed comparison
            146Ø ;or reach the end of the search string,
            147Ø ;which indicates a match.
            148Ø ;
Ø654 A4D6   149Ø          LDY  FINDEX
Ø656 C4D1   15ØØ          CPY  FNDLEN       ;Past end?
Ø658 FØ16   151Ø          BEQ  FOUND2       ;yes-match!
Ø65A B1CF   152Ø          LDA  (FNDL),Y     ;Character n
Ø65C E6D6   153Ø          INC  FINDEX       ;no, increment
Ø65E A4D3   154Ø          LDY  SINDEX       ;Compare to
Ø66Ø C8     155Ø          INY              ;source
Ø661 DØØ2   156Ø          BNE  SKIPINC      ;Hit page bound?
Ø663 E6D8   157Ø          INC  TADRH
            158Ø SKIPINC
Ø665 84D3   159Ø          STY  SINDEX       ;Update
Ø667 D1D7   16ØØ          CMP  (TADRL),Y    ;equal so far?
Ø669 FØE9   161Ø          BEQ  CONTSRCH     ;yes, continue
            162Ø ;Comparison failure,
```

```
                1630 ;Return to main loop
066B A4D4       1640        LDY  FR0
066D 18         1650        CLC              ;Used in place
066E 90BD       1660        BCC  NOTFOUND     ;of JMP (relocatable)
                1670 ;
                1680 ;Match!
                1690 ;Return address in FR0 to BASIC
                1700 FOUND2
0670 18         1710        CLC
0671 A5D4       1720        LDA  FR0
0673 65CB       1730        ADC  SADRL
0675 85D4       1740        STA  FR0
0677 A5CC       1750        LDA  SADRH
0679 6900       1760        ADC  #0
067B 85D5       1770        STA  FR0+1
067D 60         1780        RTS
                1790
067E            1800        .END

=00CB SADRL      =00CC SADRH      =00CD SLENL        =00CE SLENH
=00CF FNDL       =00D0 FNDH       =00D1 FNDLEN       =00D2 FIRSTCHAR
=00D3 SINDEX     =00D4 FR0        =00D6 FINDEX       =00D7 TADRL
=00D8 TADRH      =00D9 ENDLOOP     0600 FIND          0639 SHORT
 0621 NXTSRCH     0623 SEARCH2     0627 SEARCHLOOP    0644 FOUND1
 062D NOTFOUND    063D EXIT        0654 CONTSRCH      0670 FOUND2
 0665 SKIPINC
```

Program 8-3. 64 Search BASIC Loader.

```
799 X=PEEK(55):POKE55,X-1:REM PROTECT ML
800 FOR ADRES=40800TO40913:READ DATTA:
    POKE ADRES,DATTA:NEXT ADRES
900 PRINT"SYS40800 TO ACTIVATE"
4096 DATA 162, 0, 173, 1, 8, 133
4102 DATA 165, 173, 2, 8, 133, 166
4108 DATA 160, 0, 177, 165, 208, 6
4114 DATA 200, 177, 165, 208, 1, 96
4120 DATA 160, 0, 177, 165, 141, 167
4126 DATA 0, 200, 177, 165, 141, 168
4132 DATA 0, 200, 177, 165, 133, 57
4138 DATA 200, 177, 165, 133, 58, 165
4144 DATA 165, 24, 105, 4, 133, 165
4150 DATA 165, 166, 105, 0, 133, 166
4156 DATA 160, 0, 177, 165, 240, 28
4162 DATA 205, 6, 8, 240, 4, 200
4168 DATA 76, 158, 159, 162, 0, 232
4174 DATA 200, 189, 6, 8, 240, 7
4180 DATA 209, 165, 240, 245, 76, 158
4186 DATA 159, 32, 201, 159, 165, 167
4192 DATA 133, 165, 165, 168, 133, 166
4198 DATA 76, 108, 159, 32, 201, 189
4204 DATA 169, 32, 32, 210, 255, 96
READY.
```

Program 8-4. Apple Version.

```
700 FOR AD=768TO900: READ DA:POKE A
    D,DA:NEXT AD
768 DATA169,76,141,245,3,169
774 DATA16,141,246,3,169,3
780 DATA141,247,3,96,162,0
786 DATA173,1,8,133,1,173
792 DATA2,8,133,2,160,0
798 DATA177,1,208,6,200,177
804 DATA1,208,1,96,160,0
810 DATA177,1,133,3,200,177
816 DATA1,133,4,200,177,1
822 DATA133,117,200,177,1,133
```

```
828 DATA118,165,1,24,105,4
834 DATA133,1,165,2,105,0
840 DATA133,2,160,0,177,1
846 DATA240,28,205,6,8,240
852 DATA4,200,76,76,3,162
858 DATA0,232,200,189,6,8
864 DATA240,7,209,1,240,245
870 DATA76,76,3,76,119,3
876 DATA165,3,133,1,165,4
882 DATA133,2,76,28,3,169
888 DATA163,32,237,253,32,32
894 DATA237,169,160,32,237,253
900 DATA76,108,3
```

Program 8-5. VIC-20 Search BASIC Loader.

```
800 FOR ADRES=828TO941:READ DATTA:POKE ADR
    ES,DATTA:NEXT ADRES
810 PRINT"SYS 828 TO ACTIVATE"
828 DATA 162, 0, 173, 1, 16, 133
834 DATA 187, 173, 2, 16, 133, 188
840 DATA 160, 0, 177, 187, 208, 6
846 DATA 200, 177, 187, 208, 1, 96
852 DATA 160, 0, 177, 187, 141, 190
858 DATA 0, 200, 177, 187, 141, 191
864 DATA 0, 200, 177, 187, 133, 57
870 DATA 200, 177, 187, 133, 58, 165
876 DATA 187, 24, 105, 4, 133, 187
882 DATA 165, 188, 105, 0, 133, 188
888 DATA 160, 0, 177, 187, 240, 28
894 DATA 205, 6, 16, 240, 4, 200
900 DATA 76, 122, 3, 162, 0, 232
906 DATA 200, 189, 6, 16, 240, 7
912 DATA 209, 187, 240, 245, 76, 122
918 DATA 3, 32, 165, 3, 165, 190
924 DATA 133, 187, 165, 191, 133, 188
930 DATA 76, 72, 3, 32, 194, 221
936 DATA 169, 32, 32, 210, 255, 96
```

9

ML Equivalents Of BASIC Commands

What follows is a small dictionary, arranged alphabetically, of the major BASIC commands. If you need to accomplish something in ML — TAB for example — look it up in this chapter to see one way of doing it in ML. Often, because ML is so much freer than BASIC, there will be several ways to go about a given task. Of these choices, one might work faster, one might take up less memory, and one might be easier to program and understand. When faced with this choice, I have selected example routines for this chapter which are easier to program and understand. At ML speeds, and with increasingly inexpensive RAM memory available, it will be rare that you will need to opt for velocity or memory efficiency.

CLR

In BASIC, this clears all variables. Its primary effect is to reset pointers. It is a somewhat abbreviated form of NEW since it does not "blank out" your program, as NEW does.

We might think of CLR, in ML, as the *initialization* routine which erases (zeros) the memory locations you've set aside to hold your ML flags, pointers, counters, etc. Before your program RUNs, you may want to be sure that some of these "variables" are set to zero. If they are in different places in memory, you will need to zero them individually:

2000 LDA # 0
2002 STA 1990 (put zero into one of the "variables")
2005 STA 1994 (continue putting zero into each byte which needs to be initialized)

On the other hand, maybe you've got your tables, flags, etc., all lined up together somewhere in a *data table* at the start or end of your ML program. It's a good idea. If your table is in one chunk of RAM, say from 1985 to 1999, then you can use a loop to zero them out:

2000 LDA # 0
2002 LDY # 15 (Y will be the counter. There are 15 bytes to zero out in this example.)

2004 STA 1985,Y (the lowest of the 15 bytes)
2007 DEY
2008 BNE 2004 (let Y count down to zero, BNEing until Y is zero, then the Branch if Not Equal will let the program fall through to the next instruction at 2010)

CONT

This word allows your program to pick up where it left off after a STOP command (or after hitting the system break key). You might want to look at the discussion of STOP, below. In ML, you can't usually get a running program to stop with the BREAK (or STOP) key. If you like, you could write a subroutine which checks to see if a particular key is being held down on the keyboard and, if it is, BRK:

3000 LDA 96 (or whatever your map says is the ''key currently depressed'' location for your machine)
3002 CMP # 13 (this is likely to be the RETURN key on your machine, but you'll want CMP here to the value that appears in the ''currently pressed'' byte for the key you select as your STOP key. It could be any key. If you want to use ''A'' for your ''stop'' key, try CMP #65.)
3004 BNE 3007 (if it's not your target key, jump to RTS)
3006 BRK (if it *is* the target, BRK)
3007 RTS (back to the routine which called this subroutine)

The 6502 places the Program Counter (plus two) on the stack after a BRK.

A close analogy to BASIC is the placement of BRK within ML code for a STOP and then typing .G or GO or RUN — whatever your monitor recognizes as the signal to start execution of an ML program — to CONT.

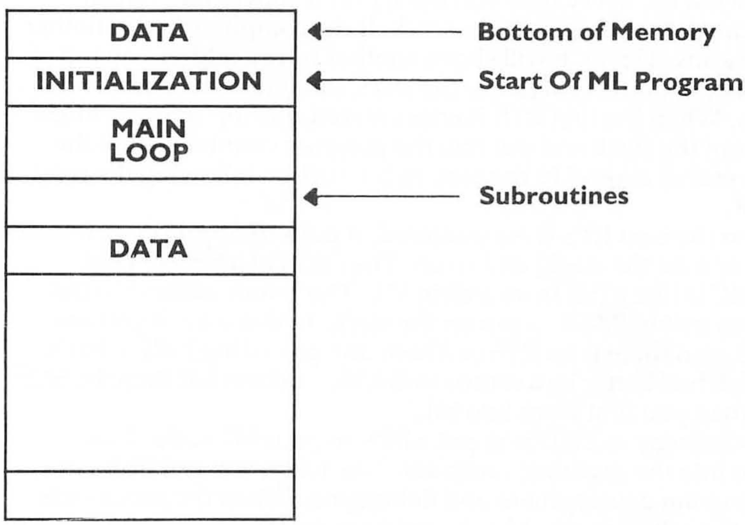
DATA

In BASIC, DATA announces that the items following the word DATA are to be considered pieces of information (as opposed to being thought of as parts of the program). That is, the program will probably *use* this data, but the data are not BASIC commands. In ML, such a zone of ''non-program'' is called a *table.* It is unique only in that the program counter never starts trying to run through a table to carry out instructions. Program control is never transferred to a table since there are no meaningful instructions inside a table. Likewise, BASIC slides right over its DATA lines.

To keep things simple, tables of data are usually stored together either below the program or above it in memory. (See Figure 9-1.)

From within the program, tables can be used to print messages to the screen, update or examine flags, etc. If you disassemble your BASIC in ROM, you'll find the words STOP, RUN, LIST, and so forth, gathered together in a table. You can suspect a data table when your disassembler starts giving lots of error messages. It cannot find groups of meaningful opcodes within tables.

Figure 9-1. Typical ML program organization with data tables — one at top or bottom of program.



DIM

With its automatic string handling, array management, and error messages, BASIC makes life easy for the programmer. The price you pay for this "hand-holding" is that a program is slow when it's RUN. In ML, the DIMensioning of space in memory for variables is not explicitly handled by the computer. You must make a note that you are setting aside memory from 6000 to 6500, or whatever, to hold variables. It helps to make a simple map of this "dimensioned" memory so you know where permanent strings, constants, variable strings, and variables, flags, etc., are *within* the dimensioned zone.

A particular chunk of memory (where, and how much, is up to you) is set aside, that's all. You don't write any instructions in ML to set aside the memory; you just jot it down so you won't later use the reserved space for some other purpose. Managing memory is left up to you. It's not difficult, but it *is* your responsibility.

END

There are several ways to make a graceful exit from ML programs. You can look for the ''warm start'' address on your particular computer (in the map of its BASIC locations) and JMP to that address. Or you can go to the ''cold start'' address. This results in the computer resetting itself as if you had turned the power off and then back on again.

If you went into the ML *from* BASIC (with a USR or SYS), you can return to BASIC with an RTS. Recall that every JSR matches up with its own RTS. Every time you use a JSR, it shoves its ''return here'' address onto the top of the stack. If the computer finds another JSR (before any RTS's), it will shove another return address on top of the first one. So, after two JRS's, the stack contains two return addresses. When the first RTS is encountered, the top return address is lifted from the stack and put into the program counter so that the program returns control to the current instruction following the most recent JSR.

When the next RTS is encountered, it pulls *its* appropriate return (waiting for it on the stack) and so on. The effect of a SYS or USR from BASIC is like a JSR from within ML. The return address to the correct spot *within BASIC* is put on the stack. In this way, if you are within ML and there is an RTS (without any preceding JSR), what's on the stack had better be a return-to-BASIC address left there by SYS or USR when you first went into ML.

Another way to END is to put a BRK in your ML code. This drops you into the machine's monitor. Normally, you put BRKs in during program development and debugging. When the program is finished, though, you would not want to make this ungraceful exit any more than you would want to end a BASIC program with STOP.

In fact, many ML programs, if they stand alone and are not part of a larger BASIC program, never END at all! They are an endless loop. The main loop just keeps cycling over and over. A game will not end until you turn off the power. After each game, you see the score and are asked to press a key when you are ready for the next game. Arcade games which cost a quarter will ask for another quarter, but they don't end. They go into ''attract mode.'' The game graphics are left running on screen to interest new customers.

An ML word processor will cycle through its main loop, waiting for keys to be pressed, words to be written, format or disk instructions to be given. Here, too, it is common to find that the word processor takes over the machine, and you cannot stop it without turning the computer off. Among other things, such an endless loop protects software from being easily pirated. Since it takes control of the machine, how is someone going to save it or examine it once it's in RAM? Some such programs are ''auto-booting'' in that they cannot be loaded without starting themselves running.

BASIC, itself a massive ML program, also loops endlessly until you power down. When a program is RUNning, all sorts of things are happening. BASIC is an *interpreter,* which means that it must look up each word (like INT) it comes across during a RUN (interpreting it, or *translating* its meanings into machine-understandable JSRs). Then BASIC executes the correct sequence of ML actions from its collection of routines.

In contrast to BASIC RUNs, BASIC spends 99 percent of its time waiting for you to *program* with it. This waiting for you to press keys is its ''endless'' loop, a tight, small loop indeed. It would look like our ''which key is pressed?'' routine.

2000 LDA 96 (or wherever your machine's map shows that the ''which key down'' value is stored)
2002 CMP #255 (or whatever value is *normally* left in this address by default when no key is being pressed)
2004 BEQ 2000 (if it says ''no key down,'' cycle back and wait for one)

FOR-NEXT

Everyone has used ''delay loops'' in BASIC (FOR T=1 TO 1000: NEXT T). These are small loops, sometimes called do-nothing loops because nothing happens between the FOR and the NEXT except the passage of time. When you need to let the user read something on the screen, it's sometimes easier just to use a delay loop than to say ''When finished reading, press any key.''

In any case, you'll need to use delay loops in ML just to *slow ML itself down.* In a game, the ball can fly across the screen. It can get so fast, in fact, that you can't see it. It just ''appears'' when it bounces off a wall. And, of course, you'll need to use loops in many other situations. Loops of all kinds are fundamental programming techniques.

In ML, you don't have that convenient little counter (''T'' in the BASIC FOR/NEXT example above) which decides when to stop the loop. When *T* becomes 1000, go to the instructions beyond the word NEXT. Again, you must set up and check your *counter variable* by yourself.

If the loop is going to be smaller than 255 cycles, you can use the X register as the counter (Y is saved for the very useful *indirect indexed* addressing discussed in Chapter 4: LDA (96),Y). So, using X, you can count to 200 by:

2000 LDX #200 (or $C8 hex)
2002 DEX
2003 BNE 2002

For loops involving counters larger than 255, you'll need to use two bytes to count down, one going from 255 to zero and then clicking (like a gear) the other (more significant) byte. To count to 512:

```
2000 LDA # 2
2002 STA 0      (put the 2 into address zero, our MSB, Most
                 Significant Byte, counter)
2004 LDX #0     (set X to zero so that its first DEX will make it 255.
                 Further DEX's will count down again to zero,
                 when it will click the MSB down from 2 to 1 and
                 then finally 0)
2006 DEX
2007 BNE 2006
2009 DEC 0      (click the number in address zero down 1)
2011 BNE 2006
```

Here we used the X register as the LSB (least significant byte) and address zero as the MSB. We could use addresses zero and one to hold the MSB/LSB if we wanted. This is commonly useful because then address zero (or some available, two-byte space in zero page) can be used for LDA (0),Y. You would print a message to the screen using the combination of a zero page counter and LDA (zero page address),Y.

FOR-NEXT-STEP

Here you would just increase your counter (usually X or Y) more than once. To create FOR I=100 TO 1 STEP -2 you could use:

```
2000 LDX # 100
2002 DEX
2003 DEX
2004 BCC 2002
```

For larger numbers you create a counter which uses two bytes working together to keep count of the events. Following our example above for FOR-NEXT, we could translate FOR I=512 TO 0 STEP -2:

```
2000 LDA # 2
2002 STA 0      (this counts the MSB)
2004 LDX # 0    (X counts the LSB)
2006 DEX
2007 DEX        (here we click X down a second time, for -2)
2008 BNE 2006
2010 DEC 0
2012 BNE 2006
```

To count up, use the CoMPare instruction. FOR I=1 TO 50 STEP 3:

```
2000 LDX # 0
2002 INX
2003 INX
2004 INX
2005 CPX # 50
2007 BNE 2002
```

For larger STEP sizes, you can use a *nested loop* within the larger one. This would avoid a whole slew of INX's. To write the ML equivalent of FOR I=1 TO 50 STEP 10:

```
2000 LDX #0
2002 LDY #0
2004 INX
2005 INY
2006 CPY #10
2008 BNE 2004
2010 CPX #50
2012 BNE 2002
```

GET

Each computer model has its own "which key is being pressed?" address, where it holds the value of a character typed in from the keyboard. To GET, you create a very small loop which just keeps testing the first address in the buffer.

For Atari (in decimal):

```
2000 LDA 764  ("which key pressed" decimal address. In
               advanced assemblers, you could freely mix
               decimal with hex, but not in the Simple
               Assembler.)
2003 CMP #255 (when an FF value is in this address, it means
               that no key is pressed)
2005 BEQ 2000 (keep going back and looking until there is some
               key pressed)
```

For PET (Upgrade and 4.0) (in decimal)

```
2000 LDA 151 ("which key pressed" decimal address)
2003 CMP #255
2005 BEQ 2000
```

For PET (Original):

```
2000 LDA 515 ("which key pressed" decimal address)
2003 CMP #255
2005 BEQ 2000
```

For Apple II (hex):

2000 LDA C000 (''which key pressed'' — note: this is in hex)
2003 BPL 2000
2005 STA C010 (clears the keyboard)
2008 AND #7F (to give you the correct character value)

For VIC and 64 (decimal):

2000 LDA 197
2003 CMP #255
2008 BEQ 2000

The Commodore computers have a GET routine similar to the one illustrated by these examples, which is built in at $FFE4 which can be used for all ROM versions (all models of CBM) because it is a fixed JMP table which does not change address when new BASIC versions are introduced. See your BASIC's map for Print a Byte to the Screen, GET a Byte, and other routines in the Commodore Jump Tables. They start at $FFBD.

The examples above do not conform to PET BASIC's GET. In this version of BASIC, the computer does not ''wait'' for a character. If no key is being held down during a GET, the computer moves on and no GET takes place. In our ML GETs above, we loop *until* some character is actually pressed.

For most programming purposes, though, you want to wait until a key has actually been pressed. If your program is supposed to fly around doing things *until* a key is pressed, you might use the above routines without the loop structure. Just use a CMP to test for the particular key that would stop the routine and branch the program somewhere else when a particular key is pressed. How you utilize and construct a GET-type command in ML is up to you. You can, with ML's flexibility, make special adjustments to use the best kind of GET for each different application.

GOSUB

This is nearly identical to BASIC in ML. Use JSR $NNNN and you will go to a subroutine at address NNNN instead of a line number, as in BASIC. (''NNNN'' just means you can put any hex number in there you want to.) Some assemblers allow you to give ''labels,'' names to JSR to instead of addresses. The Simple Assembler does not allow labels. You are responsible (as with DATA tables, variables, etc.) for keeping a list on paper of your subroutine addresses *and the parameters involved.*

Parameters are the number or numbers handed to a subroutine to give it information it needs. Quite often, BASIC subroutines work with the variables already established within the BASIC program. In ML, though, managing variables is up to you. Subroutines are useful because they can perform tasks repeatedly without needing to be programmed into the body of the program each time the task is to be carried out. Beyond this, they can be *generalized* so that a single subroutine can act in a variety of ways, depending upon the variable (the parameter) which is passed to it.

A delay loop to slow up a program could be general in the sense that the amount of delay is handed to the subroutine each time. The delay can, in this way, be of differing durations, depending on what it gets as a parameter from the main routine. Let's say that we've decided to use address zero to pass parameters to subroutines. We could pass a delay of "five" cycles of the loop by:

```
                  2000 LDA # 5
The Main Program  2002 STA 0
                  2004 JSR 5000
                  . . .
The Subroutine    5000 DEC 0
                  5002 BEQ 5012  (if address zero has
                                  counted all the way down
                                  from five to zero, RTS back
                                  to the Main Program)
                  5004 LDY # 0
                  5006 DEY
                  5007 BNE 5006
                  5009 JMP 5000
                  5012 RTS
```

A delay which lasted twice as long as the above would merely require a single change: **2000 LDA # 10.**

GOTO

In ML, it's JMP. JMP is like JSR, except the address you leap away from is not saved anywhere. You jump, but cannot use an RTS to find your way back. A *conditional* branch would be CMP #0 BEQ 5000. The condition of equality is tested by BEQ, Branch if EQual. BNE tests a condition of inequality, Branch if Not Equal. Likewise, BCC (Branch if Carry is Clear) and the rest of these branches are testing conditions within the program.

GOTO and JMP do not depend on any conditions within the program, so they are *unconditional.* The question arises, when you use a GOTO: Why did you write a part of your program that you must *always* (unconditionally) jump over? GOTO and JMP are sometimes used to patch up a program, but, used without restraint, they can make your program hard to understand later. Nevertheless, JMP can many times be the best solution to a programming problem. In fact, it is hard to imagine ML programming without it.

One additional note about JMP: it makes a program non-relocatable. If you later need to move your whole ML program to a different part of memory, all the JMP's (and JSR's) need to be checked to see if they are pointing to addresses which are no longer correct (JMP or JSR into your BASIC ROM's will still be the same, but not those which are targeted to addresses *within* the ML program). This can be an important consideration if you are going to use an ML subroutine in other programs where the locations might well differ. Fully relocatable ML routines can be convenient if you like to program by drawing from a personal collection of solved problems.

2000 JMP 2005
2003 LDY #3
2005 LDA #5

If you moved this little program up to 5000, everything would survive intact and work correctly except the JMP 2005 at address 2000. It would still say to jump to 2005, but it should say to jump to 5005, after the move. You have to go through with a disassembly and check for all these incorrect JMP's. To make your programs more "relocatable," you can use a special trick with unconditional branching which *will* move without needing to be fixed:

2000 LDY #0
2002 BEQ 2005 (since we just loaded Y with a zero, this Branch-if-EQual-to-zero instruction will *always* be true and will always cause a pseudo-JMP)
2004 NOP
2005 LDA #5

This works because we set the Z flag. Then, when BEQ tests the zero flag, it will pass the test, it will find that flag "up" and will branch. If you load X, Y, or A with a zero, the zero flag goes up.

Various monitors and assemblers include a "move it" routine, which will take an ML program and relocate it somewhere else in memory for you. On the Apple, you can go into the monitor and type *5000 < 2000.2006M (although you will have to give the monitor these numbers in hex). The first number is the target address. The second and third are the start and end of the program you want to move.

On CBM computers, the built-in monitor (the VIC-20 and the Original 2001 ROM set do not have a built-in monitor) does not have a Move it command. However, it is easy to add a "monitor extension" program to the built-in monitor. *Supermon* and *Micromon* are such extensions. The format for Moveit in Commodore machines is .T 2000 2006 5000 (start and end of the program to be moved, followed by the target address). Again, these numbers must be in hex. The *T* stands for *transfer.*

The Atari Assembler Editor Cartridge follows a convention similar to Apple's: M 5000 < 2000,2006.

IF-THEN

This familiar and primary computing structure is accomplished in ML with the combination of CMP-BNE or any other conditional branch: BEQ, BCC, etc. Sometimes, the IF half isn't even necessary. Here's how it would look:

```
2000 LDA 57     (what's in address 57?)
2002 CMP #15    (is it 15?)
2004 BEQ 2013   (IF it is, branch up to 2013)
2006 LDA #10    (or ELSE, put a 10 into address 57)
2008 STA 57
2010 JMP 2017   (and jump over the THEN part)
2013 LDA #20    (THEN, put a 20 into address 57)
2015 STA 57
2017            (continue with the program . . .)
```

Often, though, your flags are already set by an action, making the CMP unnecessary. For example, if you want to branch to 2013 if the number in address 57 is zero, just LDA 57 BEQ 2013. This is because the act of loading the accumulator will affect the status register flags. You don't need to CMP #0 because the zero flag will be set if a zero was just loaded into the accumulator. It won't hurt anything to use a CMP, but you'll find many cases in ML programming where you can shorten and simplify your coding. As you gain experience, you will see these patterns and learn how and what affects the status register flags.

INPUT

This is a series of GETs, echoed to the screen as they are typed in, which end when the typist hits the RETURN key. The reason for the *echo* (the symbol for each key typed is reproduced on the screen) is that few people enjoy typing without seeing what they've typed. This also allows for error correction using cursor control keys or DELETE and INSERT keys. To handle all of these actions, an INPUT routine must be fairly complicated. We don't want, for example, the DELETE to become a character within the string. We want it to immediately act on the string being entered during the INPUT, to erase a mistake.

Our INPUT routine must be smart enough to know what to add to the string and what keys are intended only to modify it. Here is the basis for constructing your own ML INPUT. It simply receives a character from the keyboard, stores it in the screen RAM cells, and ends when the RETURN key is pressed. This version is for Upgrade and 4.0 CBM/PETs and we'll write it as a subroutine. That simply means that when the 13 (ASCII for carriage return) is encountered, we'll perform an RTS back to a point just following the main program address which JSRed to our INPUT routine:

```
5000 LDY #0          (Y will act here as an offset for storing the
                      characters to the screen as they come in)
5002 LDA 158         (this is the "number of keys in the keyboard buffer"
                      location. If it's zero, nothing has been typed yet)
5004 BNE 5002        (so we go back to 5002)
5006 LDA 623         (get the character from the keyboard buffer)
5009 CMP #13         (is it a carriage return?)
5011 BNE 5014        (if not, continue)
5013 RTS             (otherwise return to the main program)
5014 STA 32768,Y (echo it to the screen)
5017 INY
5018 LDA #0
5020 STA 158         (reset the "number of keys" counter to zero)
5022 JMP 5002        (continue looking for the next key)
```

This INPUT could be made much larger and more complex. As it stands, it will contain the string on the screen only. To save the string, you would need to read it from screen RAM and store it elsewhere where it will not be erased. Or, you could have it echo to the screen, but (also using Y as the offset) store it into some safe location where you are keeping string variables. The routine above does not make provisions for DELETE or INSERT either. The great freedom you have with ML is that you can redefine anything you want. You can *softkey:* define a key's meaning via software; have any key perform any task. You might use the $ key to DELETE.

Along with this freedom goes the responsibility for organizing, writing, and debugging these routines.

LET

Although this word is still available on most BASICs, it is a holdover from the early days of computing. It is supposed to remind you that a statement like LET NAME=NAME+4 is an *assignment* of a value to a variable, not an algebraic equation. The two numbers on either side of the "equals" sign, in BASIC, are not intended to be equal in the algebraic sense. Most people write NAME=NAME+4 without using LET. However, the *function* of LET applies to ML as well as to BASIC: we must assign values to variables.

In the Atari, VIC, and Apple, for example, where the address of the screen RAM can change depending on how much memory is in the computer, etc. — there has to be a place where we find out the starting address of screen RAM. Likewise, a program will sometimes require that you *assign* meanings to string variables, counters, and the like. This can be part of the initialization process, the tasks performed before the real program, your main routine, gets started. Or it can happen during the execution of the main loop. In either case, there has to be an ML way to establish, to *assign*, variables. This also means that you must have zones of memory set aside to hold these variables.

For strings, you can think of LET as the establishment of a location in memory. In our INPUT example above, we might have included an instruction which would have sent the characters from the keyboard to a table of strings as well as echoing them to the screen. If so, there would have to be a way of managing these strings. For a discussion on the two most common ways of dealing with strings in ML, see Chapter 6 under the subhead "Dealing With Strings."

In general, you will probably find that you program in ML using somewhat fewer variables than in BASIC. There are three reasons for this:

1. You will probably not write many programs in ML such as data bases where you manipulate hundreds of names, addresses, etc. It might be somewhat inefficient to create an entire data base management program, an inventory program for example, in ML. Keeping track of the variables would be a nightmare. An important benefit of ML is its speed of execution, but a drawback is that it slows programming down. So, for an inventory program, you could write the bulk of the program in BASIC and simply attach ML routines for sorting and searching tasks within the program.

2. Also, the variables in ML are often handled within a series of instructions (not held elsewhere as BASIC variables are). FOR I=1 TO 10: NEXT I becomes LDY #1, INY, CPY #10, BNE. Here, the BASIC variable is counted for you and stored outside the body of the program. The ML "variable," though, is counted by the program itself. ML has no *interpreter* which handles such things. If you want a loop, you must construct all of its components yourself.

3. In BASIC, it is tempting to assign values to variables at the start of the program and then to refer to them later by their variable names, as in: 10 BALL=79. Then, any time you want to PRINT the BALL to the screen, you could say, PRINT CHR$(BALL). Alternatively, you might define it this way in BASIC: 10 BALL$="0". In either case, your program will later refer to the word BALL. In this example we are assuming that the number 79 will place a ball character on your screen.

In ML we are not free to use variable names except when using a complicated, advanced assembler. With the Simple Assembler, you will find it easier just to LDA #79, STA (screen position) each time. Some people like to put the 79 into their zone of variables (that arbitrary area of memory set up at the start of a program to hold tables, counters, and important addresses). They can pull it out of that zone whenever it's needed. That is somewhat cumbersome, though, and slower. You would LDA 1015, STA (screen position), assuming you had put a 79 into this "ball" address earlier.

Obviously a value like BALL will remain the same throughout a program. A ball will look like a ball in your game, whatever else happens. So, it's not a true variable, it does not *vary*. It is constant. A true variable must be located in your "zone of variables," your variable *table*. It cannot be part of the body of your program itself (as in: LDA #79) because it will change. You don't know when writing your program what the variable will be. So you can't use *immediate mode* addressing because it might not be a #79. You have to LDA 1015 (or whatever) from within your table of variables.

Elsewhere in the program you have one or more STA 1015's or INC 1015's or some other manipulation of this address which keeps updating this variable. In effect, ML makes you responsible for setting aside areas which are safe to hold variables. What's more, you have to remember the addresses, and update the variables in those addresses whenever necessary. This is why it is so useful to keep a piece of paper next to you when you are writing ML. The paper lists the start and end addresses of the zone of variables, the table. You also write down the specific address of each variable as you write your program.

LIST

This is done via a *disassembler*. It will not have line numbers (though, again, advanced assembler-disassembler packages do have line numbers). Instead, you will see the address of each instruction in memory. You can look over your work and debug it by working with the disassembler, setting BRKs into problem areas, etc. See Appendix D.

LOAD

The method of saving and loading an ML program varies from computer to computer. Normally, you have several options which can include loading: from within the monitor, from BASIC, or even from an assembler. When you finish working on a program, or a piece of a program, on the Simple Assmbler you will be given the starting and ending addresses of your work. Using these, you can save to tape or disk in the manner appropriate to your computer. To LOAD, the simplest way is just to LOAD as if you were bringing in a BASIC program. Unfortunately, this only works on Commodore machines. You'll get your ML program, not a BASIC program, so it won't start at the normal starting address for BASIC unless you wrote and saved it at that address. You should type NEW after loading it, however, to reset some pointers in the computer. That will not NEW out the ML program.

To save from within the monitor on Commodore machines:

.S "PROGRAM NAME",01,NNNN,NNNN* (for tape)
.L "PROGRAM NAME",01 (for tape)
.S "0:PROGRAM NAME",08,NNNN,NNNN* (for disk)
.L "0:PROGRAM NAME",08 (for disk)

*You should add one to the hex number for the end of your program or the SAVE will clip off the last byte. If your program exists in RAM from $0300 to $0350, you save it like this: .S "PROGRAM NAME",01,0300,0351.

On the Apple, you must BLOAD from disk. On the Atari, if you have DOS you can use the "L" command from the DOS menu to LOAD in an ML program. If you don't, you need to use a short BASIC program that grabs in the bytes via a series of GETs:

```
10 OPEN#1,4,0,"C:"
20 GET#1,NN:GET#1,NN: REM DISCARD THE HEADER
30 GET#1,LO:GET#1,HI: REM START ADDRESS
40 START=LO+256*HI
50 GET#1,LO:GET#1,HI: REM ENDING ADDRESS
60 FIN=LO+256*HI
70 TRAP 100
80 FORI=START TO FIN: GET#1,A: POKEI,A:NEXTI
90 GOTO 30
100 END
```

Note: This will not work correctly if the START and FIN addresses overlap this BASIC program in memory. It would then load in on top of itself.

NEW

In Microsoft BASIC, this has the effect of resetting some pointers which make the machine think you are going to start over again. The next program line you type in will be put at the "start-of-a-BASIC-program" area of memory. Some computers, the Atari for example, even *wash* memory by filling it with zeros. There is no special command in ML for NEWing an area of memory, though some monitors have a "fill memory" option which will fill a block of memory as big as you want with whatever value you choose.

The reason that NEW is not found in ML is that you do not always write your programs in the same area of memory (as you do in BASIC), building up from some predictable address. You might have a subroutine floating up in high memory, another way down low, your table of variables just above the second subroutine, and your main program in the middle. Or you might not. We've been using

2000 as our starting address for many of the examples in this book and 5000 for subroutines, but this is entirely arbitrary.

To "NEW" in ML, just start assembling over the old program. Alternatively, you could just turn the power off and then back on again. This would, however, have the disadvantage of wiping out your assembler along with your program.

ON GOSUB

In BASIC, you are expecting to test values from among a group of numbers: 1,2,3,4,5 . . . . The value of X must fall within this narrow range: ON X GOSUB 100, 200, 300 . . . (X must be 1 or 2 or 3 here). In other words, you could not conveniently test for widely separated values of X (18, 55, 220). Some languages feature an improved form of ON GOSUB where you can test for any values. If your computer were testing the temperature of your bathwater:

```
CASE
                80 OF GOSUB HOT ENDOF
                100 OF GOSUB VERYHOT ENDOF
                120 OF GOSUB INTOLERABLE ENDOF
ENDCASE
```

ML permits you the greater freedom of the CASE structure. Using CMP, you can perform a *multiple branch* test:

```
2000 LDA 150      (get a value, perhaps input from the keyboard)
2002 CMP # 80
2004 BNE 2009
2006 JSR 5000     (where you would print "hot," following your
                   example of CASE)
2009 CMP # 100
2011 BNE 2016
2013 JSR 5020     (print "very hot")
2016 CMP # 120
2018 BNE 2023
2020 JSR 5030     (print "intolerable")
```

Since you are JSRing and then will be RTSing back to *within* the multiple branch test above, you will have to be sure that the subroutines up at 5000 do not change the value of the accumulator. If the accumulator started out with a value of 80 and, somehow, the subroutine at 5000 left a 100 in the accumulator, you would print "hot" and then also print "very hot." One way around this would be to put a zero into the accumulator before returning from each of the subroutines (LDA #0). This assumes that none of your tests, none of your cases, responds to a zero.

ON GOTO

This is more common in ML than the ON GOSUB structure above. It eliminates the need to worry about what is in the accumulator when you return from the subroutines. Instead of RTSing back, you jump back, *following all the branch tests.*

```
2000 LDA 150
2002 CMP # 80
2004 BNE 2009
2006 JMP 5000    (print "hot")
2009 CMP # 100
2011 BNE 2016
2013 JMP 5020    (print "very hot")
2016 CMP # 120
2018 BNE 2023
2020 JMP 5030    (print "intolerable")
2023             (all the subroutines JMP 2023 when they finish)
```

Instead of RTS, each of the subroutines will JMP back to 2023, which lets the program continue without accidentally "triggering" one of the other tests with something left in the accumulator during the execution of one of the subroutines.

PRINT

You *could* print out a message in the following way:

```
2000 LDY #0
2002 LDA #72       (use whatever your computer's screen POKE
                    value is for the letter "H")
2004 STA 32900,Y (an address on the screen)
2007 INY
2008 LDA #69       (the letter "E")
2010 STA 32900,Y
2013 INY
2014 LDA #76       (the letter "L")
2016 STA 32900,Y
2019 INY
2020 LDA #76       (the letter "L")
2022 STA 32900,Y
2025 INY
2026 LDA #79       (the letter "O")
2028 STA 32900,Y
```

But this is clearly a cumbersome, memory-eating way to go about it. In fact, it would be absurd to print out a long message this way. The most common ML method involves putting message strings into a data table and ending each message with a zero. Zero is never a printing character in computers (excepting Atari which cannot use the technique described here). To print the ASCII *number* zero, you use 48: LDA #48, STA 32900. So, zero itself can be used as a delimiter to let the printing routine know that you've finished the message. In a data table, we first put in the message ''hello''. Recall that you should substitute your own computer's screen POKE code:

```
1000 72 H
1001 69 E
1002 76 L
1003 76 L
1004 79 O
1005  0    (the delimiter, see Chapter 6)
1006 72 H
1007 73 I  (another message)
1008  0    (another delimiter)
```

Such a message table can be as long as you need; it holds all your messages and they can be used again and again:

```
2000 LDY #0
2002 LDA 1000,Y
2005 BEQ 2012      (if the zero flag is set, it must mean that we've
                    reached the delimiter, so we branch out of this
                    printing routine)
2005 STA 39000,Y (put it on the screen)
2008 INY
2009 JMP 2002      (go back and get the next letter in the message)
2012               (continue with the program.)
```

Had we wanted to print ''HI,'' the only change necessary would have been to put 1006 into the LDA at address 2003. To change the location on the screen that the message starts printing, we could just put some other address into 2006. The message table, then, is just a mass of words, separated by zeros, in RAM memory.

The easiest way to print to the screen, especially if your program will be doing a lot of printing, is to create a subroutine and use some bytes in zero page (addresses 0 to 255) to hold the address of the message and the screen location you want to send it to. This is one reason why hex numbers can be useful. To put an address into zero page, you will need to put it into two bytes. It's too big to fit into one byte. With two bytes together forming an address, the 6502 can address any location from $0000 to the top $FFFF. So, if the message is at decimal location 1000 like ''HELLO'' above, you should turn 1000 into a hex number. It's $03E8.

Then you split the hex number in two. The left two digits, $03, are the MSB (the most significant byte) and the right digits, $E8, make up the LSB (least significant byte). If you are going to put this target address into zero page at 56 (decimal):

```
2000 LDA #232     (LSB, in decimal)
2002 STA 56
2004 LDA #3       (MSB)
2006 STA 57
2008 JSR 5000     (printout subroutine)

5000 LDY #0
5002 LDA (56),Y
5004 BEQ 5013     (if zero, return from subroutine)
5006 STA 32900,Y (to screen)
5009 INY
5010 JMP 5002
5013 RTS
```

One drawback to the subroutine is that it will always print any messages to the same place on the screen. That 32900 (or whatever you use there) is frozen into your subroutine. Solution? Use another zero page pair of bytes to hold the screen address. Then, your calling routine sets up the message address, as above, but also sets up the screen address.

The Atari contains the address of the first byte of the screen addresses in zero page for you at decimal 88 and 89. You don't need to set up a screen address byte pair on the Atari. We are using the Apple II's low resolution screen for the examples in this book, so you will want to put 0 and 4 into the LSB and MSB respectively. The PET's screen is *always* located in a particular place, unlike the Atari, Apple, VIC, and 64 screen RAM locations which can move, so you can put a $00 and an $80 into LSB and MSB for PET. The following is in decimal:

```
2000 LDA #232    (LSB)
2002 STA 56      (set up message address)
2004 LDA #3      (MSB)
2006 STA 57
2008 LDA # 0     (LSB for PET and Apple)
2010 STA 58      (we'll just use the next two bytes in zero page
                  above our message address for the screen address)
2012 LDA # 4     (this is for Apple II; use 128 ($80) for PET)
2014 STA 59
2016 JSR 5000

5000 LDY #0
5002 LDA (56),Y
5004 BEQ 5013    (if zero, return from subroutine)
```

5006 STA (58),Y (to screen)
5009 INY
5010 JMP 5002
5013 RTS

For Atari: 5006 STA (88),Y. You have less flexibility because you will always be printing your messages to the first line on screen, using address 88 as your screen storage target. To be able to put the message anywhere on screen, Atari users will have to use some other zero page for the screen address, as we did for Apple II and PET above. Atari users would have to keep track of the ''cursor position'' for themselves in that case.

READ

There is no reason for a *read*ing of data in ML. Variables are not placed into ML ''DATA statements.'' They are entered into a table when you are programming. The purpose of READ, in BASIC, is to assign variable names to raw data or to take a group of data and move it somewhere, or to manipulate it into an array of variables. These things are handled by *you,* not by the computer, in ML programming.

If you need to access a piece of information, *you* set up the addresses of the datum and the target address to which you are moving it. See the ''PRINT'' routines above. As always, in ML you are expected to keep track of the locations of your variables. You keep a map of data locations, vectors, tables, and subroutine locations. A pad of paper is always next to you as you program in ML. It seems as if you would need many notes. In practice, an average program of say 1000 bytes could be mapped out and commented on, using only one sheet.

REM

You do this on a pad of paper, too. If you want to comment or make notes about your program — and it can be a necessary, valuable explanation of what's going on — you can disassemble some ML code like a BASIC LISTing. If you have a printer, you can make notes on the printed disassembly. If you don't have a printer, make notes on your pad to explain the purpose of each subroutine, the parameters it expects to get, and the results or changes it causes when it operates.

Complex, large assemblers often permit comments within the source code. As you program with them, you can include REMarks by typing a semicolon, or parentheses, or some other signal to the assembler to ignore the REMarks when it is assembling your program. In these assemblers, you are working much closer to the way you work in BASIC. Your remarks remain part of the source program and can be listed out and studied.

RETURN

RTS works the same way that RETURN does in BASIC: it takes you back to *just after* the JSR (GOSUB) that sent control of the program away from the main program and into a subroutine. JSR pushes, onto the stack, the address which immediately follows the JSR itself. That address then sits on the stack, waiting until the next RTS is encountered. When an RTS occurs, the address is pulled from the stack and placed into the *program counter.* This has the effect of transferring program control back to the instruction just after the JSR.

RUN

There are several ways to start an ML program. If you are taking off into ML from BASIC, you just use SYS or USR or CALL. They act just like JSR and will return control to BASIC, just like RETURN would, when there is an unmatched RTS in the ML program. By *unmatched* we mean the first RTS which is not part of a JSR/RTS pair. USR and SYS and CALL can be used either in *immediate mode* (directly from the keyboard) or from within a BASIC program as one of the BASIC commands.

USR is just like SYS and CALL except that you can "send" values from BASIC to ML by attaching them to the USR ( ) within the parentheses. In Microsoft BASIC (Apple, PET/CBM, etc.), you must set up the location of your target ML program in special USR addresses, before exiting BASIC via USR. For example, to "gosub" to an ML routine located at $0360 (hex), you want to put a $60 (hex) into address 1 and an 03 into address 2. The 03 is obvious, just POKE 2,3. Atari goes from BASIC to ML via USR. The USR's argument may place several parameters on the stack along with the "count," the number of parameters which were passed.

The hex 60 means that you would multiply 16×6, since the second column in hex is the "16's" column. So you would POKE 1, 96. Recall that we always set up ML addresses to be used by "indirect indexed addressing" (LDA (00),Y) by putting the LSB (least significant byte) first. To set up 0360, then, you first separate the hex number into its two bytes, 03 60. Then you translate them into decimal since we're in BASIC when we use USR: 3 96. Then you switch them so that they conform to the correct order for ML: LSB/MSB 96 3. Finally, you POKE them into memory locations 1 and 2.

If this seems rather complex, it is. In practice, Microsoft BASIC users rarely use USR. The number which is "passed" to ML from within the parentheses is put into the *floating point accumulator*. Following this you must JSR to FPINT, a BASIC ROM routine which converts a floating point value into an integer that you could work with in ML. As we mentioned, working with floating point arithmetic in ML is an arcane art. For most applications which must pass information from BASIC to ML, it is far easier to use ordinary "integer" numbers and just POKE them into some predetermined ML variable zone that you've set aside and noted on your workpad. Then just SYS to your ML routine, which will look into the set-aside, POKEd area when it needs the values from BASIC.

In Atari BASIC, USR works in a more simplified and more convenient way. For one thing, the target ML address is contained within the argument of the USR command: USR (address). This makes it nearly the exact parallel of BASIC's GOSUB. What's more, USR passes values from BASIC by putting them on the stack as a two-byte hex number. USR (address,X) does three things. 1. It sends program controi to the ML routine which starts at "address." 2. It pushes the number X onto the stack where it can be pulled out with PLA's. 3. Finally, it pushes the total *number* of passed values onto the stack. In this case, one value, X, was passed to ML. All of these actions are useful and make the Atari version of USR a more sensible way of GOSUBing from BASIC to ML.

If you are not going between BASIC and ML, you can start (RUN) your ML program from within your "monitor." The PET/CBM and the Apple have built-in monitor programs in their ROM chips. On the Atari, a monitor is available as part of a cartridge. On the "Original" PET/CBM (sometimes called BASIC 2.0), there is no built-in monitor. A cassette with a program called TIM (terminal interface monitor) can be LOADed, though, and used in the same way that the built-in versions are on later models. Neither the VIC nor the 64 has a built-in monitor.

To enter "monitor mode" (as opposed to the normal BASIC mode), you can type SYS 1024 or SYS 4 on the PET/CBM. These locations always contain a zero and, by "landing" on a zero in ML, you cause a BRK to take place. This displays the registers of your 6502 and prints a dot on the screen while waiting for your instructions to the monitor. To enter the monitor on Apple II, type CALL -151 and you will see an asterisk (instead of PET's period) as your prompt. From within Atari's Assembler Cartridge, you would type BUG to enter the equivalent of the Apple and PET monitor. The Atari will print the word DEBUG and then the cursor will wait for your next instruction.

To RUN an ML program, all five computers use the abbreviation G to indicate "goto and run" the hex address which follows the G. Unfortunately, the format of the ML RUN (G), as always, differs between machines. To run a program which starts at address $2000:

Apple II,	you type: 2000G	(8192 in decimal)
PET, VIC,64,	you type: G 2000	
Atari,	you type: G 2000	

One other difference: the Apple II expects to encounter an unmatched RTS to end the run and return control to the monitor. Put another way, it will think that your ML program is a subroutine and 2000G causes it to JSR to the subroutine at address (in hex) 2000. The Commodores and the Atari both look for a BRK instruction (00) to throw them back into monitor mode.

SAVE

When you SAVE a BASIC program, the computer handles it automatically. The starting address and the ending address of your program are calculated for you. In ML, you must know the start and end yourself and let the computer know. From the Apple II monitor, you type the starting and ending address of what you want saved, and then "W" for *write:*

2000.2010W (This is only for cassette and these commands are in hex. These addresses are 8192.8208, in decimal.)

From BASIC to disk use:

BSAVE Name,A,L (A = address, L = length)

On the VIC, 64, and PET, the format for SAVE is similar, but includes a filename:

.S "PROGRAM NAME",01,2000,2010 (the 01 is the "device number" of the tape player)

To save to disk, you must change the device number to 08 and start the filename with the number of the drive you are SAVEing to:

.S "0:NAME",08,2000,2010

(Always add one to the "finish" address; the example above saves from 2000 to 200F.)

With the Atari Assembler Cartridge, you:

SAVE#C:NAME < 2000,2010 (do this from the EDIT, not DEBUG, mode). The NAME is not required with cassette.

To write Atari *source code* to cassette, type: **SAVE#C.** For disk, type **SAVE#D:FILENAME.EXT** or use DOS.

STOP

BRK (or an RTS with no preceding JSR, on the Apple) throws you back into the monitor mode after running an ML program. This is most often used for debugging programs because you can set "breakpoints" in the same way that you would use STOP to examine variables when debugging a BASIC program.

String Handling

ASC

In BASIC, this will give you the number of the ASCII code which stands for the character you are testing. ?ASC("A") will result in a 65 being displayed. There is never any need for this in ML. If you are manipulating the character *A* in ML, you *are using ASCII already.* In other words, the letter *A is* 65 in ML programming. If your computer stores letters and other symbols in nonstandard ways (such as Commodore character codes for lowercase, and Atari's ATASCII), you will need to write a special program to be able to translate to standard ASCII if you are using a modem or some other peripheral which uses ASCII. See your computer's manual, the *Atari BASIC Reference Manual* for example, for information on your computer's internal character code.

CHR$

This is most useful in BASIC to let you use characters which cannot be represented within normal strings, will not show up on your screen, or cannot be typed from the keyboard. For example, if you have a printer attached to your computer, you could "send" CHR$(13) to it, and it would perform a carriage return. (The correct numbers which accomplish various things sometimes differ, though decimal 13 — an ASCII code standard — is nearly universally recognized as carriage return.) Or, you could send the combination CHR$(27)CHR$(8) and the printer would backspace.

Again, there is no real use for CHR$ within ML. If you want to specify a carriage return, just LDA #13. In ML, you are not limited to the character values which can appear on screen or within strings. Any value can be dealt with directly.

The following string manipulation instructions are found in Microsoft BASIC:

LEFT$

As usual in ML, *you* are in charge of manipulating data. Here's one way to extract a five-character-long "substring" from out of the left side of a string as in the BASIC statement: LEFT$ (X$,5)

```
2000 LDY #5
2002 LDX #0        (use X as the offset for buffer storage)
2004 LDA 1000,Y    (the location of X$)
2007 STA 4000,X    (the "buffer," or temporary storage area for
                    the substring)
2010 INX
2011 DEY
2012 BNE 2004
```

LEN

In some cases, you will already know the length of a string in ML. One of the ways to store and manipulate strings is to know beforehand the length and address of a string. Then you could use the subroutine given for LEFT$ above. More commonly, though, you will store your strings with delimiters (zeros, except in Atari) at the end of each string. To find out the length of a certain string:

```
2000 LDY #0
2002 LDA 1000,Y (the address of the string you are testing)
2003 BEQ 2009   (remember, if you LDA a zero, the zero flag is set.
                So you don't really need to use a CMP #0 here to
                test whether you've loaded the zero delimiter)
2005 INY
2006 BNE 2002   (we are not using a JMP here because we assume
                that all your strings are less than 256 characters
                long.)
2008 BRK        (if we still haven't found a zero after 256 INY's, we
                avoid an endless loop by just BRKing out of the
                subroutine)
2009 DEY        (the LENgth of the string is now in the Y register)
```

We had to DEY at the end because the final INY picked up the zero delimiter. So, the true count of the LENgth of the string is one less than Y shows, and we must DEY one time to make this adjustment.

MID$

To extract a substring which starts at the fourth character from within the string and is five characters long (as in MID$(X$,4,5)):

```
2000 LDY #5      (the size of the substring we're after)
2002 LDX #0      (X is the offset for storage of the substring)
2004 LDA 1003,Y (to start at the fourth character from within the
                X$ located at 1000, simply add three to that
                address. Instead of starting our LDA,Y at
                1000, skip to 1003. This is because the first
                character is not in position one. Rather, it is at
                the zeroth position, at 1000.)
2007 STA 4000,X (the temporary buffer to hold the substring)
2010 INX
2011 DEY
2012 BNE 2004
```

RIGHT$

This, too, is complicated because normally we do not know the LENgth of a given string. To find RIGHT$(X$,5) if X$ starts at 1000, we should find the LEN first and then move the substring to our holding zone (buffer) at 4000:

```
2000 LDY #0
2002 LDX #0
2004 LDA 1000,Y
2007 BEQ 2013   (the delimiting zero is found, so we know LEN)
2009 INY
2010 JMP 2004
2013 TYA        (put LEN into A to subtract substring size from it)
2014 SEC        (always set carry before subtraction)
2015 SBC #5     (subtract the size of the substring you want to
                 extract)
2017 TAY        (put the offset back into Y, now adjusted to point to
                 five characters from the end of X$)
2018 LDA 1000,Y
2021 BEQ 2030   (we found the delimiter, so end)
2023 STA 4000,X
2026 INX
2027 DEY
2028 BNE 2018
2030 RTS
```

The above does not apply to Atari since it cannot use zero as a delimiter.

SPC

This formatting instruction is similar to TAB. The difference is that SPC(10) moves you ten spaces to the right from wherever the cursor is on screen at the time. TAB(10) moves ten spaces from the *left-hand side of the screen*. In other words, TAB always counts over from the first column on any line; SPC counts from the cursor's current position.

In ML, you would just add the amount you want to SPC over. If you were printing to the screen and wanted ten spaces between A and B so it looked like this (A   B), you could write:

```
2000 LDA #65 (A)
2002 STA 32768   (screen RAM address)
2005 LDA #66 (B)
2007 STA 32778   (you've added ten to the target address)
```

Alternatively, you could add ten to the Y offset:

```
2000 LDY #0
2002 LDA #65
2004 STA 32768,Y
2007 LDY #10     (add ten to Y)
```

```
2009 LDA #66
2011 STA 32768,Y
```

If you are printing out many columns of numbers and need a subroutine to correctly space your printout, you might want to use a subroutine which will add ten to the Y offset each time you call the subroutine:

```
5000 TYA
5001 CLC
5002 ADC #10
5004 TAY
5005 RTS
```

This subroutine directly adds ten to the Y register whenever you JSR 5000. To really do this job, however, you should use a two-byte register to keep track of the cursor.

TAB

Quite similar to SPC, except that you don't add the offset from the cursor position (whatever location you most recently printed). Rather, TAB(X) moves ten over from the left side of the screen, or, if you are using a printer, from the left margin on the piece of paper. There is no particular reason to use TAB in ML. You have much more direct control in ML over where characters are printed out.

Appendix A

ADC	Add Memory To Accumulator With Carry		
Status Flags	N Z C I D V • • • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size In Bytes**
Immediate	ADC #Arg	69	2
Zero Page	ADC Arg	65	2
Zero Page, X	ADC Arg, X	75	2
Absolute	ADC Arg	6D	3
Absolute, X	ADC Arg, X	7D	3
Absolute, Y	ADC Arg, Y	79	3
(Indirect, X)	ADC (Arg, X)	61	2
(Indirect), Y	ADC (Arg), Y	71	2

AND	"AND" Memory With Accumulator		
Status Flags	N Z C I D V • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size In Bytes**
Immediate	AND # Arg	29	2
Zero Page	AND Arg	25	2
Zero Page, X	AND Arg,X	35	2
Absolute	AND Arg	2D	3
Absolute, X	AND Arg, X	3D	3
Absolute, Y	AND Arg, Y	39	3
(Indirect, X)	AND (Arg, X)	21	2
(Indirect), Y	AND (Arg),Y	31	2

ASL	Shift Left One Bit		
Status Flags	N Z C I D V • • •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Accumulator	ASL A	0A	1
Zero Page	ASL Arg	06	2
Zero Page, X	ASL Arg, X	16	2
Absolute	ASL Arg	0E	3
Absolute, X	ASL Arg, X	1E	3

BCC	Branch On Carry Clear		
Status Flags	N Z C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Relative	BCC Arg	90	2

BCS	Branch On Carry Set		
Status Flags	N Z C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Relative	BCS Arg	B0	2

BEQ	Branch On Zero		
Status Flags	N Z C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Relative	BEQ Arg	F0	2

BIT	Test Bits In Memory Against Accumulator		
Status Flags	N Z C I D V • • • (N, Z, V)		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Zero Page	BIT Arg	24	2
Absolute	BIT Arg	2C	3

BMI	Branch On Minus		
Status Flags	N Z C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Relative	BMI Arg	30	2

BNE	Branch On Anything But Zero		
Status Flags	N Z C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Relative	BNE Arg	D0	2

BPL	Branch On Plus		
Status Flags	N Z C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Relative	BPL Arg	10	2

BRK	Break		
Status Flags	N Z C I D V • (I)		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	BRK	00	1

BVC	Branch On Overflow Clear		
Status Flags	N Z C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Relative	BVC Arg	50	2

BVS	Branch On Overflow Set		
Status Flags	N Z C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Relative	BVS Arg	70	2

CLC Clear Carry Flag

Status Flags	N	Z	C	I	D	V
			•			

Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	CLC	18	1

CLD Clear Decimal Mode

Status Flags	N	Z	C	I	D	V
					•	

Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	CLD	D8	1

CLI Clear Interrupt Disable Bit

Status Flags	N	Z	C	I	D	V
				•		

Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	CLI	58	1

CLV Clear Overflow Flag

Status Flags	N	Z	C	I	D	V
						•

Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	CLV	B8	1

CMP	Compare Memory And Accumulator		
Status Flags	N • Z • C • I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Immediate	CMP # Arg	C9	2
Zero Page	CMP Arg	C5	2
Zero Page, X	CMP Arg, X	D5	2
Absolute	CMP Arg	CD	3
Absolute, X	CMP Arg, X	DD	3
Absolute, Y	CMP Arg, Y	D9	3
(Indirect, X)	CMP (Arg, X)	C1	2
(Indirect), Y	CMP (Arg), Y	D1	2

CPX	Compare Memory Against X Register		
Status Flags	N • Z • C • I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Immediate	CPX # Arg	E0	2
Zero Page	CPX Arg	E4	2
Absolute	CPX Arg	EC	3

CPY	Compare Memory Against Y Register		
Status Flags	N • Z • C • I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Immediate	CPY # Arg	C0	2
Zero Page	CPY Arg	C4	2
Absolute	CPY Arg	CC	3

DEC	Decrement Memory By One		
Status Flags	N Z C I D V • •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Zero Page Zero Page, X Absolute Absolute, X	DEC Arg DEC Arg, X DEC Arg DEC Arg, X	C6 D6 CE DE	2 2 3 3

DEX	Decrement X Register By One		
Status Flags	N Z C I D V • •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	DEX	CA	1

DEY	Decrement Y Register By One		
Status Flags	N Z C I D V • •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	DEY	88	1

EOR	Exclusive—Or Memory With Accumulator		
Status Flags	N Z C I D V • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size In Bytes**
Immediate	EOR # Arg	49	2
Zero Page	EOR Arg	45	2
Zero Page, X	EOR Arg, X	55	2
Absolute	EOR Arg	4D	3
Absolute, X	EOR Arg, X	5D	3
Absolute, Y	EOR Arg, Y	59	3
(Indirect, X)	EOR (Arg, X)	41	2
(Indirect), Y	EOR (Arg), Y	51	2

INC	Increment Memory By One		
Status Flags	N Z C I D V • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size In Bytes**
Zero Page	INC Arg	E6	2
Zero Page, X	INC Arg, X	F6	2
Absolute	INC Arg	EE	3
Absolute, X	INC Arg, X	FE	3

INX	Increment X Register By One		
Status Flags	N Z C I D V • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size In Bytes**
Implied	INX	E8	1

INY	Increment Y Register By One		
Status Flags	N Z C I D V • •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	INY	C8	1

JMP	Jump		
Status Flags	N Z C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Absolute	JMP Arg	4C	3
Indirect	JMP (Arg)	6C	3

JSR	Jump To New Location, But Save Return Address		
Status Flags	N Z C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Absolute	JSR Arg	20	3

LDA	Load Accumulator With Memory		
Status Flags	N Z C I D V • •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Immediate	LDA # Arg	A9	2
Zero Page	LDA Arg	A5	2
Zero Page, X	LDA Arg, X	B5	2
Absolute	LDA Arg	AD	3
Absolute, X	LDA Arg, X	BD	3
Absolute, Y	LDA Arg, Y	B9	3
(Indirect, X)	LDA (Arg, X)	A1	2
(Indirect), Y	LDA (Arg), Y	B1	2

LDX	Load X Register		
Status Flags	N Z C I D V • •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Immediate	LDX # Arg	A2	2
Zero Page	LDX Arg	A6	2
Zero Page, Y	LDX Arg, Y	B6	2
Absolute	LDX Arg	AE	3
Absolute, Y	LDX Arg, Y	BE	3

LDY	Load Y Register		
Status Flags	N Z C I D V • •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Immediate	LDY # Arg	A0	2
Zero Page	LDY Arg	A4	2
Zero Page, X	LDY Arg, X	B4	2
Absolute	LDY Arg	AC	3
Absolute, X	LDY Arg, X	BC	3

LSR	Shift Right One Bit In Either Memory Or Accumulator		
Status Flags	N Z C I D V • • •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Accumulator	LSR A	4A	1
Zero Page	LSR Arg	46	2
Zero Page, X	LSR Arg, X	56	2
Absolute	LSR Arg	4E	3
Absolute, X	LSR Arg, X	5E	3

NOP	No Operation		
Status Flags	N Z C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	NOP	EA	1

ORA	OR Memory With Accumulator		
Status Flags	N Z C I D V ● ●		
Addressing Mode	**Mnemonics**	**Opcode**	**Size In Bytes**
Immediate	ORA # Arg	09	2
Zero Page	ORA Arg	05	2
Zero Page, X	ORA Arg, X	15	2
Absolute	ORA Arg	0D	3
Absolute, X	ORA Arg, X	1D	3
Absolute, Y	ORA Arg, Y	19	3
(Indirect, X)	ORA (Arg, X)	01	2
(Indirect), Y	ORA (Arg), Y	11	2

PHA	Push Accumulator Onto The Stack		
Status Flags	N Z C I D V		
Addressing Mode	**Mnemonics**	**Opcode**	**Size In Bytes**
Implied	PHA	48	1

PHP	Push Processor Status Onto The Stack		
Status Flags	N Z C I D V		
Addressing Mode	**Mnemonics**	**Opcode**	**Size In Bytes**
Implied	PHP	08	1

PLA	Pull Accumulator From The Stack		
Status Flags	N • Z • C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	PLA	68	1

PLP	Pull Processor Status From The Stack		
Status Flags	N Z C I D V From Stack		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	PLP	28	1

ROL	Rotate One Bit Left In Memory Or The Accumulator		
Status Flags	N • Z • C • I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Accumulator	ROL A	2A	1
Zero Page	ROL Arg	26	2
Zero Page, X	ROL Arg, X	36	2
Absolute	ROL Arg	2E	3
Absolute, X	ROL Arg, X	3E	3

ROR

ROR	Rotate One Bit Right In Memory Or The Accumulator		
Status Flags	N Z C I D V • • •		
Addressing Mode	**Mnemonics**	**Opcode**	**Size In Bytes**
Accumulator	ROR A	6A	1
Zero Page	ROR Arg	66	2
Zero Page, X	ROR Arg, X	76	2
Absolute	ROR Arg	6E	3
Absolute, X	ROR Arg, X	7E	3

RTI	Return From Interrupt		
Status Flags	N Z C I D V From Stack		
Addressing Mode	**Mnemonics**	**Opcode**	**Size In Bytes**
Implied	RTI	40	1

RTS	Return From Subroutine		
Status Flags	N Z C I D V		
Addressing Mode	**Mnemonics**	**Opcode**	**Size In Bytes**
Implied	RTS	60	1

SBC	Subtract Memory From Accumulator, With Borrow		
Status Flags	N Z C I D V • • • •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Immediate	SBC # Arg	E9	2
Zero Page	SBC Arg	E5	2
Zero Page, X	SBC Arg, X	F5	2
Absolute	SBC Arg	ED	3
Absolute, X	SBC Arg, X	FD	3
Absolute, Y	SBC Arg, Y	F9	3
(Indirect, X)	SBC (Arg, X)	E1	2
(Indirect), Y	SBC (Arg), Y	F1	2

SEC	Set Carry Flag		
Status Flags	N Z C I D V •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	SEC	38	1

SED	Set Decimal Mode		
Status Flags	N Z C I D V •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	SED	F8	1

SEI

SEI	Set Interrupt Disable Status		
Status Flags	N Z C I D V • (under I)		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	SEI	78	1

STA	Store Accumulator In Memory		
Status Flags	N Z C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Zero Page	STA Arg	85	2
Zero Page, X	STA Arg, X	95	2
Absolute	STA Arg	8D	3
Absolute, X	STA Arg, X	9D	3
Absolute, Y	STA Arg, Y	99	3
(Indirect, X)	STA (Arg, X)	81	2
(Indirect), Y	STA (Arg), Y	91	2

STX	Store X Register In Memory		
Status Flags	N Z C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Zero Page	STX Arg	86	2
Zero Page, Y	STX Arg, Y	96	2
Absolute	STX Arg	8E	3

STY	Store Y Register In Memory		
Status Flags	N Z C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Zero Page	STY Arg	84	2
Zero Page, X	STY Arg, X	94	2
Absolute	STY Arg	8C	3

TAX	Transfer Accumulator To X Register		
Status Flags	N Z C I D V • •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	TAX	AA	1

TAY	Transfer Accumulator To Y Register		
Status Flags	N Z C I D V • •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	TAY	A8	1

TSX	Transfer Stack Pointer To X Register		
Status Flags	N Z C I D V • •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	TSX	BA	1

TXA	Transfer X Register To Accumulator		
Status Flags	N Z C I D V • •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	TXA	8A	1

TXS	Transfer X Register To Stack Pointer		
Status Flags	N Z C I D V		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	TXS	9A	1

TYA	Transfer Y Register To Accumulator		
Status Flags	N Z C I D V • •		
Addressing Mode	Mnemonics	Opcode	Size In Bytes
Implied	TYA	98	1

Appendix B

These maps, primarily the work of Jim Butterfield, all originally appeared in COMPUTE! Magazine. (See the copyright page for references.)

Map 1. PET Original And Upgrade BASIC.

ORIG	UPGR	DESCRIPTION
C357	C355	?OUT OF MEMORY
C359	C357	Send BASIC error message
C38B	C389	Warm start, BASIC
C3AC	C3AB	Crunch & insert line
C430	C439	Fix chaining & READY
C433	C442	Fix chaining
C48D	C495	Crunch tokens
C522	C52C	Find line in BASIC
C553	C55D	Do NEW
C56A	C572	Do CLR
C59A	C5A7	Reset BASIC to start
C6B5	C6C4	Continue BASIC execution
C863	C873	Get fixed-point number from BASIC
C9CE	C9DE	Send Return, LF if in screen mode
C9D2	C9E2	Send Return, Linefeed
CA27	CA1C	Print string
CA2D	CA22	Print precomputed string
CA49	CA45	Print character
CE11	CDF8	Check for comma
CE13	CDFA	Check for specific character
CE1C	CE03	'SYNTAX ERROR'
D079	D069	Bump Variable Address by 2
D0A7	D09A	Float to Fixed conversion
D278	D26D	Fixed to Float conversion
D679	D67B	Get byte to X reg
D68D	D68F	Evaluate String
D6C4	D6C6	Get two parameters
D73C	D773	Add (from memory)
D8FD	D934	Multiply by memory location
D9B4	D9EE	Multiply by ten
DA74	DAAE	Unpack memory variable to Accum #1
DB1B	DB55	Completion of Fixed to Float conversion
DC9F	DCD9	Print fixed-point value
DCA9	DCE3	Print floating-point value

DCAF	DCE9	Convert number to ASCII string
E3EA	E3D8	Print a character
na	E775	Output byte as 2 hex digits
na	E7A7	Input 2 hex digits to A
na	E7B6	Input 1 hex digit to A
F0B6	F0B6	Send 'talk' to IEEE
F0BA	F0BA	Send 'listen' to IEEE
F12C	F128	Send Secondary Address
E7DE	F156	Send canned message
F167	F16F	Send character to IEEE
F17A	F17F	Send 'untalk'
F17E	F183	Send 'unlisten'
F187	F18C	Input from IEEE
F2C8	F2A9	Close logical file
F2CD	F2AE	Close logical file in A
F32A	F301	Check for Stop key
F33F	F315	Send message if Direct mode
na	F322	LOAD subroutine
F3DB	F3E6	?LOAD ERROR
F3E5	F3EF	Print READY & reset BASIC to start
F3FF	F40A	Print SEARCHING . . .
F411	F41D	Print file name
F43F	F447	Get LOAD/SAVE type parameters
F462	F466	Open IEEE channel for output
F495	F494	Find specific tape header block
F504	F4FD	Get string
F52A	F521	Open logical file from input parameters
F52D	F524	Open logical file
F579	F56E	?FILE NOT FOUND, clear I/O
F57B	F570	Send error message
F5AE	F5A6	Find any tape header block
F64D	F63C	Get pointers for tape LOAD
F667	F656	Set tape buffer start address
F67D	F66C	Set cassette buffer pointers
F6E6	F6F0	Close IEEE channel
F78B	F770	Set input device from logical file number
F7DC	F7BC	Set output device from LFN
F83B	F812	PRESS PLAY. .; wait
F87F	F855	Read tape to buffer
F88A	F85E	Read tape
F8B9	F886	Write tape from buffer
F8C1	F883	Write tape, leader length in A
F913	F8E6	Wait for I/O complete or Stop key
FBDC	FB76	Reset tape I/O pointer
FD1B	FC9B	Set interrupt vector

FFC6	FFC6	Set input device
FFC9	FFC9	Set output device
FFCC	FFCC	Restore default I/O devices
FFCF	FFCF	Input character
FFD2	FFD2	Output character
FFE4	FFE4	Get character

Map 2. Upgrade PET/CBM Map.

0000-0002	0-2	USR Jump instruction
0003	3	Search character
0004	4	Scan-between-quotes flag
0005	5	BASIC input buffer pointer;#subscripts
0006	6	Default DIM flag
0007	7	Type: FF=string, 00=numeric
0008	8	Type: 80=integer, 00=floating point
0009	9	DATA scan flag; LIST quote flag; memory flag
000A	10	Subscript flag; FNx flag
000B	11	0=input; 64=get; 152=read
000C	12	ATN sign flag; comparison evaluation flag
000D	13	input flag; suppress output if negative
000E	14	current I/O device for prompt-suppress
0011-0012	17-18	BASIC integer address (for SYS, GOTO, etc.)
0013	19	Temporary string descriptor stack pointer
0014-0015	20-21	Last temporary string vector
0016-001E	22-30	Stack of descriptors for temporary strings
001F-0020	31-32	Pointer for number transfer
0021-0022	33-34	Misc. number pointer
0023-0027	35-39	Product staging area for multiplication
0028-0029	40-41	Pointer: Start-of-BASIC memory
002A-002B	42-43	Pointer: End-of-BASIC, Start-of-Variables
002C-002D	44-45	Pointer: End-of-Variables, Start-of-Arrays
002E-002F	46-47	Pointer: End-of-Arrays
0030-0031	48-49	Pointer: Bottom-of-strings (moving down)
0032-0033	50-51	Utility string pointer
0034-0035	52-53	Pointer: Limit of BASIC Memory
0036-0037	54-55	Current BASIC line number
0038-0039	56-57	Previous BASIC line number
003A-003B	58-59	Pointer to BASIC statement (for CONT)
003C-003D	60-61	Line number, current DATA line
003E-003F	62-63	Pointer to current DATA item
0040-0041	64-65	Input vector

0042-0043	66-67	Current variable name
0044-0045	68-69	Current variable address
0046-0047	70-71	Variable pointer for FOR/NEXT
0048	72	Y save register; new-operator save
004A	74	Comparison symbol accumulator
004B-004C	75-76	Misc. numeric work area
004D-0050	77-80	Work area; garbage yardstick
0051-0053	81-83	Jump vector for functions
0054-0058	84-88	Misc. numeric storage area
0059-005D	89-93	Misc. numeric storage area
005E-0063	94-99	Accumulator#1:E,M,M,M,M,S
0064	100	Series evaluation constant pointer
0065	101	Accumulator hi-order propagation word
0066-006B	102-107	Accumulator #2
006C	108	Sign comparison, primary vs. secondary
006D	109	low-order rounding byte for Acc #1
006E-006F	110-111	Cassette buffer length/Series pointer
0070-0087	112-135	Subrtn: Get BASIC Char; 77,78 = pointer
0088-008C	136-140	RND storage and work area
008D-008F	141-143	Jiffy clock for TI and TI$
0090-0091	144-145	Hardware interrupt vector
0092-0093	146-147	Break interrupt vector
0094-0095	148-149	NMI interrupt vector
0096	150	Status word ST
0097	151	Which key depressed: 255 = no key
0098	152	Shift key: 1 if depressed
0099-009A	153-154	Correction clock
009B	155	Keyswitch PIA: STOP and RVS flags
009C	156	Timing constant buffer
009D	157	Load = 0, Verify = 1
009E	158	#characters in keyboard buffer
009F	159	Screen reverse flag
00A0	160	IEEE-488 mode
00A1	161	End-of-line-for-input pointer
00A3-00A4	163-164	Cursor log (row, column)
00A5	165	PBD image for tape I/O
00A6	166	Key image
00A7	167	0 = flashing cursor, else no cursor
00A8	168	Countdown for cursor timing
00A9	169	Character under cursor
00AA	170	Cursor blink flag
00AB	171	EOT bit received
00AC	172	Input from screen/input from keyboard
00AD	173	X save flag
00AE	174	How many open files

00AF	175	Input device, normally 0
00B0	176	Output CMD device, normally 3
00B1	177	Tape character parity
00B2	178	Byte received flag
00B4	180	Tape buffer character
00B5	181	Pointer in file name transfer
00B7	183	Serial bit count
00B9	185	Cycle counter
00BA	186	Countdown for tape write
00BB	187	Tape buffer #1 count
00BC	188	Tape buffer #2 count
00BD	189	Write leader count; Read pass 1/pass 2
00BE	190	Write new byte; Read error flag
00BF	191	Write start bit; Read bit seq error
00C0	192	Pass 1 error log pointer
00C1	193	Pass 2 error correction pointer
00C2	194	0=Scan; 1-15=Count; $40=Load; $80=End
00C3	195	Checksum
00C4-00C5	196-197	Pointer to screen line
00C6	198	Position of cursor on above line
00C7-00C8	199-200	Utility pointer: tape buffer, scrolling
00C9-00CA	201-202	Tape end address/end of current program
00CB-00CC	203-204	Tape timing constants
00CD	205	00=direct cursor, else programmed cursor
00CE	206	Timer 1 enabled for tape read; 00=disabled
00CF	207	EOT signal received from tape
00D0	208	Read character error
00D1	209	# characters in file name
00D2	210	Current logical file number
00D3	211	Current secondary addrs, or R/W command
00D4	212	Current device number
00D5	213	Line length (40 or 80) for screen
00D6-00D7	214-215	Start of tape buffer, address
00D8	216	Line where cursor lives
00D9	217	Last key input; buffer checksum; bit buffer
00DA-00DB	218-219	File name pointer
00DC	220	Number of keyboard INSERTs outstanding
00DD	221	Write shift word/Receive input character
00DE	222	# blocks remaining to write/read
00DF	223	Serial word buffer
00E0-00F8	224-248	Screen line table: hi order address & line wrap
00F9	249	Cassette #1 status switch
00FA	250	Cassette #2 status switch
00FB-00FC	251-252	Tape start address
0100-010A	256-266	Binary to ASCII conversion area

0100-013E	256-318	Tape read error log for correction
0100-01FF	256-511	Processor stack area
0200-0250	512-592	BASIC input buffer
0251-025A	593-602	Logical file number table
025B-0264	603-612	Device number table
0265-026E	613-622	Secondary address, or R/W cmd, table
026F-0278	623-632	Keyboard input buffer
027A-0339	634-825	Tape #1 buffer
033A-03F9	826-1017	Tape #2 buffer
03FA-03FB	1018-1019	Vector for Machine Language Monitor
0400-7FFF	1024-32767	Available RAM including expansion
8000-8FFF	32768-36863	Video RAM
9000-BFFF	36864-49151	Available ROM expansion area
C000-E0F8	49152-57592	Microsoft BASIC interpreter
E0F9-E7FF	57593-59391	Keyboard, Screen, Interrupt programs
E810-E813	59408-59411	PIA1 - Keyboard I/O
E820-E823	59424-59427	PIA2 - IEEE-488 I/O
E840-E84F	59456-59471	VIA - I/O and Timers
F000-FFFF	61440-65535	Reset, tape, diagnostic monitor

Map 3. PET/CBM 4.0 BASIC. Zero Page.

Hex	Decimal	Description
0000-0002	0-2	USR jump
0003	3	Search character
0004	4	Scan-between-quotes flag
0005	5	Input buffer pointer; # of subscripts
0006	6	Default DIM flag
0007	7	Type: FF=string, 00=numeric
0008	8	Type: 80=integer, 00=floating point
0009	9	Flag: DATA scan; LIST quote; memory
000A	10	Subscript flag; FNX flag
000B	11	0=INPUT; $40=GET; $98=READ
000C	12	ATN sign/Comparison Evaluation flag
000D-000F	13-15	Disk status DS$ descriptor
0010	16	Current I/O device for prompt-suppress
0011-0012	17-18	Integer value (for SYS, GOTO etc)
0013-0015	19-21	Pointers for descriptor stack
0016-001E	22-30	Descriptor stack(temp strings)
001F-0022	31-34	Utility pointer area
0023-0027	35-39	Product area for multiplication
0028-0029	40-41	Pointer: Start-of-Basic
002A-002B	42-43	Pointer: Start-of-Variables
002C-002D	44-45	Pointer: Start-of-Arrays
002E-002F	46-47	Pointer: End-of-Arrays
0030-0031	48-49	Pointer: String-storage(moving down)
0032-0033	50-51	Utility string pointer
0034-0035	52-53	Pointer: Limit-of-memory
0036-0037	54-55	Current Basic line number
0038-0039	56-57	Previous Basic line number
003A-003B	58-59	Pointer: Basic statement for CONT

003C-003D	60-61	Current DATA line number
003E-003F	62-63	Current DATA address
0040-0041	64-65	Input vector
0042-0043	66-67	Current variable name
0044-0045	68-69	Current variable address
0046-0047	70-71	Variable pointer for FOR/NEXT
0048-0049	72-73	Y-save; op-save; Basic pointer save
004A	74	Comparison symbol accumulator
004B-0050	75-80	Misc work area, pointers, etc
0051-0053	81-83	Jump vector for functions
0054-005D	84-93	Misc numeric work area
005E	94	Accum#1: Exponent
005F-0062	95-98	Accum#1: Mantissa
0063	99	Accum#1: Sign
0064	100	Series evaluation constant pointer
0065	101	Accum#1 hi-order (overflow)
0066-006B	102-107	Accum#2: Exponent, etc.
006C	108	Sign comparison, Acc#1 vs #2
006D	106	Accum#1 lo-order (rounding)
006E-006F	110-111	Cassette buff len/Series pointer
0070-0087	112-135	CHRGET subroutine; get Basic char
0077-0078	119-120	Basic pointer (within subrtn)
0088-008C	136-140	Random number seed.
008D-008F	141-143	Jiffy clock for TI and TI$
0090-0091	144-145	Hardware interrupt vector
0092-0093	146-147	BRK interrupt vector
0094-0095	148-149	NMI interrupt vector
0096	150	Status word ST
0097	151	Which key down; 255=no key
0098	152	Shift key: 1 if depressed
0099-009A	153-154	Correction clock
009B	155	Keyswitch PIA: STOP and RVS flags
009C	156	Timing constant for tape
009D	157	Load=0, Verify=1
009E	158	Number of characters in keybd buffer
009F	159	Screen reverse flag
00A0	160	IEEE output; 255=character pending
00A1	161	End-of-line-for-input pointer
00A3-00A4	163-164	Cursor log (row, column)
00A5	165	IEEE output buffer
00A6	166	Key image
00A7	167	0=flash cursor
00A8	168	Cursor timing countdown
00A9	169	Character under cursor
00AA	170	Cursor in blink phase
00AB	171	EOT received from tape
00AC	172	Input from screen/from keyboard
00AD	173	X save
00AE	174	How many open files
00AF	175	Input device, normally 0
00B0	176	Output CMD device, normally 3
00B1	177	Tape character parity
00B2	178	Byte received flag
00B3	179	Logical Address temporary save
00B4	180	Tape buffer character; MLM command
00B5	181	File name pointer; MLM flag, counter
00B7	183	Serial bit count
00B9	185	Cycle counter

00BA	186	Tape writer countdown
00BB-00BC	187-188	Tape buffer pointers, #1 and #2
00BD	189	Write leader count; read pass1/2
00BE	190	Write new byte; read error flag
00BF	191	Write start bit; read bit seq error
00C0-00C1	192-193	Error log pointers, pass1/2
00C2	194	0=Scan/1-15=Count/$40=Load/$80=End
00C3	195	Write leader length; read checksum
00C4-00C5	196-197	Pointer to screen line
00C6	198	Position of cursor on above line
00C7-00C8	199-200	Utility pointer: tape, scroll
00C9-00CA	201-202	Tape end addrs/End of current program
00CB-00CC	203-204	Tape timing constants
00CD	205	0=direct cursor, else programmed
00CE	206	Tape read timer 1 enabled
00CF	207	EOT received from tape
00D0	208	Read character error
00D1	209	# characters in file name
00D2	210	Current file logical address
00D3	211	Current file secondary addrs
00D4	212	Current file device number
00D5	213	Right-hand window or line margin
00D6-00D7	214-215	Pointer: Start of tape buffer
00D8	216	Line where cursor lives
00D9	217	Last key/checksum/misc.
00DA-00DB	218-219	File name pointer
00DC	220	Number of INSERTs outstanding
00DD	221	Write shift word/read character in
00DE	222	Tape blocks remaining to write/read
00DF	223	Serial word buffer
00E0-00F8	224-248	(40-column) Screen line wrap table
00E0-00E1	224-225	(80-column) Top, bottom of window
00E2	226	(80-column) Left window margin
00E3	227	(80-column) Limit of keybd buffer
00E4	228	(80-column) Key repeat flag
00E5	229	(80-column) Repeat countdown
00E6	230	(80-column) New key marker
00E7	231	(80-column) Chime time
00E8	232	(80-column) HOME count
00E9-00EA	233-234	(80-column) Input vector
00EB-00EC	235-236	(80-column) Output vector
00F9-00FA	249-250	Cassette status, #1 and #2
00FB-00FC	251-252	MLM pointer/Tape start address
00FD-00FE	253-254	MLM, DOS pointer, misc.
0100-010A	256-266	STR$ work area, MLM work
0100-013E	256-318	Tape read error log
0100-01FF	256-511	Processor stack
0200-0250	512-592	MLM work area; Input buffer
0251-025A	593-602	File logical address table
025B-0264	603-612	File device number table
0265-026E	613-622	File secondary adds table
026F-0278	623-632	Keyboard input buffer
027A-0339	634-825	Tape#1 input buffer
033A-03F9	826-1017	Tape#2 input buffer
033A	826	DOS character pointer
033B	827	DOS drive 1 flag
033C	828	DOS drive 2 flag
033D	829	DOS length/write flag

033E	830	DOS syntax flags
033F-0340	831-832	DOS disk ID
0341	833	DOS command string count
0342-0352	834-850	DOS file name buffer
0353-0380	851-896	DOS command string buffer
03EE-03F7	1006-1015	(80-column) Tab stop table
03FA-03FB	1018-1019	Monitor extension vector
03FC	1020	IEEE timeout defeat
0400-7FFF	1024-32767	Available RAM including expansion
8000-83FF	32768-33791	(40-column) Video RAM
8000-87FF	32768-34815	(80-column) Video RAM
9000-AFFF	36864-45055	Available ROM expansion area
B000-DFFF	45056-57343	Basic, DOS, Machine Lang Monitor
E000-E7FF	57344-59391	Screen, Keyboard, Interrupt programs
E810-E813	59408-59411	PIA 1 - Keyboard I/O
E820-E823	59424-59427	PIA 2 - IEEE-488 I/O
E840-E84F	59456-59471	VIA - I/O and timers
E880-E881	59520-59521	(80-column) CRT Controller
F000-FFFF	61440-65535	Reset, I/O handlers, Tape routines

Map 4. PET/CBM 4.0 BASIC ROM Routines.

	Description
B000-B065	Action addresses for primary keywords
B066-B093	Action addresses for functions
B094-B0B1	Hierarchy and action addresses for operators
B0B2-B20C	Table of Basic keywords
B20D-B321	Basic messages, mostly error messages
B322-B34F	Search the stack for FOR or GOSUB activity
B350-B392	Open up space in memory
B393-B39F	Test: stack too deep?
B3A0-B3CC	Check available memory
B3CD	Send canned error message, then:
B3FF-B41E	Warm start; wait for Basic command
B41F-B4B5	Handle new Basic line input
B4B6-B4E1	Rebuild chaining of Basic lines
B4E2-B4FA	Receive line from keyboard
B4FB-B5A2	Crunch keywords into Basic tokens
B5A3-B5D1	Search Basic for given line number
B5D2	Perform NEW, and;
B5EC-B621	Perform CLR
B622-B62F	Reset Basic execution to start
B630-B6DD	Perform LIST
B6DE-B784	Perform FOR
B785-B7B6	Execute Basic statement
B7B7-B7C5	Perform RESTORE
B7C6-B7ED	Perform STOP or END
B7EE-B807	Perform CONT
B808-B812	Perform RUN
B813-B82F	Perform GOSUB
B830-B85C	Perform GOTO
B85D	Perform RETURN, then:

B883-B890	Perform DATA: skip statement
B891	Scan for next Basic statement
B894-B8B2	Scan for next Basic line
B8B3	Perform IF, and perhaps:
B8C6-B8D5	Perform REM: skip line
B8D6-B8F5	Perform ON
B8F6-B92F	Accept fixed-point number
B930-BA87	Perform LET
BA88-BA8D	Perform PRINT#
BA8E-BAA1	Perform CMD
BAA2-BB1C	Perform PRINT
BB1D-BB39	Print string from memory
BB3A-BB4B	Print single format character
BB4C-BB79	Handle bad input data
BB7A-BBA3	Perform GET
BBA4-BBBD	Perform INPUT#
BBBE-BBF4	Perform INPUT
BBF5-BC01	Prompt and receive input
BC02-BCF6	Perform READ
BCF7-BD18	Canned Input error messages
BD19-BD71	Perform NEXT
BD72-BD97	Check type mismatch
BD98	Evaluate expression
BEE9	Evaluate expression within parentheses
BEEF	Check parenthesis, comma
BF00-BF0B	Syntax error exit
BF8C-C046	Variable name setup
C047-C085	Set up function references
C086-C0B5	Perform OR, AND
C0B6-C11D	Perform comparisons
C11E-C12A	Perform DIM
C12B-C1BF	Search for variable
C1C0-C2C7	Create new variable
C2C8-C2D8	Setup array pointer
C2D9-C2DC	32768 in floating binary
C2DD-C2FB	Evaluate integer expression
C2FC-C4A7	Find or make array
C4A8	Perform FRE, and:
C4BC-C4C8	Convert fixed-to-floating
C4C9-C4CE	Perform POS
C4CF-C4DB	Check not Direct
C4DC-C509	Perform DEF
C50A-C51C	Check FNx syntax
C51D-C58D	Evaluate FNx
C58E-C59D	Perform STR$
C59E-C5AF	Do string vector
C5B0-C61C	Scan, set up string
C61D-C669	Allocate space for string
C66A-C74E	Garbage collection
C74F-C78B	Concatenate
C78C-C7B4	Store string
C7B5-C810	Discard unwanted string

C811-C821	Clean descriptor stack
C822-C835	Perform CHR$
C836-C861	Perform LEFT$
C862-C86C	Perform RIGHT$
C86D-C896	Perform MID$
C897-C8B1	Pull string data
C8B2-C8B7	Perform LEN
C8B8-C8C0	Switch string to numeric
C8C1-C8D0	Perform ASC
C8D1-C8E2	Get byte parameter
C8E3-C920	Perform VAL
C921-C92C	Get two parameters for POKE or WAIT
C92D-C942	Convert floating-to-fixed
C943-C959	Perform PEEK
C95A-C962	Perform POKE
C963-C97E	Perform WAIT
C97F-C985	Add 0.5
C986	Perform subtraction
C998-CA7C	Perform addition
CA7D-CAB3	Complement accum#1
CAB4-CAB8	Overflow exit
CAB9-CAF1	Multiply-a-byte
CAF2-CB1F	Constants
CB20	Perform LOG
CB5E-CBC1	Perform multiplication
CBC2-CBEC	Unpack memory into accum#2
CBED-CC09	Test & adjust accumulators
CC0A-CC17	Handle overflow and underflow
CC18-CC2E	Multiply by 10
CC2F-CC33	10 in floating binary
CC34	Divide by 10
CC3D	Perform divide-by
CC45-CCD7	Perform divide-into
CCD8-CCFC	Unpack memory into accum#1
CCFD-CD31	Pack accum#1 into memory
CD32-CD41	Move accum#2 to #1
CD42-CD50	Move accum#1 to #2
CD51-CD60	Round accum#1
CD61-CD6E	Get accum#1 sign
CD6F-CD8D	Perform SGN
CD8E-CD90	Perform ABS
CD91-CDD0	Compare accum#1 to memory
CDD1-CE01	Floating-to-fixed
CE02-CE28	Perform INT
CE29-CEB3	Convert string to floating-point
CEB4-CEE8	Get new ASCII digit
CEE9-CEF8	Constants
CF78	Print IN, then:
CF7F-CF92	Print Basic line #
CF93-D0C6	Convert floating-point to ASCII
D0C7-D107	Constants
D108	Perform SQR

```
D112      Perform power function
D14B-D155 Perform negation
D156-D183 Constants
D184-D1D6 Perform EXP
D1D7-D220 Series evaluation
D221-D228 RND constants
D229-D281 Perform RND
D282      Perform COS
D289-D2D1 Perform SIN
D2D2-D2FD Perform TAN
D2FE-D32B Constants
D32C-D35B Perform ATN
D35C-D398 Constants
D399-D3B5 CHRGET sub for zero page
D3B6-D471 Basic cold start
D472-D716 Machine Language Monitor
D717-D7AB MLM subroutines
D7AC-D802 Perform RECORD
D803-D837 Disk parameter checks
D838-D872 Dummy disk control messages
D873-D919 Perform CATALOG or DIRECTORY
D91A-D92E Output
D92F-D941 Find spare secondary address
D942-D976 Perform DOPEN
D977-D990 Perform APPEND
D991-D9D1 Get disk status
D9D2-DA06 Perform HEADER
DA07-DA30 Perform DCLOSE
DA31-DA64 Set up disk record
DA65-DA7D Perform COLLECT
DA7E-DAA6 Perform BACKUP
DAA7-DAC6 Perform COPY
DAC7-DAD3 Perform CONCAT
DAD4-DB0C Insert command string values
DB0D-DB39 Perform DSAVE
DB3A-DB65 Perform DLOAD
DB66-DB98 Perform SCRATCH
DB99-DB9D Check Direct command
DB9E-DBD6 Query ARE YOU SURE?
DBD7-DBE0 Print BAD DISK
DBE1-DBF9 Clear DS$ and ST
DBFA-DC67 Assemble disk command string
DC68-DE29 Parse Basic DOS command
DE2C-DE48 Get Device number
DE49-DE86 Get file name
DE87-DE9C Get small variable parameter
** Entry points only for E000-E7FF **
E000      Register/screen initialization
E0A7      Input from keyboard
E116      Input from screen
E202      Output character
E442      Main Interrupt entry
```

```
E455      Interrupt: clock, cursor, keyboard
E600      Exit from Interrupt
**                                   **
F000-F0D1 File messages
F0D2      Send 'Talk'
F0D5      Send 'Listen'
F0D7      Send IEEE command character
F109-F142 Send byte to IEEE
F143-F150 Send byte and clear ATN
F151-F16B Option: timeout or wait
F16C-F16F DEVICE NOT PRESENT
F170-F184 Timeout on read, clear control lines
F185-F192 Send canned file message
F193-F19D Send byte, clear control lines
F19E-F1AD Send normal (deferred) IEEE char
F1AE-F1BF Drop IEEE device
F1C0-F204 Input byte from IEEE
F205-F214 GET a byte
F215-F265 INPUT a byte
F266-F2A1 Output a byte
F2A2      Abort files
F2A6-F2C0 Restore default I/O devices
F2C1-F2DC Find/setup file data
F2DD-F334 Perform CLOSE
F335-F342 Test STOP key
F343-F348 Action STOP key
F349-F350 Send message if Direct mode
F351-F355 Test if Direct mode
F356-F400 Program load subroutine
F401-F448 Perform LOAD
F449-F46C Print SEARCHING
F46D-F47C Print LOADING or VERIFYING
F47D-F4A4 Get Load/Save parameters
F4A5-F4D2 Send name to IEEE
F4D3-F4F5 Find specific tape header
F4F6-F50C Perform VERIFY
F50D-F55F Get Open/Close parameters
F560-F5E4 Perform OPEN
F5E5-F618 Find any tape header
F619-F67A Write tape header
F67B-F694 Get start/end addrs from header
F695-F6AA Set buffer address
F6AB-F6C2 Set buffer start & end addrs
F6C3-F6CB Perform SYS
F6CC-F6DC Set tape write start & end
F6DD-F767 Perform SAVE
F768-F7AE Update clock
F7AF-F7FD Connect input device
F7FE-F84A Connect output device
F84B-F856 Bump tape buffer pointer
F857-F879 Wait for PLAY
```

```
F87A-F88B Test cassette switch
F88C-F899 Wait for RECORD
F89A      Initiate tape read
F8CB      Initiate tape write
F8E0-F92A Common tape I/O
F92B-F934 Test I/O complete
F935-F944 Test STOP key
F945-F975 Tape bit timing adjust
F976-FA9B Read tape bits
FA9C-FBBA Read tape characters
FBBB-FBC3 Reset tape read address
FBC4-FBC8 Flag error into ST
FBC9-FBD7 Reset counters for new byte
FBD8-FBF3 Write a bit to tape
FBF4-FC85 Tape write
FC86-FCBF Write tape leader
FCC0-FCDA Terminate tape; restore interrupt
FCDB-FCEA Set interrupt vector
FCEB-FCF8 Turn off tape motor
FCF9-FD0A Checksum calculation
FD0B-FD15 Advance load/save pointer
FD16-FD4B Power-on Reset
FD4C-FD5C Table of interrupt vectors
** Jump table:                   **
FF93-FF9E CONCAT,DOPEN,DCLOSE,RECORD
FF9F-FFAA HEADER,COLLECT,BACKUP,COPY
FFAB-FFB6 APPEND,DSAVE,DLOAD,CATALOG
FFB7-FFBC RENAME,SCRATCH
FFBD      Get disk status
FFC0      OPEN
FFC3      CLOSE
FFC6      Set input device
FFC9      Set output device
FFCC      Restore default I/O devices
FFCF      INPUT a byte
FFD2      Output a byte
FFD5      LOAD
FFD8      SAVE
FFDB      VERIFY
FFDE      SYS
FFE1      Test stop key
FFE4      GET byte
FFE7      Abort all files
FFEA      Update clock
FFFA-FFFF Hard vectors:  NMI, Reset, INT
```

Map 5. VIC Zero Page And BASIC ROMs.

Hex	Decimal	Description
0000-0002	0-2	USR jump
0003-0004	3-4	Float-Fixed vector
0005-0006	5-6	Fixed-Float vector
0007	7	Search character
0008	8	Scan-quotes flag
0009	9	TAB column save
000A	10	0=LOAD, 1=VERIFY
000B	11	Input buffer pointer/# subscrpt
000C	12	Default DIM flag
000D	13	Type: FF=string, 00=numeric
000E	14	Type: 80=integer, 00=floating point
000F	15	DATA scan/LIST quote/memry flag
0010	16	Subscript/FNx flag
0011	17	0=INPUT;$40=GET;$98=READ
0012	18	ATN sign/Comparison eval flag
0013	19	Current I/O prompt flag
0014-0015	20-21	Integer value
0016	22	Pointer: temporary strg stack
0017-0018	23-24	Last temp string vector
0019-0021	25-33	Stack for temporary strings
0022-0025	34-37	Utility pointer area
0026-002A	38-42	Product area for multiplication
002B-002C	43-44	Pointer: Start-of-Basic

002D-002E	45-46	Pointer: Start-of-Variables
002F-0030	47-48	Pointer: Start-of-Arrays
0031-0032	49-50	Pointer: End-of-Arrays
0033-0034	51-52	Pointer: String-storage(moving down)
0035-0036	53-54	Utility string pointer
0037-0038	55-56	Pointer: Limit-of-memory
0039-003A	57-58	Current Basic line number
003B-003C	59-60	Previous Basic line number
003D-003E	61-62	Pointer: Basic statement for CONT
003F-0040	63-64	Current DATA line number
0041-0042	65-66	Current DATA address
0043-0044	67-68	Input vector
0045-0046	69-70	Current variable name
0047-0048	71-72	Current variable address
0049-004A	73-74	Variable pointer for FOR/NEXT
004B-004C	75-76	Y-save; op-save; Basic pointer save
004D	77	Comparison symbol accumulator
004E-0053	78-83	Misc work area, pointers, etc
0054-0056	84-86	Jump vector for functions
0057-0060	87-96	Misc numeric work area
0061	97	Accum#1: Exponent
0062-0065	98-101	Accum#1: Mantissa
0066	102	Accum#1: Sign
0067	103	Series evaluation constant pointer
0068	104	Accum#1 hi-order (overflow)
0069-006E	105-110	Accum#2: Exponent, etc.

006F	111	Sign comparison, Acc#1 vs #2
0070	112	Accum#1 lo-order (rounding)
0071-0072	113-114	Cassette buff len/Series pointer
0073-008A	115-138	CHRGET subroutine; get Basic char
007A-007B	122-123	Basic pointer (within subrtn)
008B-008F	139-143	RND seed value
0090	144	Status word ST
0091	145	Keyswitch PIA: STOP and RVS flags
0092	146	Timing constant for tape
0093	147	Load=0, Verify=1
0094	148	Serial output: deferred char flag
0095	149	Serial deferred character
0096	150	Tape EOT received
0097	151	Register save
0098	152	How many open files
0099	153	Input device, normally 0
009A	154	Output CMD device, normally 3
009B	155	Tape character parity
009C	156	Byte-received flag
009D	157	Direct=$80/RUN=0 output control
009E	158	Tp Pass 1 error log/char buffer
009F	159	Tp Pass 2 err log corrected
00A0-00A2	160-162	Jiffy Clock HML
00A3	163	Serial bit count/EOI flag
00A4	164	Cycle count
00A5	165	Countdown,tape write/bit count

00A6	166	Tape buffer pointer
00A7	167	Tp Wrt ldr count/Rd pass/inbit
00A8	168	Tp Wrt new byte/Rd error/inbit cnt
00A9	169	Wrt start bit/Rd bit err/stbit
00AA	170	Tp Scan;Cnt;Ld;End/byte assy
00AB	171	Wr lead length/Rd checksum/parity
00AC-00AD	172-173	Pointer: tape bufr, scrolling
00AE-00AF	174-175	Tape end adds/End of program
00B0-00B1	176-177	Tape timing constants
00B2-00B3	178-179	Pntr: start of tape buffer
00B4	180	1=Tp timer enabled; bit cnt
00B5	181	Tp EOT/RS232 next bit to send
00B6	182	Read character error/outbyte buf
00B7	183	# characters in file name
00B8	184	Current logical file
00B9	185	Current secndy address
00BA	186	Current device
00BB-00BC	187-188	Pointer to file name
00BD	189	Wr shift word/Rd input char
00BE	190	# blocks remaining to Wr/Rd
00BF	191	Serial word buffer
00C0	192	Tape motor interlock
00C1-00C2	193-194	I/O start adds
00C3-00C4	195-196	Kernel setup pointer
00C5	197	Last key pressed
00C6	198	# chars in keybd buffer

00C7	199	Screen reverse flag
00C8	200	End-of-line for input pointer
00C9-00CA	201-202	Input cursor log (row, column)
00CB	203	Which key: 64 if no key
00CC	204	0=flash cursor
00CD	205	Cursor timing countdown
00CE	206	Character under cursor
00CF	207	Cursor in blink phase
00D0	208	Input from screen/from keyboard
00D1-00D2	209-210	Pointer to screen line
00D3	211	Position of cursor on above line
00D4	212	0=direct cursor, else programmed
00D5	213	Current screen line length
00D6	214	Row where curosr lives
00D7	215	Last inkey/checksum/buffer
00D8	216	# of INSERTs outstanding
00D9-00F0	217-240	Screen line link table
00F1	241	Dummy screen link
00F2	242	Screen row marker
00F3-00F4	243-244	Screen color pointer
00F5-00F6	245-246	Keyboard pointer
00F7-00F8	247-248	RS-232 Rcv pntr
00F9-00FA	249-250	RS-232 Tx pntr
00FF-010A	255-266	Floating to ASCII work area

```
FF8A-FFF5 65418-65525 Jump Table, Including:
  FFC6 - Set Input channel
  FFC9 - Set Output channel
  FFCC - Restore default I/O channels
  FFCF - INPUT
  FFD2 - PRINT
  FFE1 - Test Stop key
  FFE4 - GET

c000   ROM control vectors
c00c   Keyword action vectors
c052   Function vectors
c080   Operator vectors
c09e   Keywords
c19e   Error messages
c328   Error message vectors
c365   Miscellaneous messages
c38a   Scan stack for FOR/GOSUB
c3b8   Move memory
c3fb   Check stack depth
c408   Check memory space
c435   'OUT OF MEMORY'
c437   Error routine
c469   Break entry
c474   'READY.'
c480   Ready for Basic
c49c   Handle new line
c533   Re-chain lines
c560   Receive input line
c579   Crunch tokens
c613   Find Basic line
c642   Perform [NEW]
c65e   Perform [CLR]
c68e   Back up text pointer
c69c   Perform [LIST]
c742   Perform [FOR]
c7ed   Execute statement
c81d   Perform [RESTORE]
c82c   Break
c82f   Perform [STOP]
c831   Perform [END]
c857   Perform [CONT]
c871   Perform [RUN]
```

c883	Perform [GOSUB]
c8a0	Perform [GOTO]
c8d2	Perform [RETURN]
c8f8	Perform [DATA]
c906	Scan for next statement
c928	Perform [IF]
c93b	Perform [REM]
c94b	Perform [ON]
c96b	Get fixed point number
c9a5	Perform [LET]
ca80	Perform [PRINT#]
ca86	Perform [CMD]
caa0	Perform [PRINT]
cb1e	Print message from (y,a)
cb3b	Print format character
cb4d	Bad-input routines
cb7b	Perform [GET]
cba5	Perform [INPUT#]
cbbf	Perform [INPUT]
cbf9	Prompt & input
cc06	Perform [READ]
ccfc	Input error messages
cd1e	Perform [NEXT]
cd78	Type-match check
cd9e	Evaluate expression
cea8	Constant - PI
cef1	Evaluate within brackets
cef7	Check for ')'
ceff	Check for comma
cf08	Syntax error
cf14	Check range
cf28	Search for variable
cfa7	Set up FN reference
cfe6	Perform [OR]
cfe9	Perform [AND]
d016	Compare
d081	Perform [DIM]
d08b	Locate variable
d113	Check alphabetic
d11d	Create variable
d194	Array pointer subroutine
d1a5	Value 32768
d1b2	Float-fixed conversion
d1d1	Set up array
d245	'BAD SUBSCRIPT'
d248	'ILLEGAL QUANTITY'
d34c	Compute array size
d37d	Perform [FRE]
d391	Fixed-float conversion
d39e	Perform [POS]
d3a6	Check direct
d3b3	Perform [DEF]

d3e1	Check FN syntax	d824	Perform [POKE]
d3f4	Perform [FN]	d82d	Perform [WAIT]
d465	Perform [STR$]	d849	Add 0.5
d475	Calculate string vector	d850	Subtract-from
d487	Set up string	d853	Perform [SUBTRACT]
d4f4	Make room for string	d86a	Perform [ADD]
d526	Garbage collection	d947	Complement fac#1
d5bd	Check salvageability	d97e	'OVERFLOW'
d606	Collect string	d983	Multiply by zero byte
d63d	Concatenate	d9ea	Perform [LOG]
d67a	Build string to memory	da2b	Perform [MULTIPLY]
d6a3	Discard unwanted string	da59	Multiply-a-bit
d6db	Clean descriptor stack	da8c	Memory to FAC#2
d6ec	Perform [CHR$]	dab7	Adjust FAC#1/#2
d700	Perform [LEFT$]	dad4	Underflow/overflow
d72c	Perform [RIGHT$]	dae2	Multiply by 10
d737	Perform [MID$]	daf9	+10 in floating pt
d761	Pull string parameters	dafe	Divide by 10
d77c	Perform [LEN]	db12	Perform [DIVIDE]
d782	Exit string-mode	dba2	Memory to fac#1
d78b	Perform [ASC]	dbc7	FAC#1 to memory
d79b	Input byte parameter	dbfc	FAC#2 to fac#1
d7ad	Perform [VAL]	dc0c	FAC#1 to FAC#2
d7eb	Get params for poke/wait	dc1b	Round FAC#1
d7f7	Float-fixed	dc2b	Get sign
d80d	Perform [PEEK]	dc39	Perform [SGN]

dc58	Perform [ABS]
dc5b	Compare FAC#1 to mem
dc9b	Float-fixed
dccc	Perform [INT]
dcf3	String to fac
dd7e	Get ascii digit
dddd	Float to ascii
df16	Decimal constants
df3a	TI constants
df71	Perform [SQR]
df7b	Perform [POWER]
dfb4	Perform [NEGATIVE]
dfed	Perform [EXP]
e040	Series evaluate 1
e056	Series evaluate 2
e094	Perform [RND]
e0f6	?? Breakpoints ??
e127	Perform [SYS]
e153	Perform [SAVE]
e162	Perform [VERIFY]
e165	Perform [LOAD]
e1bb	Perform [OPEN]
e1c4	Perform [CLOSE]
e1d1	Parameters for load/save
e203	Check default parameters
e20b	Check for comma
e216	Parameters for open/close
e261	Perform [COS]
e268	Perform [SIN]
e2b1	Perform [TAN]
e30b	Perform [ATN]
e378	Initialize
e387	CHRGET for zero page
e3a4	Initialize Basic
e429	Power-up message
e44f	Vectors for $300
e45b	Initialize vectors
e467	Warm restart
e476	Program patch area
e4a0	Serial output '1'
e4a9	Serial output '0'
e4b2	Get serial input & clock
e4bc	Program patch area
e500	Set 6522 addrs
e505	Set screen limits
e50a	Track cursor location
e518	Initalize I/O
e54c	Normalize screen
e55f	Clear screen
e581	Home cursor
e587	Set screen pointers
e5bb	Set I/o defaults

e5c3	Set vic chip defaults
e5cf	Input from keyboard
e64f	Input from screen
e6b8	Quote mark test
e6c5	Set up screen print
e6ea	Advance cursor
e715	Retreat cursor
e72d	Back into previous line
e742	Output to screen
e8c3	Go to next line
e8d8	Do 'RETURN'
e8e8	Check line decrement
e8fa	Check line increment
e912	Set colour code
e921	Colour code table
e929	Code conversion
e975	Scroll screen
e9ee	Open space on screen
ea56	Move screen line
ea6e	Synch colour transfer
ea7e	Set start-of-line
ea8d	Clear screen line
eaa1	Print to screen
eaaa	Store on screen
eab2	Synch colour to char
eabf	Interrupt (IRQ)
eb1e	Check keyboard
ec00	Set text mode
ec46	Keyboard vectors
ec5e	Keyboard maps
ed21	Graphics/text control
ed30	Set graphics mode
ed5b	Wrap up screen line
ed6a	Shifted key matrix
eda3	Control key matrix
ede4	Vic chip defaults
edfd	Screen line adds low
ee14	Send 'talk'
ee17	Send 'listen'
ee1c	Send control char
ee49	Send to serial bus
eeb7	Timeout on serial
eec0	Send listen SA
eec5	Clear ATN
eece	Send talk SA
eee4	Send serial deferred
eef6	Send 'untalk'
ef04	Send 'unlisten'
ef19	Receive from serial bus
ef84	Clock line on
ef8d	Clock line off
ef96	Delay 1 ms

efa3	RS232 send (NMI)
efee	New RS232 byte send
f016	Error or quit
f027	Compute bit count
f036	RS232 receive (NMI)
f05b	Setup to receive
f09d	Receive parity error
f0a2	Receive overrun error
f0a5	Receive break error
f0a8	Receive frame error
f0b9	Bad device
f0bc	File to RS232
f0ed	Send to RS232 buffer
f116	Input from RS232 buffer
f14f	Get from RS232 buffer
f160	Check serial bus idle
f174	Messages
f1e2	Print if direct
f1f5	Get..
f205	..from RS232
f20e	Input
f250	Get.. tape/serial/RS232
f27a	Output..
f290	..to tape
f2c7	Set input device
f309	Set output device
f34a	Close
f3cf	Find file
f3df	Set file values
f3ef	Abort all files
f3f3	Restore default I/O
f40a	Do file opening
f495	Send SA
f4c7	Open RS232
f542	Load program
f647	'SEARCHING'
f659	Print file name
f66a	'LOADING/VERIFYING'
f675	Save program
f728	'SAVING'
f734	Bump clock
f760	Get time
f767	Set time
f770	Action stop key
f77e	File Error Messages
f7af	Find any tape header
f7e7	Write tape header
f84d	Get buffer address
f854	Set buffer start, end pointers
f867	Find specific header
f88a	Bump tape pointer

f894	'PRESS PLAY .. '
f8ab	Check cassette status
f8b7	'PRESS RECORD ..'
f8c0	Initiate tape read
f8e3	Initiate tape write
f8f4	Common tape read/write
f94b	Check tape stop
f95d	Set timing
f98e	Read bits (IRQ)
faad	Store characters
fbd2	Reset pointer
fbdb	New tape character setup
fbea	Toggle tape
fc06	Data write
fc0b	Tape write (IRQ)
fc95	Leader write (IRQ)
fccf	Restore vectors
fcf6	Set vector
fd08	Kill motor
fd11	Check read/write pointer
fd1b	Bump read/write pointer
fd22	Powerup entry
fd3f	Check A-rom
fd52	Set kernal2
fd8d	Initialize system constants
fdf1	IRQ vectors
fdf9	Initialize I/O regs
fe49	Save data name
fe50	Save file details
fe57	Get status
fe66	Flag ST
fe6f	Set timeout
fe73	Read/set top of memory
fe82	Read/set bottom of memory
fe91	Test memory location
fea9	NMI interrupt entry
fed2	RESET/STOP warm start
fede	NMI RS232 sequences
ff56	Restore & exit
ff5c	RS232 timing table
ff72	Main IRQ entry
ff8a	Jumbo jump table
fffa	Hardware vectors

Map 6. Commodore 64 Memory Map.

SID (6581) Commodore 64

V1	V2	V3	Register		V1	V2	V3
D400	D407	D40E	Frequency	L	54272	54279	54286
D401	D408	D40F	Frequency	H	54273	54280	54287
D402	D409	D410	Pulse Width	L	54274	54281	54288
D403	D40A	D411	0 0 0 0 / Pulse Width	H	54275	54282	54289
D404	D40B	D412	Voice Type: NSE PUL SAW TRI	Key	54276	54283	54290
D405	D40C	D413	Attack Time 2 ms - 8 sec	Decay Time 6 ms - 24 sec	54277	54284	54291
D406	D40D	D414	Sustain Level	Release Time 6 ms - 24 sec	54278	54285	54292

Voices
(Write Only)

Address	Register		Decimal
D415	0 0 0 0 0 / Filter Frequency	L	54293
D416	Filter Frequency	H	54294
D417	Resonance	Filter Voices: EXT V3 V2 V1	54295
D418	Passband: V3 Off, Hi, Bd, Lo	Master Volume	54296

Filter & Volume
(Write Only)

D419	Paddle X	54297
D41A	Paddle Y	54298
D41B	Noise 3 (Random)	54299
D41C	Envelope 3	54300

Sense

(Read Only)

Special voice features (TEST, RING MOD, SYNC) are omitted from the above diagram.

CIA 2 (NMI) (6526) Commodore 64

$DD00	Serial In	Clock In	Serial Out	Clock Out	ATN Out	RS-232 Out			PRA	56576
$DD01	DSR In	CTS In		DCD* In	RI* In	DTR Out	RTS* Out	RS-232 In	PRB	56577
	Parallel User Port									
$DD02	IN	IN	Out	Out	Out $3F	Out	Out	Out	DDRA	56578
$DD03	$06 For RS-232								DDRB	56579
$DD04	Timer A								TAL	56580
$DD05									TAH	56581
$DD06	Timer B								TBL	56582
$DD07									TBH	56583
~										
$DD0D				RS-232 In			Timer B	Timer A	ICR	56589
$DD0E								Timer A Start	CRA	56590
$DD0F								Timer B Start	CRB	56591

*Connected but not used by system.

Processor I/O Port (6510) Commodore 64

$0000	IN	IN	Out	IN	Out	Out	Out	Out	DDR	0
$0001			Tape Motor	Tape Sense	Tape Write	D-Rom Switch	EF.RAM Switch	AB.RAM Switch	PR	1

CIA 1 (IRQ) (6526) Commodore 64

$DC00	Paddle SEL A	Paddle SEL B			Joystick 0 R	Joystick 0 L	Joystick 0 D	Joystick 0 U	PRA	56320
	Keyboard Row Select (Inverted)									
$DC01					Joystick 1				PRB	56321
	Keyboard Column Read									
$DC02	$FF — All Output								DDRA	56322
$DC03	$00 — All Input								DDRB	56323
$DC04	Timer A								TAL	56324
$DC05									TAH	56325
$DC06	Timer B								TBL	56326
$DC07									TBH	56327
~										
$DC0D				Tape Input			Timer B	Interr. A	ICR	56333
$DC0E					One Shot	Out Mode	Time PB6 Out	Timer A Start	CRA	56334
$DC0F					One Shot	Out Mode	Time PB7 Out	Timer B Start	CRB	56335

64 Memory Map

Hex	Decimal	Description
0000	0	Chip directional register
0001	1	Chip I/O; memory & tape control
0003-0004	3-4	Float-Fixed vector
0005-0006	5-6	Fixed-Float vector
0007	7	Search character
0008	8	Scan-quotes flag
0009	9	TAB column save
000A	10	0=LOAD, 1=VERIFY
000B	11	Input buffer pointer/# subscrpt
000C	12	Default DIM flag
000D	13	Type: FF=string, 00=numeric
000E	14	Type: 80=integer, 00=floating point
000F	15	DATA scan/LIST quote/memry flag
0010	16	Subscript/FNx flag
0011	17	0=INPUT;$40=GET;$98=READ
0012	18	ATN sign/Comparison eval flag
0013	19	Current I/O prompt flag
0014-0015	20-21	Integer value
0016	22	Pointer: temporary strg stack
0017-0018	23-24	Last temp string vector
0019-0021	25-33	Stack for temporary strings
0022-0025	34-37	Utility pointer area
0026-002A	38-42	Product area for multiplication
002B-002C	43-44	Pointer: Start-of-Basic

002D-002E	45-46	Pointer: Start-of-Variables
002F-0030	47-48	Pointer: Start-of-Arrays
0031-0032	49-50	Pointer: End-of-Arrays
0033-0034	51-52	Pointer: String-storage(moving down)
0035-0036	53-54	Utility string pointer
0037-0038	55-56	Pointer: Limit-of-memory
0039-003A	57-58	Current Basic line number
003B-003C	59-60	Previous Basic line number
003D-003E	61-62	Pointer: Basic statement for CONT
003F-0040	63-64	Current DATA line number
0041-0042	65-66	Current DATA address
0043-0044	67-68	Input vector
0045-0046	69-70	Current variable name
0047-0048	71-72	Current variable address
0049-004A	73-74	Variable pointer for FOR/NEXT
004B-004C	75-76	Y-save; op-save; Basic pointer save
004D	77	Comparison symbol accumulator
004E-0053	78-83	Misc work area, pointers, etc
0054-0056	84-86	Jump vector for functions
0057-0060	87-96	Misc numeric work area
0061	97	Accum#1: Exponent
0062-0065	98-101	Accum#1: Mantissa
0066	102	Accum#1: Sign
0067	103	Series evaluation constant pointer
0068	104	Accum#1 hi-order (overflow)

0069-006E	105-110	Accum#2: Exponent, etc.
006F	111	Sign comparison, Acc#1 vs #2
0070	112	Accum#1 lo-order (rounding)
0071-0072	113-114	Cassette buff len/Series pointer
0073-008A	115-138	CHRGET subroutine; get Basic char
007A-007B	122-123	Basic pointer (within subrtn)
008B-008F	139-143	RND seed value
0090	144	Status word ST
0091	145	Keyswitch PIA: STOP and RVS flags
0092	146	Timing constant for tape
0093	147	Load=0, Verify=1
0094	148	Serial output: deferred char flag
0095	149	Serial deferred character
0096	150	Tape EOT received
0097	151	Register save
0098	152	How many open files
0099	153	Input device, normally 0
009A	154	Output CMD device, normally 3
009B	155	Tape character parity
009C	156	Byte-received flag
009D	157	Direct=$80/RUN=0 output control
009E	158	Tp Pass 1 error log/char buffer
009F	159	Tp Pass 2 err log corrected
00A0-00A2	160-162	Jiffy Clock HML
00A3	163	Serial bit count/EOI flag

00A4	164	Cycle count
00A5	165	Countdown,tape write/bit count
00A6	166	Tape buffer pointer
00A7	167	Tp Wrt ldr count/Rd pass/inbit
00A8	168	Tp Wrt new byte/Rd error/inbit cnt
00A9	169	Wrt start bit/Rd bit err/stbit
00AA	170	Tp Scan;Cnt;Ld;End/byte assy
00AB	171	Wr lead length/Rd checksum/parity
00AC-00AD	172-173	Pointer: tape bufr, scrolling
00AE-00AF	174-175	Tape end adds/End of program
00B0-00B1	176-177	Tape timing constants
00B2-00B3	178-179	Pntr: start of tape buffer
00B4	180	1=Tp timer enabled; bit count
00B5	181	Tp EOT/RS232 next bit to send
00B6	182	Read character error/outbyte buf
00B7	183	# characters in file name
00B8	184	Current logical file
00B9	185	Current secndy address
00BA	186	Current device
00BB-00BC	187-188	Pointer to file name
00BD	189	Wr shift word/Rd input char
00BE	190	# blocks remaining to Wr/Rd
00BF	191	Serial word buffer
00C0	192	Tape motor interlock
00C1-00C2	193-194	I/O start address

00C3-00C4	195-196	Kernel setup pointer
00C5	197	Last key pressed
00C6	198	# chars in keybd buffer
00C7	199	Screen reverse flag
00C8	200	End-of-line for input pointer
00C9-00CA	201-202	Input cursor log (row, column)
00CB	203	Which key: 64 if no key
00CC	204	0=flash cursor
00CD	205	Cursor timing countdown
00CE	206	Character under cursor
00CF	207	Cursor in blink phase
00D0	208	Input from screen/from keyboard
00D1-00D2	209-210	Pointer to screen line
00D3	211	Position of cursor on above line
00D4	212	0=direct cursor, else programmed
00D5	213	Current screen line length
00D6	214	Row where curosr lives
00D7	215	Last inkey/checksum/buffer
00D8	216	# of INSERTs outstanding
00D9-00F2	217-242	Screen line link table
00F3-00F4	243-244	Screen color pointer
00F5-00F6	245-246	Keyboard pointer
00F7-00F8	247-248	RS-232 Rcv pntr
00F9-00FA	249-250	RS-232 Tx pntr
00FF-010A	255-266	Floating to ASCII work area

0100-Ø13E	256-318	Tape error log
0100-01FF	256-511	Processor stack area
0200-0258	512-600	Basic input buffer
0259-0262	601-610	Logical file table
0263-026C	611-620	Device # table
026D-0276	621-630	Sec Adds table
0277-0280	631-640	Keybd buffer
0281-0282	641-642	Start of Basic Memory
0283-0284	643-644	Top of Basic Memory
0285	645	Serial bus timeout flag
0286	646	Current color code
0287	647	Color under cursor
0288	648	Screen memory page
0289	649	Max size of keybd buffer
028A	650	Repeat all keys
028B	651	Repeat speed counter
028C	652	Repeat delay counter
028D	653	Keyboard Shift/Control flag
028E	654	Last shift pattern
028F-0290	655-656	Keyboard table setup pointer
0291	657	Keyboard shift mode
0292	658	0=scroll enable
0293	659	RS-232 control reg
0294	660	RS-232 command reg

0295-0296	661-662	Bit timing
0297	663	RS-232 status
0298	664	# bits to send
0299-029A	665	RS-232 speed/code
029B	667	RS232 receive pointer
029C	668	RS232 input pointer
029D	669	RS232 transmit pointer
029E	670	RS232 output pointer
029F-02A0	671-672	IRQ save during tape I/O
02A1	673	CIA 2 (NMI) Interrupt Control
02A2	674	CIA 1 Timer A control log
02A3	675	CIA 1 Interrupt Log
02A4	676	CIA 1 Timer A enabled flag
02A5	677	Screen row marker
02C0-02FE	704-766	(Sprite 11)
0300-0301	768-769	Error message link
0302-0303	770-771	Basic warm start link
0304-0305	772-773	Crunch Basic tokens link
0306-0307	774-775	Print tokens link
0308-0309	776-777	Start new Basic code link
030A-030B	778-779	Get arithmetic element link
030C	780	SYS A-reg save
030D	781	SYS X-reg save
030E	782	SYS Y-reg save
030F	783	SYS status reg save

0310-0312	784-785	USR function jump	(B248)
0314-0315	788-789	Hardware interrupt vector	(EA31)
0316-0317	790-791	Break interrupt vector	(FE66)
0318-0319	792-793	NMI interrupt vector	(FE47)
031A-031B	794-795	OPEN vector	(F34A)
031C-031D	796-797	CLOSE vector	(F291)
031E-031F	798-799	Set-input vector	(F20E)
0320-0321	800-801	Set-output vector	(F250)
0322-0323	802-803	Restore I/O vector	(F333)
0324-0325	804-805	INPUT vector	(F157)
0326-0327	806-807	Output vector	(F1CA)
0328-0329	808-809	Test-STOP vector	(F6ED)
032A-032B	810-811	GET vector	(F13E)
032C-032D	812-813	Abort I/O vector	(F32F)
032E-032F	814-815	Warm start vector	(FE66)
0330-0331	816-817	LOAD link	(F4A5)
0332-0333	818-819	SAVE link	(F5ED)
033C-03FB	828-1019	Cassette buffer	
0340-037E	832-894	(Sprite 13)	
0380-03BE	896-958	(Sprite 14)	
03C0-03FE	960-1022	(Sprite 15)	
0400-07FF	1024-2047	Screen memory	
0800-9FFF	2048-40959	Basic RAM memory	
8000-9FFF	32768-40959	Alternate: ROM plug-in area	

```
A000-BFFF 40960-49151 ROM:  Basic
A000-BFFF 49060-49151 Alternate:  RAM
C000-CFFF 49152-53247 RAM memory, including alternate
D000-D02E 53248-53294 Video Chip (6566)
D400-D41C 54272-54300 Sound Chip (6581 SID)
D800-DBFF 55296-56319 Color nybble memory
DC00-DC0F 56320-56335 Interface chip 1, IRQ (6526 CIA)
DD00-DD0F 56576-56591 Interface chip 2, NMI (6526 CIA)
D000-DFFF 53248-57343 Alternate:  Character set
E000-FFFF 57344-65535 ROM:  Operating System
E000-FFFF 57344-65535 Alternate:  RAM
FF81-FFF5 65409-65525 Jump Table, Including:
  FFC6 - Set Input channel
  FFC9 - Set Output channel
  FFCC - Restore default I/O channels
  FFCF - INPUT
  FFD2 - PRINT
  FFE1 - Test Stop key
  FFE4 - GET
```

Map 7. Atari Memory.

```
                    ;
                    ;      PAGE ZERO RAM ASSIGNMENTS
                    ;
0000                LINZBS  =      $0000   ;LINBUG RAM (WILL BE REPLACED BY MONITOR RAM)
                    ;
                    ;      THESE LOCATIONS ARE NOT CLEARED
0002                CASINI  =      $0002   ;CASSETTE INIT LOCATION
0004                RAMLO   =      $0004   ;RAM POINTER FOR MEMORY TEST
0006                TRAMSZ  =      $0006   ;TEMPORARY REGISTER FOR RAM SIZE
0007                TSTDAT  =      $0007   ;RAM TEST DATA REGISTER
                    ;
                    ;      CLEARED ON COLD START ONLY
0008                WARMST  =      $0008   ;WARM START FLAG
0009                BOOTQ   =      $0009   ;SUCCESSFUL BOOT FLAG <WAS BOOT?>
000A                DOSVEC  =      $000A   ;DISK SOFTWARE START FLAG
000C                DOSINI  =      $000C   ;DISK SOFTWARE INIT ADDRESS
000E                APPMHI  =      $000E   ;APPLICATIONS MEMORY HI LIMIT
                    ;
                    ;      CLEARED ON A COLD OR WARM START
0010                INTZBS  =      $0010   ;INTERRUPT HANDLER
0010                POKMSK  =      $0010   ;SYSTEM MASK FOR POKEY IRQ HANDLER
0011                BRKKEY  =      $0011   ;BREAK KEY FLAG
0012                RTCLOCK =      $0012   ;REAL TIME CLOCK (IN 16 MSEC UNITS)
                    ;
0015                BUFADR  =      $0015   ;INDIRECT BUFFER ADDRESS REGISTER
                    ;
0017                ICCOMT  =      $0017   ;COMMAND FOR VECTOR
                    ;
0018                DSKFMS  =      $0018   ;DISK FILE MANAGER POINTER
                    ;
```

```
001A        DSKUTL  =   $001A   ;DISK UTILITIES POINTER
                    ;
001C        PTIMOT  =   $001C   ;PRINTER TIME OUT REGISTER
001D        PBPNT   =   $001D   ;PRINTER BUFFER POINTER
001E        PBUFSZ  =   $001E   ;PRINT BUFFER SIZE
001F        PTEMP   =   $001F   ;TEMPORARY REGISTER
                    ;
0020        ZIOCB   =   $0020   ;ZERO PAGE I/O CONTROL BLOCK
0010        IOCBSZ  =   16      ;NUMBER OF BYTES PER IOCB
0080        MAXIOC  =   8*IOCBSZ        ;LENGTH OF THE IOCB AREA
0020        IOCBAS  =   $0020
0020        ICHIDZ  =   $0020   ;HANDLER INDEX NUMBER (FF == IOCB FREE)
0021        ICDNOZ  =   $0021   ;DEVICE NUMBER (DRIVE NUMBER)
0022        ICCOMZ  =   $0022   ;COMMAND CODE
0023        ICSTAZ  =   $0023   ;STATUS OF LAST IOCB ACTION
0024        ICBALZ  =   $0024   ;BUFFER ADDRESS LOW BYTE
0025        ICBAHZ  =   $0025   ;BUFFER ADDRESS HIGH BYTE
0026        ICPTLZ  =   $0026   ;PUT BYTE ROUTINE ADDRESS - 1
0027        ICPTHZ  =   $0027   ;
0028        ICBLLZ  =   $0028   ;BUFFER LENGTH LOW BYTE
0029        ICBLHZ  =   $0029
002A        ICAX1Z  =   $002A   ;AUXILIARY INFORMATION FIRST BYTE
002B        ICAX2Z  =   $002B
002C        ICSPRZ  =   $002C   ;TWO SPARE BYTES (CIO LOCAL USE)
002E        ICIDNO  =   ICSPRZ+2        ;ICOB NUMBER X 16
002F        CIOCHR  =   ICSPRZ+3        ;CHARACTER BYTE FOR CURRENT OPERATION
                    ;
0030        STATUS  =   $0030   ;INTERNAL STATUS STORAGE
0031        CHKSUM  =   $0031   ;CHECKSUM (SINGLE BYTE SUM WITH CARRY)
0032        BUFRLO  =   $0032   ;POINTER TO DATA BUFFER (LO BYTE)
0033        BUFRHI  =   $0033   ;POINTER TO DATA BUFFER (HI BYTE)
0034        BFENLO  =   $0034   ;NEXT BYTE PAST END OF DATA BUFFER (LO BYTE)
```

```
0035        BFENHI  =   $0035   ;NEXT BYTE PAST END OF DATA BUFFER (HI BYTE)
0036        CRETRY  =   $0036   ;NUMBER OF COMMAND FRAME RETRIES
0037        DRETRY  =   $0037   ;NUMBER OF DEVICE RETRIES
0038        BUFRFL  =   $0038   ;DATA BUFFER FULL FLAG
0039        RECVDN  =   $0039   ;RECIEVE DONE FLAG
003A        XMTDON  =   $003A   ;TRANSMISSION DONE FLAG
003B        CHKSNT  =   $003B   ;CHECKSUM SENT FLAG
003C        NOCKSM  =   $003C   ;NO CHECKSUM FOLLOWS DATA FLAG
            ;
            ;
003D        BPTR    =   $003D
003E        FTYPE   =   $003E
003F        FEOF    =   $003F
0040        FREQ    =   $0040
0041        SOUNDR  =   $0041   ;NOISY I/O FLAG (ZERO IS QUIET)
0042        CRITIC  =   $0042   ;DEFINES CRITICAL SECTION (CRITICAL IF NON-ZERO)
            ;
0043        FMSZPG  =   $0043   ;TOTAL OF 7 BYTES FOR DISK FILE MANAGER ZERO PAGE
            ;
            ;
004A        CKEY    =   $004A   ;FLAG SET WHEN GAME START PRESSED
004B        CASSBT  =   $004B   ;CASSETTE BOOT FLAG
004C        DSTAT   =   $004C   ;DISPLAY STATUS
            ;
004D        ATRACT  =   $004D   ;ATRACT FLAG
004E        DRKMSK  =   $004E   ;DARK ATRACT FLAG
004F        COLRSH  =   $004F   ;ATRACT COLOR SHIFTER (EOR'D WITH PLAYFIELD COLORS)
            ;
            ;LEDGE  =   2       ;LMARGN'S VALUE AT COLD START
            ;REDGE  =   39      ;
0050        TMPCHR  =   $0050
0051        HOLD1   =   $0051
```

```
0052        LMARGN  =      $0052   ;LEFT MARGIN (SET TO ONE AT POWER ON)
0053        RMARGN  =      $0053   ;RIGHT MARGIN (SET TO ONE AT POWER ON)
0054        ROWCRS  =      $0054   ;CURSOR COUNTERS
0055        COLCRS  =      $0055
0057        DINDEX  =      $0057
0058        SAVMSC  =      $0058
005A        OLDROW  =      $005A
005B        OLDCOL  =      $005B
005D        OLDCHR  =      $005D   ;DATA UNDER CURSOR
005E        OLDADR  =      $005E
0060        NEWROW  =      $0060   ;POINT DRAW GOES TO
0061        NEWCOL  =      $0061
0063        LOGCOL  =      $0063   ;POINTS AT COLUMN IN LOGICAL LINE
0064        ADRESS  =      $0064
0066        MLTTMP  =      $0066
0068        SAVADR  =      $0068
006A        RAMTOP  =      $006A   ;RAM SIZE DEFINED BY POWER ON LOGIC
006B        BUFCNT  =      $006B   ;BUFFER COUNT
006C        BUFSTR  =      $006C   ;EDITOR GETCH POINTER
006E        BITMSK  =      $006E   ;BIT MASK
            ; LOTS OF RANDOM TEMPS
007B        SWPFLG  =      $007B   ;NON-0 IF TXT AND REGULAR RAM IS SWAPPED
007C        HOLDCH  =      $007C   ;CH IS MOVED HERE IN KGETCH BEfore CNTL & SHIFT PROC
            ;
            ;
            ;
            ;       80 - FF ARE FOR FP, USER, FMS AND DOS
            ;
            ;
            ;       PAGE 1 -- STACK
            ;
            ;
```

```
                ;       PAGE TWO RAM ASSIGNMENTS
                ;
0200            INTABS  =       $0200   ;INTERRUPT RAM
0200            VDBLST  =       $0200   ;DISPLAY LIST NMI VECTOR
0202            VPRCED  =       $0202   ;PROCEED LINE IRQ VECTOR
0204            VINTER  =       $0204   ;INTERRUPT LINE IRQ VECTOR
0206            VBREAK  =       $0206   ;SOFTWARE BREAK (00) INSTRUCTION IRQ VECTOR
0208            VKEYBD  =       $0208   ;POKEY KEYBOARD IRQ VECTOR
020A            VSERIN  =       $020A   ;POKEY SERIAL INPUT READY IRQ
020C            VSEROR  =       $020C   ;POKEY SERIAL OUTPUT READY IRQ
020E            VSEROC  =       $020E   ;POKEY SERIAL OUTPUT COMPLETE IRQ
0210            VTIMR1  =       $0210   ;POKEY TIMER 1 IRQ
0212            VTIMR2  =       $0212   ;POKEY TIMER 2 IRQ
0214            VTIMR4  =       $0214   ;POKEY TIMER 4 IRQ
0216            VIMIRQ  =       $0216   ;IMMEDIATE IRQ VECTOR
0218            CDTMV1  =       $0218   ;COUNT DOWN TIMER 1
021A            CDTMV2  =       $021A   ;COUNT DOWN TIMER 2
021C            CDTMV3  =       $021C   ;COUNT DOWN TIMER 3
021E            CDTMV4  =       $021E   ;COUNT DOWN TIMER 4
0220            CDTMV5  =       $0220   ;COUNT DOWN TIMER 5
0222            VVBLKI  =       $0222   ;IMMEDIATE VERTICAL BLANK NMI VECTOR
0224            VVBLKD  =       $0224   ;DEFERRED VERTICAL BLANK NMI VECTOR
0226            CDTMA1  =       $0226   ;COUNT DOWN TIMER 1 JSR ADDRESS
0228            CDTMA2  =       $0228   ;COUNT DOWN TIMER 2 JSR ADDRESS
022A            CDTMF3  =       $022A   ;COUNT DOWN TIMER 3 FLAG
022B            SRTIMR  =       $022B   ;SOFTWARE REPEAT TIMER
022C            CDTMF4  =       $022C   ;COUNT DOWN TIMER 4 FLAG
022E            CDTMF5  =       $022E   ;COUNT DOWN TIMER 5 FLAG
022F            SDMCTL  =       $022F   ;SAVE DMACTL REGISTER
0230            SDLSTL  =       $0230   ;SAVE DISPLAY LIST LOW BYTE
0231            SDLSTH  =       $0231   ;SAVE DISPLAY LIST HIGH BYTE
0232            SSKCTL  =       $0232   ;SKCTL REGISTER RAM
```

```
                ;
0234            LPENH   =       $0234   ;LIGHT PEN HORIZONTAL VALUE
0235            LPENV   =       $0235   ;LIGHT PEN VERTICAL VALUE

026F            GPRIOR  =       $26F    ;Global priority cell


                ;
                ;
                ;       POTENTIOMETERS
                ;
                ;
0270            PADDL0  =       $0270
0271            PADDL1  =       $0271
0272            PADDL2  =       $0272
0273            PADDL3  =       $0273
0274            PADDL4  =       $0274
0275            PADDL5  =       $0275
0276            PADDL6  =       $0276
0277            PADDL7  =       $0277
                ;
                ;
                ;       JOYSTICKS
                ;
                ;
0278            STICK0  =       $0278
0279            STICK1  =       $0279
027A            STICK2  =       $027A
027B            STICK3  =       $027B
                ;
                ;
```

```
                ;       PADDLE TRIGGER
                ;
                ;
027C            PTRIG0  =       $027C
027D            PTRIG1  =       $027D
027E            PTRIG2  =       $027E
027F            PTRIG3  =       $027F
0280            PTRIG4  =       $0280
0281            PTRIG5  =       $0281
0282            PTRIG6  =       $0282
0283            PTRIG7  =       $0283
                ;
                ;
                ;       JOYSTICK TRIGGER
                ;
                ;
0284            STRIG0  =       $0284
0285            STRIG1  =       $0285
0286            STRIG2  =       $0286
0287            STRIG3  =       $0287
                ;
                ;       Many random OS variables, the following were commented
                ;
0290            TXTROW  =       $0290   ;Text rowcrs
0291            TXTCOL  =       $0291   ;Text colcrs
0293            TINDEX  =       $0293   ;Text index
0294            TXTMSC  =       $0294   ;fools convert into new msc
0296            TXTOLD  =       $0296   ;oldrow and oldcol for text (etc.)
02A2            ESCFLG  =       $02A2   ;Escape flag
02B2            LOGMAP  =       $02B2   ;Logical line start bit map
02B6            INVFLAG =       $02B6   ;Inverse video flag (toggled by Atari key)
02B7            FILFLG  =       $02B7   ;Fill flag for draw
```

```
02BB          SCRFLG  =     $02BB   ;Set if scroll occures
02BE          SHFLOK  =     $02BE   ;Shift lock
02BF          BOTSCR  =     $02BF   ;Bottom of screen: 24 Norm, 4 Split.
              ;
              ;
              ;       COLORS
              ;
              ;
02C0          PCOLR0  =     $02C0   ;P0 COLOR
02C1          PCOLR1  =     $02C1   ;P1 COLOR
02C2          PCOLR2  =     $02C2   ;P2 COLOR
02C3          PCOLR3  =     $02C3   ;P3 COLOR
02C4          COLOR0  =     $02C4   ;COLOR 0
02C5          COLOR1  =     $02C5   ;COLOR 1
02C6          COLOR2  =     $02C6   ;COLOR 2
02C7          COLOR3  =     $02C7   ;COLOR 3
02C8          COLOR4  =     $02C8   ;COLOR 4
              ;
              ;
              ;       GLOBAL VARIABLES
              ;
              ;
02E4          RAMSIZ  =     $02E4   ;RAM SIZE (HI BYTE ONLY)
02E5          MEMTOP  =     $02E5   ;TOP OF AVAILABLE USER MEMORY
02E7          MEMLO   =     $02E7   ,BOTTOM OF AVAILABLE USER MEMORY
02EA          DVSTAT  =     $02EA   ;STATUS BUFFER
              ;
02F0          CRSINH  =     $02F0   ;CURSOR INHIBIT (00 = CURSOR ON)
02F1          KEYDEL  =     $02F1   ;Key delay
02F3          CHACT   =     $02F3   ;CHACTL REGISTER RAM
02F4          CHBAS   =     $02F4   ;CHBAS REGISTER RAM
02FD          FILDAT  =     $02FD   ;RIGHT FILL DATA (DRAW)
```

```
02FB        ATACHR  =       $02FB   ;Atascii character
02FC        CH      =       $02FC   ;global variable for keyboard
02FE        DSPFLA  =       $02FE   ;DISPLAY FLAG: DISPLAYS CNTLS IF NON ZERO;
02FF        SSFLAG  =       $02FF   ;Start/stop flag for paging (CNTL 1). Cleared by Brea

            ;
            ;       Page three RAM assignments
            ;
            ;
            ;       Device control blocks
            ;       (SIO)
0300        DCB     =       $0300   ;Device control block
0300        DDEVIC  =       $0300   ;Peripheral Unit 1 bus I.D. number
0301        DUNIT   =       $0301   ;Unit number
0302        DCOMND  =       $0302   ;Bus command
0303        DSTATS  =       $0303   ;Command Type/status return
0304        DBUFLO  =       $0304   ;Data buffe pointe low
0305        DBUFHI  =       $0305
0306        DTIMLO  =       $0306   ;Device time out in 1 second units
0308        DBYTLO  =       $0308   ;Number of bytes to be transvered low byte
0309        DBYTHI  =       $0309
030A        DAUX1   =       $030A   ;Command Aux byte 1
030B        DAUX2   =       $030B
            ;
0340        IOCB    =       $0340
0340        ICHID   =       $0340   ;Handler index number (FF = IOCB free)
0341        ICDNO   =       $0341   ;Device number (drive number)
0342        ICCOM   =       $0342   ;Command code
0343        ICSTA   =       $0343   ;Status of last IOCB action
0344        ICBAL   =       $0344   ;Buffer address low byte
0345        ICBAH   =       $0345
0346        ICPTL   =       $0346   ;Put byte routine address - 1
```

```
0347        ICPTH   =       $0347
0348        ICBLL   =       $0348   ;Buffer length low byte
0349        ICBLH   =       $0349
034A        ICAX1   =       $034A   ;Auxiliary information first byte
034B        ICAX2   =       $034B
034C        ICSPR   =       $034C   ;four spare bytes
            ;
03C0        PRNBUF  =       $03C0   ;Printer buffer (40 bytes)
            ;       (21 spare bytes)
            ;
            ;       Page Four Ram Assignments
03FD        CASBUF  =       $03FD   ;Cassette Buffer (131 bytes)
            ;
0480        USAREA  =       $0480   ; (0480 thru 05FF for the user)
            ;                         (except for floating point...)
            ;
            ;       FLOATING POINT ROM ROUTINES
            ;
            ;
            ;       IF CARRY IS USED THEN CARRY CLEAR => NO ERROR, CARRY SET => ERROR
            ;
D800        AFP     =       $D800   ;ASCII -> FLOATING POINT (FP)
            ;                       INBUFF + CIX -> FR0, CIX, CARRY
D8E6        FASC    =       $D8E6   ;FP -> ASCII    FR0 -> FOR,FO0+1, CARRY
D9AA        IFP     =       $D9AA   ;INTEGER -> FP
            ;                       0-$FFFF (LSB, MSB) IN FR0,FR0+1->FR0
D92D        FPI     =       $D92D   ;FP -> INTEGER  FR0 -> FR0,FR0+1, CARRY
DA60        FSUB    =       $DA60   ;FR0 <- FR0 - FR1, CARRY
DA66        FADD    =       $DA66   ;FR0 <- FR0 + FR1  ,CARRY
DADB        FMUL    =       $DADB   ;FR0 <- FR0 * FR1  ,CARRY
DB28        FDIV    =       $DB28   ;FR0 <- FR0 / FR1  ,CARRY
DD89        FLD0R   =       $DD89   ;FLOATING LOAD REG0     FR0 <- (X,Y)
```

```
DD8D        FLD0P   =     $DD8D   ;                         FR0 <- (FLPTR)
DD98        FLD1R   =     $DD98   ;                         FR1 <- (X,Y)
DD9C        FLD1P   =     $DD9C   ;                         FR1 <- (FLPTR)
DDA7        FSTOR   =     $DDA7   ;FLOATING STORE REG0  (X,Y) <- FR0
DDAB        FSTOP   =     $DDAB   ;                    (FLTPTR)<- FR0
DDB6        FMOVE   =     $DDB6   ;FR1 <- FR0
DD40        PLYEVL  =     $DD40   ;FR0 <- P(Z) = SUM(I = N TO 0) (A(I) *Z**I)     CARRY

            ;                      INPUT:  (X,Y) = A(N), A(N-1)...A(0) -> PLYARG
            ;                              ACC   = # OF COEFFICIENTS = DEGREE + 1
            ;                              FR0   = Z
DDC0        EXP     =     $DDC0   ;FR0 <- E**FR0 = EXP10(FR0 * LOG10(E))  CARRY
DDCC        EXP10   =     $DDCC   ;FR0 <- 10**FR0 CARRY
DECD        LOG     =     $DECD   ;FR0 <- LN(FR0) = LOG10(FR0) / LOG10(E) CARRY
DED1        LOG10   =     $DED1   ;FR0 <- LOG10(FR0)      CARRY
            ;
            ;
            ;       THE FOLLOWING ARE IN THE BASIC CARTRIDGE:
            ;
            ;
BD81        SIN     =     $BD81   ;FR0 <- SIN(FR0) DEGFLG=0 => RADS, 6=>DEG.      CARRY

BD73        COS     =     $BD73   ;FR0 <- COS(FR0)        CARRY
BD43        ATAN    =     $BD43   ;FR0 <- ATN(FR0)        CARRY
BEB1        SQR     =     $BEB1   ;FR0 <- SQUAREROOT(FR0) CARRY
            ;
            ;
            ;       FLOATING POINT ROUTINES ZERO PAGE (NEEDED ONLY IF F.P.
            ;             ROUTINES ARE CALLED)
00D4        FR0     =     $00D4   ;FP REG0
00E0        FR1     =     $00E0   ;FP REG1
00F2        CIX     =     $00F2   ;CURRENT INPUT INDEX
```

```
00F3        INBUFF  =     $00F3   ;POINTS TO USER'S LINE INPUT BUFFER
00FB        RADFLG  =     $00FB   ;0 = RADIANS, 6 = DEGREES
00FC        FLTPTR  =     $00FC   ;POINTS TO USERS FLOATING POINT NUMBER
            ;
            ;
            ;       FLOATING POINT ROUTINES' NON-ZP RAM
            ;
            ;       (057E to 05FF)

0580        LBUFF   =     $0580   ;LINE BUFFER
05E0        PLYARG  =     LBUFF+$60       ;POLYNOMILA ARGUMENTS
            ;
            ;
            ;
            ;
            ;
            ;
            ;
            ;       COLLEEN MNEMONICS
            ;
            ;
            ;
D200        POKEY   =     $D200           ;VBLANK ACTION:      DESCRIPTION:
D200        POT0    =     POKEY+0         ;POT0-->PADDL0       0-227 IN RAM CELL
D201        POT1    =     POKEY+1         ;POT1-->PADDL1       0-227 IN RAM CELL
D202        POT2    =     POKEY+2         ;POT2-->PADDL2       0-227 IN RAM CELL
D203        POT3    =     POKEY+3         ;POT3-->PADDL3       0-227 IN RAM CELL
D204        POT4    =     POKEY+4         ;POT4-->PADDL4       0-227 IN RAM CELL
D205        POT5    =     POKEY+5         ;POT5-->PADDL5       0-227 IN RAM CELL
D206        POT6    =     POKEY+6         ;POT6-->PADDL6       0-227 IN RAM CELL
D207        POT7    =     POKEY+7         ;POT7-->PADDL7       0-227 IN RAM CELL
```

```
D208        ALLPOT  =   POKEY+8     ;???
D209        KBCODE  =   POKEY+9
D20A        RANDOM  =   POKEY+10
D20B        POTGO   =   POKEY+11    ;Strobed
            ;           n/a
D20D        SERIN   =   POKEY+13
D20E        IRQST   =   POKEY+14
D20F        SKSTAT  =   POKEY+15
            ;
D200        AUDF1   =   POKEY+0
D201        AUDC1   =   POKEY+1
D202        AUDF2   =   POKEY+2
D203        AUDC2   =   POKEY+3
D204        AUDF3   =   POKEY+4
D205        AUDC3   =   POKEY+5
D206        AUDF4   =   POKEY+6
D207        AUDC4   =   POKEY+7
D208        AUDCT   =   POKEY+8     ;NONE                   AUDCTL<--[SIO]
D209        STIMER  =   POKEY+9
D20A        SKRES   =   POKEY+10    ;NONE                   SKRES<--[SIO]
D20B        POTGO   =   POKEY+11
D20D        SEROUT  =   POKEY+13    ;NONE                   SEROUT<--[SIO]
D20E        IRQEN   =   POKEY+14    ;POKMSK-->IRQEN (AFFECTED BY OPEN S: OR E:)
D20F        SKCTL   =   POKEY+15    ;SSKCTL-->SKCTL         SSKCTL<--[SIO]
            ;
D000        CTIA    =   $D000
D000        HPOSP0  =   CTIA+0
D001        HPOSP1  =   CTIA+1
D002        HPOSP2  =   CTIA+2
D003        HPOSP3  =   CTIA+3
D004        HPOSM0  =   CTIA+4
D005        HPOSM1  =   CTIA+5
```

```
D006          HPOSM2  =    CTIA+6
D007          HPOSM3  =    CTIA+7
D008          SIZEP0  =    CTIA+8
D009          SIZEP1  =    CTIA+9
D00A          SIZEP2  =    CTIA+10
D00B          SIZEP3  =    CTIA+11
D00C          SIZEM   =    CTIA+12
D00D          GRAFP0  =    CTIA+13
D00E          GRAFP1  =    CTIA+14
D00F          GRAFP2  =    CTIA+15
D010          GRAFP3  =    CTIA+16
D011          GRAFM   =    CTIA+17
D012          COLPM0  =    CTIA+18      ;PCOLR0-->COLPM0           WITH ATTRACT MODE

D013          COLPM1  =    CTIA+19      ;ETC.N
D014          COLPM2  =    CTIA+20
D015          COLPM3  =    CTIA+21
D016          COLPF0  =    CTIA+22
D017          COLPF1  =    CTIA+23
D018          COLPF2  =    CTIA+24
D019          COLPF3  =    CTIA+25
D01A          COLBK   =    CTIA+26
D01B          PRIOR   =    CTIA+27
D01C          VDELAY  =    CTIA+28
D01D          GRACTL  =    CTIA+29
D01E          HITCLR  =    CTIA+30
D01F          CONSOL  =    CTIA+31      ;$08-->CONSOL              TURN OFF SPEAKER

              ;
D000          M0PF    =    CTIA+0
D002          M2PF    =    CTIA+2
D003          M3PF    =    CTIA+3
```

```
D004    P0PF    =   CTIA+4
D005    P1PF    =   CTIA+5
D006    P2PF    =   CTIA+6
D007    P3PF    =   CTIA+7
D008    M0PL    =   CTIA+8
D009    M1PL    =   CTIA+9
D00A    M2PL    =   CTIA+10
D00B    M3PL    =   CTIA+11
D00C    P0PL    =   CTIA+12
D00D    P1PL    =   CTIA+13
D00E    P2PL    =   CTIA+14
D00F    P3PL    =   CTIA+15
D010    TRIG0   =   CTIA+16    ;TRIG0-->STRIG1
D011    TRIG1   =   CTIA+17    ;ETC.
D012    TRIG2   =   CTIA+18
D013    TRIG3   =   CTIA+19
D014    PAL     =   CTIA+20
        ;
D400    ANTIC   =   $D400
D400    DMACTL  =   ANTIC+0    ;DMACTL<--SDMCTL    ON OPEN S: OR E:

D401    CHARCTL =   ANTIC+1    ;CHACTL<--CHACT     ON OPEN S: OR E:

D402    DLISTL  =   ANTIC+2    ;DLISTL<--SDLSTL    ON OPEN S: OR E:

D403    DLISTH  =   ANTIC+3    ;DLISTH<--SDLSTH    ON OPEN S: OR E:

D404    HSCROL  =   ANTIC+4
D405    VSCROL  =   ANTIC+5
D407    PMBASE  =   ANTIC+7
D409    CHBASE  =   ANTIC+9    ;CHBASE<--CHBAS     ON OPEN S: OR E:
```

```
D40A        WSYNC   =    ANTIC+10
D40B        VCOUNT  =    ANTIC+11
D40C        PENH    =    ANTIC+12
D40D        PENV    =    ANTIC+13
D40E        NMIEN   =    ANTIC+14         ;NMIEN<--40              POWER ON AND [SETVBV]
D40F        NMIRES  =    ANTIC+15         ;STROBED
D40F        NMIST   =    ANTIC+15
E400        EDITRV  =    $E400   ;EDITOR
E410        SCRENV  =    $E410   ;TELEVISION SCREEN
E420        KEYBDV  =    $E420   ;KEYBOARD
E430        PRINTV  =    $E430   ;PRINTER
E440        CASETV  =    $E440   ;CASSETTE
E459        SIOV    =    $E459   ;serial input output routine
E45C        SETVBV  =    $E45C   ;set system timers routine
            ; With respect to SETVBV, the call sequence is
            ; X - MSB of vector/timer
            ; Y - LSB of vector/timer
            ; A - # of vector to hack
0001        SETMR1  =    1       ;Timer  1
0002        SETMR2  =    2       ;       2
0003        SETMR3  =    3       ;       3
0004        SETMR4  =    4       ;       4
0005        SETMR5  =    5       ;       5
0006        SETIMM  =    6       ;Immediate VBLANK
0007        SETDEF  =    7       ;Deffered VBLANK


E45F        SYSVBV  =    $E45F   ;SYSTEM VERTICAL BLANK CALCULATIONS
E462        XITVBL  =    $E462   ;EXIT VERTICAL BLANK CALCULATIONS
E465        SIOINV  =    $E465   ;SERIAL INPUT OUTPUT INITIALIZATION
```

For Further Reference

Apple: *What's Where In The Apple,* William Luebbert, Micro Ink, Inc., 34 Chelmsford St., Chelmsford, MA 01824, 1981.

Atari: *Mapping The Atari,* Ian Chadwick, *COMPUTE!* Books, P.O. Box 5406, Greensboro, NC 27403, 1983. (This covers the operating system and provides lengthy cross-referenced explanations of Atari's memory addresses.)

Atari: *The Atari BASIC Sourcebook,* Bill Wilkinson, *COMPUTE!* Books, P.O. Box 5406, Greensboro, NC 27403, 1983. (Complete commented source code of Atari BASIC, with explanatory text.)

Appendix C
Simple Assembler
Notes On Assembling

This program is written in BASIC because there is no reason not to. Since the program runs quickly enough and there is some complicated arithmetic involved, BASIC is the language of choice. There are assemblers in ML which make two ''passes'' through the source code and do need the extra speed. But this is a simple, ''one-pass'' assembler. The virtue of simplicity is that you can easily and quickly make small ML routines, test them, and debug them. An added bonus is that modifying the Simple Assembler is easy in BASIC. We'll see how you can customize it in a minute.

The assembler accepts your opcodes and their arguments, translates them into the correct numeric values, and POKEs them into RAM memory. You have a choice between using hex or decimal during your ML programming on the Simple Assembler (SA). If you remove line 10, the SA will accept only decimal numbers as arguments, will print all addresses in decimal, and will display the object code (the numbers it is POKEing) in decimal. Leaving line 10 in the program will result in the SA accepting, addressing, and displaying only hexadecimal numbers.

The circumflex in lines 4010 and 5030 — the character following the number 16 — means ''to the power of'' and generally appears on computer keyboards as an arrow pointing up. Since this is not a complicated assembler, a decision had to be made concerning whether or not to include two of the conventions which have been traditional in ML programming. They were left out because it saves programming time to avoid them and they are unnecessary.

The first one is the dollar sign ($). When an assembler can accept either hex or decimal *simultaneously* it must have a way to tell, if you type in ''10'', whether you mean decimal 10 or hex 10 (decimal 16). The convention requires that you write decimal ten as ''10'' and hex as ''$10.''However, this can quickly become a burden. In the SA, you let it know which kinds of numbers you are using by setting H in line ten. After that, just type in the numbers. No $ is used. The second convention that is not included in the SA is the use of the comma. Again, there is no particular reason to use commas, but it has been the tradition to include them for certain addressing modes. They, too, can become burdensome when you are programming. Also, each line of your ML program is brought into the computer via the INPUT statement in line 240. Microsoft BASIC's INPUT statement dislikes seeing commas. So, it is expedient in several ways to drop the comma convention. There is just no reason to use them.

One additional note. The SA does not accept the indirect jump: JMP ($0FFF). You could add it if you wish, but because of a bug in the 6502, it is far safer to avoid it.

Here is a list of the traditional conventions used in most assemblers compared to the simplified conventions of the SA. Notice that each addressing mode has its own appearance, its own punctuation. This is how an assembler knows which addressing mode you mean to use.

Spaces *are important.*

Addressing Mode Conventions	Simple Assembler	Traditional
Immediate	LDA #15	LDA #$15
Absolute	LDA 1500	LDA $1500
Zero Page	LDA 15	LDA $15 (sometimes LDA *$15)
Accumulator	ASL	ASL A
Zero Page, X	LDA 15X	LDA $15,X
Zero Page, Y	LDX 15Y	LDX $15,Y
Absolute, X	LDA 1500X	LDA $1500,X
Absolute, Y	LDA 1500Y	LDA $1500,Y
Indexed Indirect	LDA (15X)	LDA ($15,X)
Indirect Indexed	LDA (15)Y	LDA ($15),Y

Customizing The Simple Assembler

An assembler is only supposed to get your typed opcodes and their arguments, translate them into the right numbers, and put them in memory for you. Nevertheless, the assembler is there for your benefit and it *is* a computer program. It can be taught to do whatever else would assist you in your ML programming. This is where "pseudo-ops" come in. They are not part of the 6502 ML instruction set. They are false opcodes. When you enter one of these, the assembler doesn't put it into 6502 and POKE it. It can't. It does something for you like figure out the hex equivalent of a decimal number or whatever.

The SA has four built-in pseudo-ops and you can add others. Following the input of the opcode (line 240) there is a short quiz. The first question the computer asks itself is: "did they type the word 'FORWARD'?" If so, it means that you are planning to branch forward, but you don't yet know how far. It will make a mental note of this and later, when you type in another pseudo-op, "RESOLVE," it will go back and put in the correct address for the branch. Also, you can hand-POKE in any number in any address by typing the pseudo-op ''POKE''. And, when you are finished with a program, type ''END'' and the assembler will quit, reporting the starting and ending addresses of your program in decimal.

A full-featured assembler can include dozens of pseudo-ops. Let's briefly examine several popular ones to see if there are some that you might want to add to the SA. Then we'll add a hex/decimal pseudo-op to the SA to show how it's done.

BA — Begin Assembly. The SA asks you directly for the starting address (variable SA$). BA signifies the location in RAM memory where you want the object code to start. Example: BA $0400

BY — Bytes. This is for the creation of data tables. The BY is followed by numbers or text characters which are POKEd into memory at the current address. You put these BYtes at the start or end of a program (it could result in havoc if it were in the middle of a program; they would likely be meaningless as instructions). Example: BY 46 46 48 42 12 11 or BY ''THIS IS A MESSAGE''

DE — Define a label. Labels require a two-pass assembler that goes through the source code first to create a table of labels which would look something like this:

```
START           1500
LETTER.A        65
PRINTROUTINE    64422
```

Then, the second time through your source code, the assembler would replace all the labels with their correct values. This is called ''resolving'' the labels. DE is usually part of the initialization process. A number of the example programs in this book start off with a series of DE pseudo-ops, telling the assembler the meaning of various important labels that will be used later in the source code instead of literal numbers. Example: START DE 1500 or LETTER.A DE 65.

EN — The end of the source program. Stop assembling at this point. The SA uses END.

MC — Move code. This interesting pseudo-op takes care of a problem that sometimes comes up when you want your object code to be ultimately used in an address that is now being used by the assembler itself or cannot be directly POKEd at this time with the object code. For instance, if your computer's RAM memory starts at address 2048 like the Commodore 64, and you want to put your final ML object code there, what do you do? If the SA was told to start assembly there, it would begin to nibble away at itself. It's in RAM starting at 2048.

To allow you to store object code elsewhere, but have it *assembled* appropriately for final use in 2048, you could instruct the assembler:

MC 25000 (temporarily store it here)
BA 2048 (but make internal JMPs, JSRs, and table references correct for this starting address)

You can add your own pseudo-ops to the SA following line 240. Many times when you are working along in hex you will want to know the decimal equivalent of a number and vice versa. It's nice to be able to just ask for the translation right during assembling. The answer is printed on the screen and you continue on with your programming. The assembler will do nothing to the ML during all this; it's just giving you an answer.

If you are working in the hex mode and want a decimal number, just type DECIMAL and the computer will accept a hex number from you and give back its decimal equivalent. Conversely, type HEX and give a decimal number for that translation.

To include this pseudo-op in the SA, add the following lines:

Program C-1. Adding The Conversion Pseudo-op.

```
245 IFMN$="HEX"THENGOTO7000
246 IFMN$="DECIMAL"THENGOTO7200
7000 PRINT"ENTER DECIMAL NUMBER";:INPUTDE:IFD
    E>255THENSZ=3:GOTO7020
7010 SZ=1
7020 GOSUB4000:PRINT"                    = $
    "H$:GOTO230
7200 PRINT"ENTER HEX NUMBER";:INPUTH$
7210 SX=LEN(H$):BK$="000":H$=LEFT$(BK$,4-SX)+
    H$
7220 GOSUB5000:PRINT"                    = "
    DE:GOTO230
```

The Simple Assembler has a few error messages that it will print when it can't make sense out of something. The primary responsibility for finding errors, however, is yours. You can create and save ML routines and then look at them with the Disassembler to see if they look like they should. SA takes up about 4.5K so it will not run on an unexpanded VIC. A 3K RAM expansion will provide 2000 bytes for storage of your ML routines.

Program C-2. Simple Assembler (VIC, PET, Apple, 64 Version).

```
10 H=1:REM IF H = 0 THEN ASSEMBLY IS IN DEC
    IMAL
50 HE$="0123456789ABCDEF":SZ=1:ZO$="000"
100 PRINT"   SIMPLE   ASSEMBLER  CONVENTIONS
    :"
110 DIMM$(56),TY(56),OP(56)
120 FORI=1TO56:READM$(I)
122 ROP$=MID$(M$(I),4,1):TY(I)=VAL(ROP$)
124 OP$=RIGHT$(M$(I),3):OP(I)=VAL(OP$)
126 M$(I)=LEFT$(M$(I),3)
140 NEXTI: PRINT
150 PRINT"IMMEDIATE      LDA #15
155 PRINT"ABSOLUTE       LDA 1500
160 PRINT"ZERO PAGE      LDA 15
165 PRINT"ACCUMULATOR    ASL
170 PRINT"INDIRECT X     LDA (15X)
175 PRINT"INDIRECT Y     LDA (15)Y
177 PRINT"ZERO PAGE X    LDA 15X
179 PRINT"ZERO PAGE Y    LDX 15Y
180 PRINT"ABSOLUTE X     LDA 1500X
185 PRINT"ABSOLUTE Y     LDA 1500Y
189 PRINT:PRINT"    ENTER ALL NUMBERS IN ";
190 IFH=1 THENPRINT"HEX":GOTO200
195 PRINT"DECIMAL"
200 PRINT:PRINT"PLEASE INPUT STARTING ADDRES
    S FOR ML PROGRAM":INPUT SA$
210 IFH=1THENH$=SA$:GOSUB5000:SA=DE:GOTO220
215 SA=VAL(SA$)
220 TA=SA:PRINT"{CLEAR}":REM CLEAR THE SCREE
    N
230 IFH=1THENDE=SA:SZ=3:GOSUB4000:PRINTH$;:G
    OTO240
235 PRINTSA"  ";
240 INPUTMN$:PRINT"{UP}"SPC(20);:REM GO UP O
    NE LINE AND OVER 20 SPACES
241 REM ADD NEW PSEUDO-OPS HERE
242 IFRIGHT$(MN$,7)="FORWARD"THENFB=SA
243 IFRIGHT$(MN$,7)="RESOLVE"THENFR=SA-FB:PO
    KEFB+1,FR-2:PRINT"  OK":GOTO230
244 IFRIGHT$(MN$,4)="POKE"THENPRINT"ADDR,NUM
    BER(DEC)";:INPUTADR,NUM:POKEADR,NUM
    :GOTO230
```

```
250 IFMN$="END"THENPRINT:PRINT"        PROGRAM
     IS FROM"TA"TO"SA:END
260 L=LEN(MN$):L$=LEFT$(MN$,3)
270 FORI=1TO56:IFL$=M$(I)THEN300
280 NEXTI
290 GOTO850
300 REM PRIMARY OPCODE CATEGORIES
301 TY=TY(I):OP=OP(I)
305 IFFB=SATHENTN=0:GOTO2010
310 IFTY=0THENGOTO1000
320 IFTY=3THENTY=1:IFL=3THENOP=OP+8:GOTO1000
330 R$=RIGHT$(MN$,L-4):IFH=1THENGOSUB6000
340 LR$=LEFT$(R$,1):LL=LEN(R$):IFLR$="#"THEN
    480
350 IFLR$="("THEN520
360 IFTY=8THEN600
370 IFTY=3THENOP=OP+8:GOTO1000
380 IFRIGHT$(R$,1)="X"ORRIGHT$(R$,1)="Y"THEN
    630
390 IFLEFT$(L$,1)="J"THEN820
400 TN=VAL(R$):IFTN>255THEN430
410 IFTY=1ORTY=3ORTY=4ORTY=5THENOP=OP+4
420 GOTO2000
430 H%=TN/256:L%=TN-256*H%:IFTY=2ORTY=7THENO
    P=OP+8:GOTO470
440 IFTY=1ORTY=3ORTY=4ORTY=5THENOP=OP+12:GOT
    O470
450 IFTY=6ORTY=9THEN470
460 GOTO850
470 GOTO3000
480 TN=VAL(RIGHT$(R$,LL-1))
490 IFTY=1THENOP=OP+8:GOTO2000
500 IFTY=4ORTY=5THENGOTO2000
510 GOTO850
520 IFRIGHT$(R$,2)=")Y"THEN540
530 IFRIGHT$(R$,2)="X)"THEN570
540 TN=VAL(MID$(R$,2,LL-3))
550 IFTY=1THENOP=OP+16:GOTO2000
560 GOTO850
570 TN=VAL(MID$(R$,2,LL-3))
580 IFTY=1THENGOTO2000
590 GOTO850
600 TN=VAL(R$):TN=TN-SA-2:IFTN<-128ORTN>127T
    HENPRINT"TOO FAR ";:GOTO850
```

```
610 IFTN<0THENTN=TN+256
620 GOTO2000
630 IFRIGHT$(R$,2)=")Y"THEN540
640 IFRIGHT$(R$,1)="X"THEN720
650 REM *ZERO Y
660 TN=VAL(LEFT$(R$,LL-1)):IFTN>255THEN680
670 IFTY=2ORTY=5THEN730
675 IFTY=1THEN760
680 GOSUB770:IFTY=1THENOP=OP+24:GOTO710
690 IFTY=5THENOP=OP+28:GOTO710
700 GOTO850
710 GOTO3000
720 TN=VAL(LEFT$(R$,LL-1)):IFTN>255THENGOSUB
    770:GOTO780
730 IFTY=2THENOP=OP+16:GOTO760
740 IFTY=1ORTY=3ORTY=5THENOP=OP+20:GOTO760
750 GOTO850
760 GOTO2000
770 H%=TN/256:L%=TN-256*H%:RETURN
780 IFTY=2THENOP=OP+24:GOTO810
790 IFTY=1ORTY=3ORTY=5THENOP=OP+28:GOTO810
800 GOTO850
810 GOTO3000
820 TN=VAL(R$)
830 GOSUB770
840 GOTO710
850 PRINT"{REV} ERROR ":GOTO230
1000 REM 1 BYTE INSTRUCTIONS
1010 POKESA,OP:SA=SA+1:IFH=1THEN 1030
1020 PRINTOP:GOTO230
1030 DE = OP:GOSUB4000:PRINTH$:GOTO230
2000 REM 2 BYTE INSTRUCTIONS
2005 IFTN>256THENPRINT" INCORRECT ARGUMENT. (
    #5 IN HEX IS #05)":GOTO230
2010 POKESA,OP:POKESA+1,TN:SA=SA+2:IFH=1THEN2
    030
2020 PRINTOP;TN:GOTO230
2030 DE = OP:GOSUB4000:PRINTH$" ";
2040 DE = TN:GOSUB4000:PRINTH$:GOTO230
3000 REM 3 BYTE INSTRUCTIONS
3010 POKESA,OP:POKESA+1,L%:POKESA+2,H%:SA=SA+
    3:IFH=1THEN3030
3020 PRINTOP;L%;H%:GOTO230
3030 DE = OP:GOSUB4000:PRINTH$" ";
```

```
3040 DE = L%:GOSUB4000:PRINTH$" ";
3050 DE = H%:GOSUB4000:PRINTH$:GOTO230
4000 REM  DECIMAL TO HEX (DE TO H$)
4010 H$="":FORM=SZTO0STEP-1:N%=DE/(16^M):DE=D
     E-N%*16^M:H$=H$+MID$(HE$,N%+1,1)
4020 NEXT:SZ=1:RETURN
5000 REM  HEX TO DECIMAL (H$ TO DE)
5010 D=0:Q=3:FORM=1TO4:FORW=0TO15:IFMID$(H$,M
     ,1)=MID$(HE$,W+1,1)THEN5030
5020 NEXTW
5030 D1=W*(16^(Q)):D=D+D1:Q=Q-1:NEXTM:DE=INT(
     D):RETURN
6000 REM ACCEPT HEX OPCODE INPUT AND TRANSLAT
     E IT TO DECIMAL
6010 IFLEFT$(R$,1)="#"THENH$="00"+RIGHT$(R$,2
     ):GOSUB5000:R$="#"+STR$(DE):RETURN
6020 LS=LEN(R$):AZ$=LEFT$(R$,1):ZA$=MID$(R$,L
     S,1):IFAZ$<>"("THEN6050
6030 IFZA$="Y"THENH$="00"+MID$(R$,2,2):GOSUB5
     000:R$="("+STR$(DE)+")Y":RETURN
6040 IFZA$=")"THENH$="00"+MID$(R$,2,2):GOSUB5
     000:R$="("+STR$(DE)+"X)":RETURN
6050 IFZA$="X"ORZA$="Y"THEN6070
6060 H$=LEFT$(ZO$,4-LS)+R$:GOSUB5000:R$=STR$(
     DE):RETURN
6070 IFLS=5THENH$=LEFT$(R$,4):GOTO6090
6080 H$="00"+LEFT$(R$,2)
6090 GOSUB5000:R$=STR$(DE)+ZA$:RETURN
20000 DATAADC1097,AND1033,ASL3002,BCC8144,
      BCS8176,BEQ8240,BIT7036,BMI8048
20010 DATABNE8208,BPL8016,BRK0000,BVC8080,BVS8
     112,CLC0024,CLD0216,CLI0088
20020 DATACLV0184,CMP1193,CPX4224,CPY4192,DEC2
     198,DEX0202,DEY0136,EOR1065
20030 DATAINC2230,INX0232,INY0200,JMP6076,JSR9
     032,LDA1161,LDX5162,LDY5160
20040 DATALSR3066,NOP0234,ORA1001,PHA0072,PHP0
     008,PLA0104,PLP0040,ROL3034
20050 DATAROR3098,RTI0064,RTS0096,SBC1225,SEC0
     056,SED0248,SEI0120,STA1129
20060 DATASTX2134,STY2132,TAX0170,TAY0168,TSX0
     186,TXA0138,TXS0154,TYA0152
```

Program C-3. Simple Assembler: Atari Version.

```
10 HX=1:REM IF HX= 0 THEN ASSEMBLY I
   S IN DECIMAL
20 DIM HE$(16),ZO$(3),R$(10),MN$(12)
   ,ZA$(1),AZ$(1),L$(3),SA$(4),H$(4)
   ,LR$(1)
30 OPEN #1,12,0,"E:"
50 HE$="0123456789ABCDEF":SZ=1:ZO$="
   000"
100 PRINT "{3 SPACES}SIMPLE
    {3 SPACES}ASSEMBLER  CONVENTIONS
    B";
110 DIM M$(56*3),TY(56),OP(56)
120 FOR I=1 TO 56:READ MN$:M$(I*3-2,
    I*3)=MN$(1,3)
122 TY(I)=VAL(MN$(4,4)):OP(I)=VAL(MN
    $(5))
130 NEXT I
140 PRINT :?
150 PRINT "Immediate{5 SPACES}LDA #1
    5"
155 PRINT "Absolute{6 SPACES}LDA 150
    0"
160 PRINT "Zero page{5 SPACES}LDA 15
    "
165 PRINT "Accumulator{3 SPACES}ASL"
170 PRINT "Indirect X{4 SPACES}LDA (
    15X)"
175 PRINT "Indirect Y{4 SPACES}LDA (
    15)Y"
177 PRINT "Zero page X{3 SPACES}LDA
    15X"
179 PRINT "Zero page Y{3 SPACES}LDX
    15Y"
180 PRINT "Absolute X{4 SPACES}LDA 1
    500X"
185 PRINT "Absolute Y{4 SPACES}LDA 1
    500Y"
189 PRINT :PRINT "{4 SPACES}Enter al
    l numbers in ";
190 IF HX=1 THEN PRINT "hexe";
```

```
195 PRINT "decimal"
197 ? :? "Addresses:Use 1536-1791 ($
    0600-$06FF)":? :?
200 PRINT "{2 DEL LINE}Please enter
    starting":? "address for ML prog
    ram";:INPUT SA$:IF SA$="" THEN ?
     "{2 UP}";:GOTO 200
210 IF HX=1 THEN H$=SA$:GOSUB 5000:S
    A=DE:GOTO 217
215 SA=VAL(SA$)
217 IF SA<256 OR SA>=40960 THEN ? "
    {4 UP}Not ZPAGE or ROM!":? :GOTO
     200
220 TA=SA:PRINT "{CLEAR}":GOTO 230
225 ? :? "{BELL} INPUT ERROR ":? :IF
     HX=1 THEN ? "(e.g. #5 should be
     #05)":?
230 IF HX=1 THEN DE=SA:SZ=3:GOSUB 40
    00:PRINT H$;": ";:GOTO 240
235 PRINT SA;": ";
240 TRAP 225:INPUT #1;MN$:? "{UP}";:
    POKE 85,20:IF MN$="" THEN ? "
    {DEL LINE}";:GOTO 230
241 REM ADD NEW PSEUDO-OPS HERE
242 IF LEN(MN$)>6 THEN IF MN$(LEN(MN
    $)-6)="FORWARD" THEN FB=SA
243 IF MN$="RESOLVE" THEN FR=SA-FB:P
    OKE FB+1,FR-2:PRINT "  OK":GOTO
    230
244 IF MN$="POKE" THEN PRINT "ADDR,N
    UMBER(DEC)";:INPUT ADDR,NUM:POKE
     ADDR,NUM:GOTO 230
250 IF MN$="END" THEN 8000
260 L=LEN(MN$):L$=MN$(1,3)
270 FOR I=1 TO 56:IF L$=M$(I*3-2,I*3
    ) THEN 300
280 NEXT I
290 GOTO 850
300 REM PRIMARY OPCODE CATEGORIES
301 TY=TY(I):OP=OP(I)
305 IF FB=SA THEN TN=0:GOTO 2010
310 IF TY=0 THEN GOTO 1000
```

```
32Ø IF TY=3 THEN TY=1:IF L=3 THEN OP
    =OP+8:GOTO 1ØØØ
33Ø R$=MN$(5):IF HX=1 THEN GOSUB 6ØØ
    Ø
34Ø LR$=R$(1,1):LL=LEN(R$):IF LR$="#
    " THEN 48Ø
35Ø IF LR$="(" THEN 52Ø
36Ø IF TY=8 THEN 6ØØ
37Ø IF TY=3 THEN OP=OP+8:GOTO 1ØØØ
38Ø IF R$(LL)="X" OR R$(LL)="Y" THEN
     63Ø
39Ø IF L$(1,1)="J" THEN 82Ø
4ØØ TN=VAL(R$):IF TN>255 THEN 43Ø
41Ø IF TY=1 OR TY=3 OR TY=4 OR TY=5
    THEN OP=OP+4
42Ø GOTO 2ØØØ
43Ø H=INT(TN/256):L=(TN-256*H):IF TY
    =2 OR TY=7 THEN OP=OP+8:GOTO 47Ø
44Ø IF TY=1 OR TY=3 OR TY=4 OR TY=5
    THEN OP=OP+12:GOTO 47Ø
45Ø IF TY=6 OR TY=9 THEN 47Ø
46Ø GOTO 85Ø
47Ø GOTO 3ØØØ
48Ø TN=VAL(R$(2))
49Ø IF TY=1 THEN OP=OP+8:GOTO 2ØØØ
5ØØ IF TY=4 OR TY=5 THEN GOTO 2ØØØ
51Ø GOTO 85Ø
52Ø IF R$(LL-1)=")Y" THEN 54Ø
53Ø IF R$(LL-1)="X)" THEN 57Ø
54Ø TN=VAL(R$(2,LL-1))
55Ø IF TY=1 THEN OP=OP+16:GOTO 2ØØØ
56Ø GOTO 85Ø
57Ø TN=VAL(R$(2,LL-1))
58Ø IF TY=1 THEN GOTO 2ØØØ
59Ø GOTO 85Ø
6ØØ TN=VAL(R$):TN=TN-SA-2:IF TN<-128
     OR TN>127 THEN PRINT "TOO FAR";
    :GOTO 85Ø
61Ø IF TN<Ø THEN TN=TN+256
62Ø GOTO 2ØØØ
63Ø IF R$(LL-1)=")Y" THEN 54Ø
64Ø IF R$(LL-1)="X" THEN 72Ø
```

```
650 REM *ZERO Y
660 TN=VAL(R$(1,LL-1)):IF TN>255 THE
    N 680
670 IF TY=2 OR TY=5 THEN 730
675 IF TY=1 THEN 760
680 GOSUB 770:IF TY=1 THEN OP=OP+24:
    GOTO 710
690 IF TY=5 THEN OP=OP+28:GOTO 710
700 GOTO 850
710 GOTO 3000
720 TN=VAL(R$(1,LL-1)):IF TN>255 THE
    N GOSUB 770:GOTO 780
730 IF TY=2 THEN OP=OP+16:GOTO 760
740 IF TY=1 OR TY=3 OR TY=5 THEN OP=
    OP+20:GOTO 760
750 GOTO 850
760 GOTO 2000
770 H=INT(TN/256):L=TN-256*H:RETURN
780 IF TY=2 THEN OP=OP+24:GOTO 810
790 IF TY=1 OR TY=3 OR TY=5 THEN OP=
    OP+28:GOTO 810
800 GOTO 850
810 GOTO 3000
820 TN=VAL(R$)
830 GOSUB 770
840 GOTO 710
850 PRINT "{BELL}ERROR ":GOTO 230
1000 REM 1 BYTE INSTRUCTIONS
1010 POKE SA,OP:SA=SA+1:IF HX=1 THEN
      1030
1020 PRINT OP:GOTO 230
1030 DE=OP:GOSUB 4000:PRINT H$:GOTO
     230
2000 REM 2 BYTE INSTRUCTIONS
2005 IF TN>256 THEN ? :? "Error--";T
     N;">256 ($100)":GOTO 230
2010 POKE SA,OP:POKE SA+1,TN:SA=SA+2
     :IF HX=1 THEN 2030
2020 PRINT OP;" ";TN:GOTO 230
2030 DE=OP:GOSUB 4000:PRINT H$;" ";
2040 DE=TN:GOSUB 4000:PRINT H$:GOTO
     230
```

```
3000 REM 3 BYTE INSTRUCTIONS
3010 POKE SA,OP:POKE SA+1,L:POKE SA+2,H:SA=SA+3:IF HX=1 THEN 3030
3020 PRINT OP;" ";L;" ";H:GOTO 230
3030 DE=OP:GOSUB 4000:PRINT H$;" ";
3040 DE=L:GOSUB 4000:PRINT H$;" ";
3050 DE=H:GOSUB 4000:PRINT H$:GOTO 230
4000 REM DECIMAL TO HEX (DE TO H$)
4010 H$="":A=INT(DE/256):IF A>0 THEN AH=INT(A/16):AL=A-AH*16:H$=HE$(AH+1,AH+1):H$(2)=HE$(AL+1,AL+1)
4020 A=DE-A*256:AH=INT(A/16):AL=A-AH*16:H$(LEN(H$)+1)=HE$(AH+1,AH+1):H$(LEN(H$)+1)=HE$(AL+1,AL+1):SZ=1:RETURN
5000 REM HEX TO DECIMAL (H$ TO DE)
5010 D=0:Q=3:FOR M=1 TO 4:W=ASC(H$(M))-48:IF W>9 THEN W=W-7
5030 D=D*16+W:NEXT M:DE=INT(D):RETURN
6000 REM ACCEPT HEX OPCODE INPUT AND TRANSLATE IT TO DECIMAL
6010 IF R$(1,1)="#" THEN H$="00":H$(3)=R$(2):GOSUB 5000:R$="#":R$(2)=STR$(DE):RETURN
6020 LS=LEN(R$):AZ$=R$(1,1):ZA$=R$(LS):IF AZ$<>"(" THEN 6050
6030 IF ZA$="Y" THEN H$="00":H$(3)=R$(2,4):GOSUB 5000:R$="(":R$(2)=STR$(DE):R$(LEN(R$)+1)=")Y":RETURN
6040 IF ZA$=")" THEN H$="00":H$(3)=R$(2,4):GOSUB 5000:R$="(":R$(2)=STR$(DE):R$(LEN(R$)+1)="X)":RETURN
6050 IF ZA$="X" OR ZA$="Y" THEN 6070
6060 H$="":IF LS<4 THEN H$=ZO$(1,4-LS)
6065 H$(LEN(H$)+1)=R$:GOSUB 5000:R$=STR$(DE):RETURN
```

```
6070 IF LS=5 THEN H$=R$(1,4):GOTO 60
     90
6080 H$="00":H$(3)=R$(1,2)
6090 GOSUB 5000:R$=STR$(DE):R$(LEN(R
     $)+1)=ZA$:RETURN
8000 PRINT :PRINT "*STARTS ";TA;:SZ=
     3:DE=TA:GOSUB 4000:PRINT " ($";
     H$;")"
8010 PRINT " ENDS{3 SPACES}";SA;:DE=
     SA:SZ=3:GOSUB 4000:PRINT " ($";
     H$;")":END
20000 DATA ADC1097,AND1033,ASL3002,B
      CC8144,BCS8176,BEQ8240,BIT7036
      ,BMI8048
20010 DATA BNE8208,BPL8016,BRK0000,B
      VC8080,BVS8112,CLC0024,CLD0216
      ,CLI0088
20020 DATA CLV0184,CMP1193,CPX4224,C
      PY4192,DEC2198,DEX0202,DEY0136
      ,EOR1065
20030 DATA INC2230,INX0232,INY0200,J
      MP6076,JSR9032,LDA1161,LDX5162
      ,LDY5160
20040 DATA LSR3066,NOP0234,ORA1001,P
      HA0072,PHP0008,PLA0104,PLP0040
      ,ROL3034
20050 DATA ROR3098,RTI0064,RTS0096,S
      BC1225,SEC0056,SED0248,SEI0120
      ,STA1129
20060 DATA STX2134,STY2132,TAX0170,T
      AY0168,TSX0186,TXA0138,TXS0154
      ,TYA0152
```

Appendix D

Note: The ^ means "to the power of" as in 2 ^ 2=4.

Program D-1. Disassembler (VIC, PET, Apple, 64 Version).

```
1 HE$="Ø123456789ABCDEF"
2 L$="----------------------------------------
   ---"
4 J$="     --->"
13 PRINT"  DISASSEMBLER
14 PRINT
16 DIMM$(15,15)
17 FORI=ØTO15:FORB=ØTO14:READM$(I,B):NEXTB:
   NEXTI
25 REM START MAIN LOOP
3Ø PRINT"STARTING ADDRESS (DECIMAL)";:INPUT
   SA:TA=SA
31 PRINT"START ADDRESS HEX  ";:DE=SA:ZX=3:G
   OSUB12ØØ:PRINTH$"   "
35 IFSA<ØTHENEND
41 I=SA
45 REM PRINT ADDRESS
46 PRINTI" ";
5Ø X=PEEK(I)
55 GOSUB5ØØØ
56 IFL%=15ORM$(H%,L%)="Ø"THENPRINT" ?       "
   X:CK=Ø:LN=LN+1:GOTO7Ø
58 PRINTM$(H%,L%);
6Ø GOSUB6ØØØ:IFEQTHENEQ=Ø
7Ø I=I+1
72 IFLN=2ØTHENLN=Ø:GOTO2ØØØ
8Ø GOTO45
6ØØ IFCK=12THEN6Ø3
6Ø1 B=PEEK(I+1):IFB>127THENB=((NOTB)AND255)+
    1:B=-B
6Ø2 BAD=I+2+B:PRINT"          "BAD:I=I+1:RETUR
    N
6Ø3 IFH%>8THEN8ØØ
6Ø4 IFH%=2THENJ=1:GOTO85Ø
6Ø5 IFH%=6THENPRINT:PRINTL$:EQ=1:RETURN
```

```
606 IFH%=6THENRETURN
607 PRINT
608 RETURN
610 IFCK=12THEN615
611 PRINT" ("PEEK(I+1)"),Y"
612 I=I+1:RETURN
615 PRINT" ("PEEK(I+1)",X)"
616 I=I+1:RETURN
630 IFCK=12THEN635
631 PRINT" "PEEK(I+1)",X"
632 I=I+1:RETURN
635 PRINT"  "PEEK(I+1)
636 I=I+1:RETURN
640 IFCK=12THEN645
641 PRINT" "PEEK(I+1)",X"
642 I=I+1:RETURN
645 PRINT"  "PEEK(I+1)
646 I=I+1:RETURN
660 IFCK=12THEN645
661 IFH%=9ORH%=11THENPRINT" "PEEK(I+1)",Y"
662 IFH%=7ORH%=15ORH%=5ORH%=3THEN640
663 IFH%=13THEN631
664 PRINT:GOTO642
680 PRINT: RETURN
690 IFCK=12THEN800
691 I$="Y":GOTO850
720 IFCK=12THEN725
722 I$="X":GOTO850
725 IFH%=6THENPRINT" (IND.  ";:I=I+1
726 IFH%=2THEN850
727 IFH%=4THENPRINTJ$;:GOTO850
728 IFH%=8ORH%=10ORH%=12ORH%=14THEN850
729 GOTO610
730 IFCK=12THEN850
731 I$="X":GOTO850
740 IFCK=12THEN850
741 IFH%=11THENI$="Y":GOTO850
742 I$="X":GOTO850
800 PRINT"   #"PEEK(I+1)
801 I=I+1:RETURN
850 N=PEEK(I+1)+PEEK(I+2)*256
860 IFI$=""THEN900
870 IFI$="X"THENPRINT"  "N",X"
880 IFI$="Y"THENPRINT"  "N",Y"
```

```
890 I$="":I=I+2:RETURN
900 PRINT"   "N:I=I+2
906 RETURN
1000 DATABRK,ORA,Ø,Ø,Ø,ORA,ASL,Ø,PHP,ORA,ASL,
    Ø,Ø,ORA,ASL,BPL,ORA,Ø,Ø,Ø,ORA,ASL
1010 DATAØ,CLC,ORA,Ø,Ø,Ø,ORA,ASL,JSR,AND,Ø,Ø,
    BIT,AND,ROL,Ø,PLP,AND,ROL,Ø,BIT
1020 DATAAND,ROL,BMI,AND,Ø,Ø,Ø,AND,ROL,Ø,SEC,
    AND,Ø,Ø,Ø,AND,ROL,RTI,EOR,Ø,Ø,Ø
1030 DATAEOR,LSR,Ø,PHA,EOR,LSR,Ø,JMP,EOR,LSR,
    BVC,EOR,Ø,Ø,Ø,EOR,LSR,Ø,CLI,EOR,Ø
1040 DATAØ,Ø,EOR,LSR,RTS,ADC,Ø,Ø,Ø,ADC,ROR,Ø,
    PLA,ADC
1045 DATAROR,Ø,JMP,ADC,ROR,BVS,ADC,Ø,Ø,Ø
1050 DATAADC,ROR,Ø,SEI,ADC,Ø,Ø,Ø,ADC,ROR,Ø,ST
    A
1055 DATAØ,Ø,STY,STA,STX,Ø,DEY,Ø,TXA,Ø,STY,ST
    A
1060 DATASTX,BCC,STA,Ø,Ø,STY,STA,STX,Ø,TYA,ST
    A,TXS,Ø,Ø,STA,Ø,LDY,LDA,LDX,Ø
1070 DATALDY,LDA,LDX,Ø,TAY,LDA,TAX,Ø,LDY,LDA,
    LDX,BCS,LDA,Ø,Ø,LDY,LDA,LDX,Ø
1080 DATACLV,LDA,TSX,Ø
1090 DATALDY,LDA,LDX,CPY,CMP,Ø,Ø,CPY,CMP,DEC,
    Ø,INY,CMP,DEX,Ø,CPY,CMP,DEC
1095 DATABNE,CMP,Ø,Ø,Ø,CMP,DEC,Ø,CLD,CMP,Ø,Ø,
    Ø,CMP,DEC,CPX,SBC,Ø,Ø,CPX,SBC,INC
1098 DATAØ,INX,SBC,NOP,Ø,CPX,SBC,INC,BEQ,SBC,
    Ø,Ø,Ø,SBC,INC,Ø,SED,SBC,Ø,Ø,Ø,SBC
1099 DATAINC
1200 REM MAKE DECIMAL INTO HEX
1201 H$="":FORM=ZXTOØSTEP-1:N%=DE/(16^M):DE=D
    E-N%*16^M:H$=H$+MID$(HE$,N%+1,1)
1202 NEXT:RETURN
2000 PRINT"TYPE C TO CONTINUE FROM" I
2001 GETK$:IFK$=""THEN2001
2002 IFK$="C"THENSA=I:TA=SA:GOTO35
2003 INPUTSA:TA=SA:GOTO35
5000 REM ANALYZE H & L OF OPCODE
5010 H%=X/16:L%=X-H%*16
5020 :RETURN
6000 REM FIND ADDRESS TYPE & GOSUB
6020 CK=H%/2:IFCK=INT(CK)THENCK=12
6025 L%=L%+1
```

```
6030 ONL%GOSUB600,610,800,6050,640,640,660,60
    50,680,690,680,6050,720,730,740
6040 CK=0
6045 LN=LN+1
6050 RETURN
```

Program D-2. Atari Disassembler.

```
100 REM DISASSEMBLER
105 GRAPHICS 0:POSITION 11,0:? " FAST
    DISASSEMBLER ":? :? "Loading opc
    odes..."
110 DIM OPCODE$(256*10),LN(255),NB(25
    5),T$(10),D$(5)
120 FOR I=0 TO 255
125 READ T$,NB
130 LN(I)=LEN(T$)
140 OPCODE$(I*10+1,I*10+LN(I))=T$
150 NB(I)=NB
160 NEXT I
170 GRAPHICS 0:POSITION 11,0:? " FAST
    DISASSEMBLER "
180 ? :?
190 TRAP 190:? "{UP}{DEL LINE}Startin
    g Address (Decimal)";:INPUT ADDR:
    TRAP 40000
200 IF ADDR<0 OR ADDR>65535 THEN 190
210 OP=PEEK(ADDR):NB=NB(OP)
220 T$=OPCODE$(OP*10+1,OP*10+LN(OP))
230 PRINT ADDR;:POKE 85,10:PRINT OP;:
    POKE 85,15
240 ON NB+2 GOTO 242,244,250,260,270
242 NB=2:T=PEEK(ADDR+1):IF T>128 THEN
     T=T-256
243 PRINT T;:POKE 85,20:PRINT T$;"   "
    ;ADDR+2+T:GOTO 300
244 ? "Unimplemented":NB=1:GOTO 300
246 PRINT T$;" ";ADDR+2+T:GOTO 300
250 POKE 85,20:PRINT T$:GOTO 300
260 PRINT PEEK(ADDR+1);:POKE 85,20:D$
    =STR$(PEEK(ADDR+1)):GOSUB 400:GOT
    O 300
```

```
270 PRINT PEEK(ADDR+1);:POKE 85,15:PR
    INT PEEK(ADDR+2);:POKE 85,20
280 D$=STR$((PEEK(ADDR+1)+256*PEEK(AD
    DR+2))):GOSUB 400
300 ADDR=ADDR+NB:IF ADDR<0 THEN ADDR=
    65536-T
310 IF ADDR>65535 THEN ADDR=T
320 IF PEEK(53279)=7 THEN 210
330 GOTO 190
400 ? T$(1,4+(LN(OP)>4));D$;T$(4+2*(L
    N(OP)>5)):RETURN
500 DATA BRK,1,ORA (X),2,?,0,?,0,?,0,
    ORA  ,2,ASL  ,2,?,0,PHP,1,ORA # ,
    2
510 DATA ASL A,1,?,0,?,0,ORA  ,3,ASL
     ,3,?,0,BPL,-1,ORA ()Y,2,?,0,?,0
520 DATA ?,0,ORA  X,2,ASL  X,2,?,0,CL
    C,1,ORA  Y,3,?,0,?,0,?,0,ORA  X,3
530 DATA ASL ,2,?,0,JSR  ,3,AND (X),2
    ,?,0,?,0,BIT ,2,AND ,2,ROL ,2,?,0
540 DATA PLP,1,AND # ,2,ROL A,1,?,0,B
    IT ,3,AND ,3,ROL ,3,?,0,BMI,-1,AN
    D ()Y,2
550 DATA ?,0,?,0,?,0,AND  X,2,ROL  X,
    2,?,0,SEC,1,AND  Y,3,CLI,1,?,0
560 DATA ?,0,AND  X,3,ROL  X,3,?,0,RT
    I,1,EOR (X),2,?,0,?,0,?,0,EOR  ,2
570 DATA LSR  ,2,?,0,PHA,1,EOR # ,2,L
    SR  ,3,?,0,JMP  ,3,EOR  ,3,LSR  ,
    3,?,0
580 DATA BVC,-1,EOR ()Y,2,?,0,?,0,?,0
    ,EOR  X,2,LSR  X,2,?,0,CLI,1,EOR
     Y,2
590 DATA ?,0,?,0,?,0,EOR  X,3,LSR  X,
    3,?,0,RTS,1,ADC (X),2,?,0,?,0
600 DATA ?,0,ADC  ,2,ROR  ,2,?,0,PLA,
    1,ADC # ,2,ROR A,1,?,0,JMP (),108
    ,ADC  ,3
610 DATA ROR  ,3,?,0,BVS,-1,ADC ()Y,2
    ,?,0,?,0,?,0,ADC  X,2,ROR  X,2,?,
    0
```

```
620 DATA SEI,1,ADC  Y,3,?,0,?,0,?,0,A
    DC  X,3,ROR  X,3,?,0,?,0,STA (X),
    2
630 DATA ?,0,?,0,STY  ,2,STA  ,2,STX
     ,2,?,0,DEY,1,?,0,TXA,1,?,0
640 DATA STY  ,3,STA  ,3,STX  ,3,?,0,
    BCC,-1,STA ()Y,2,?,0,?,0,STY  X,2
    ,STA  X,2
650 DATA STX  Y,2,?,0,TYA,1,STA  Y,3,
    TXS,1,?,0,?,0,STA  X,3,?,0,?,0
660 DATA LDY # ,2,LDA (X),2,LDX # ,2,
    ?,0,LDY  ,2,LDA  ,2,LDX  ,2,?,0,T
    AY,1,LDA # ,2
670 DATA TAX,1,?,0,LDY  ,3,LDA  ,3,LD
    X  ,3,?,0,BCS,-1,LDA ()Y,2,?,0,?,
    0
680 DATA LDY  X,2,LDA  X,2,LDX  Y,2,?
    ,0,CLV,1,LDA  Y,3,TSX,1,?,0,LDY
    X,3,LDA  X,3
690 DATA LDX  Y,3,?,0,CPY # ,2,CMP (X
    ),2,?,0,?,0,CPY  ,2,CMP  ,2,DEC
    ,2,?,0
700 DATA INY,1,CMP # ,2,DEX,1,?,0,CPY
      ,3,CMP  ,3,DEC  ,3,?,0,BNE,-1,C
    MP ()Y,2
710 DATA ?,0,?,0,?,0,CMP  X,2,DEC  X,
    2,?,0,CLD,1,CMP  Y,3,?,0,?,0
720 DATA ?,0,CMP  X,3,DEC  X,3,?,0,CP
    X # ,2,SBC (X),2,?,0,?,0,CPX  ,2,
    SBC  ,2
730 DATA INC  ,2,?,0,INX,1,SBC # ,2,N
    OP,1,?,0,CPX  ,3,SBC  ,3,INC  ,3,
    ?,0
740 DATA BEQ,-1,SBC (Y),2,?,0,?,0,?,0
    ,SBC  X,2,INC  X,2,?,0,SED,1,SBC
     Y,3
750 DATA ?,0,?,0,?,0,SBC  X,3,INC  X,
    3,?,0
```

Appendix E
Number Tables

This lookup table should make it convenient when you need to translate hex, binary, or decimal numbers. The first column lists the decimal numbers between 1 and 255. The second column is the hexadecimal equivalent. The third column is the decimal equivalent of a hex *most significant byte* or ''MSB.'' The fourth column is the binary.

If you need to find out the decimal equivalent of the hex number $FD15, look up $FD in the MSB column and you'll see that it's 64768. Then look up the $15 in the LSB column (it's 21 decimal) and add 21 + 64768 to get the answer: 64789.

Going the other way, from decimal to hex, you could translate 64780 into hex by looking in the MSB column for the closest number (it must be smaller, however). In this case, the closest smaller number is 64768 so jot down $FD as the hex MSB. Then subtract 64768 from 64780 to get the LSB: 12. Look up 12 in the decimal column (it is $0C hex) and put the $FD MSB together with the $0C LSB for your answer: $FD0C.

With a little practice, you can use this chart for fairly quick conversions between the number systems. Most of your translations will only involve going from hex to decimal or vice versa with the LSB of hex numbers, the first 255 numbers, which require no addition or subtraction. Just look them up in the table.

Table E-1.

Decimal	Hex (LSB)	Hex (MSB)	Binary
1	01	256	00000001
2	02	512	00000010
3	03	768	00000011
4	04	1024	00000100
5	05	1280	00000101
6	06	1536	00000110
7	07	1792	00000111
8	08	2048	00001000
9	09	2304	00001001
10	0A	2560	00001010

Decimal	Hex (LSB)	Hex (MSB)	Binary
11	0B	2816	00001011
12	0C	3072	00001100
13	0D	3328	00001101
14	0E	3584	00001110
15	0F	3840	00001111
16	10	4096	00010000
17	11	4352	00010001
18	12	4608	00010010
19	13	4864	00010011
20	14	5120	00010100
21	15	5376	00010101
22	16	5632	00010110
23	17	5888	00010111
24	18	6144	00011000
25	19	6400	00011001
26	1A	6656	00011010
27	1B	6912	00011011
28	1C	7168	00011100
29	1D	7424	00011101
30	1E	7680	00011110
31	1F	7936	00011111
32	20	8192	00100000
33	21	8448	00100001
34	22	8704	00100010
35	23	8960	00100011
36	24	9216	00100100
37	25	9472	00100101
38	26	9728	00100110
39	27	9984	00100111
40	28	10240	00101000
41	29	10496	00101001
42	2A	10752	00101010
43	2B	11008	00101011
44	2C	11264	00101100
45	2D	11520	00101101
46	2E	11776	00101110
47	2F	12032	00101111
48	30	12288	00110000
49	31	12544	00110001

Decimal	Hex (LSB)	Hex (MSB)	Binary
50	32	12800	00110010
51	33	13056	00110011
52	34	13312	00110100
53	35	13568	00110101
54	36	13824	00110110
55	37	14080	00110111
56	38	14336	00111000
57	39	14592	00111001
58	3A	14848	00111010
59	3B	15104	00111011
60	3C	15360	00111100
61	3D	15616	00111101
62	3E	15872	00111110
63	3F	16128	00111111
64	40	16384	01000000
65	41	16640	01000001
66	42	16896	01000010
67	43	17152	01000011
68	44	17408	01000100
69	45	17664	01000101
70	46	17920	01000110
71	47	18176	01000111
72	48	18432	01001000
73	49	18688	01001001
74	4A	18944	01001010
75	4B	19200	01001011
76	4C	19456	01001100
77	4D	19712	01001101
78	4E	19968	01001110
79	4F	20224	01001111
80	50	20480	01010000
81	51	20736	01010001
82	52	20992	01010010
83	53	21248	01010011
84	54	21504	01010100
85	55	21760	01010101
86	56	22016	01010110
87	57	22272	01010111
88	58	22528	01011000

Decimal	Hex (LSB)	Hex (MSB)	Binary
89	59	22784	01011001
90	5A	23040	01011010
91	5B	23296	01011011
92	5C	23552	01011100
93	5D	23808	01011101
94	5E	24064	01011110
95	5F	24320	01011111
96	60	24576	01100000
97	61	24832	01100001
98	62	25088	01100010
99	63	25344	01100011
100	64	25600	01100100
101	65	25856	01100101
102	66	26112	01100110
103	67	26368	01100111
104	68	26624	01101000
105	69	26880	01101001
106	6A	27136	01101010
107	6B	27392	01101011
108	6C	27648	01101100
109	6D	27904	01101101
110	6E	28160	01101110
111	6F	28416	01101111
112	70	28672	01110000
113	71	28928	01110001
114	72	29184	01110010
115	73	29440	01110011
116	74	29696	01110100
117	75	29952	01110101
118	76	30208	01110110
119	77	30464	01110111
120	78	30720	01111000
121	79	30976	01111001
122	7A	31232	01111010
123	7B	31488	01111011
124	7C	31744	01111100
125	7D	32000	01111101
126	7E	32256	01111110
127	7F	32512	01111111

Decimal	Hex (LSB)	Hex (MSB)	Binary
128	80	32768	10000000
129	81	33024	10000001
130	82	33280	10000010
131	83	33536	10000011
132	84	33792	10000100
133	85	34048	10000101
134	86	34304	10000110
135	87	34560	10000111
136	88	34816	10001000
137	89	35072	10001001
138	8A	35328	10001010
139	8B	35584	10001011
140	8C	35840	10001100
141	8D	36096	10001101
142	8E	36352	10001110
143	8F	36608	10001111
144	90	36864	10010000
145	91	37120	10010001
146	92	37376	10010010
147	93	37632	10010011
148	94	37888	10010100
149	95	38144	10010101
150	96	38400	10010110
151	97	38656	10010111
152	98	38912	10011000
153	99	39168	10011001
154	9A	39424	10011010
155	9B	39680	10011011
156	9C	39936	10011100
157	9D	40192	10011101
158	9E	40448	10011110
159	9F	40704	10011111
160	A0	40960	10100000
161	A1	41216	10100001
162	A2	41472	10100010
163	A3	41728	10100011
164	A4	41984	10100100
165	A5	42240	10100101
166	A6	42496	10100110

Decimal	Hex (LSB)	Hex (MSB)	Binary
167	A7	42752	10100111
168	A8	43008	10101000
169	A9	43264	10101001
170	AA	43520	10101010
171	AB	43776	10101011
172	AC	44032	10101100
173	AD	44288	10101101
174	AE	44544	10101110
175	AF	44800	10101111
176	B0	45056	10110000
177	B1	45312	10110001
178	B2	45568	10110010
179	B3	45824	10110011
180	B4	46080	10110100
181	B5	46336	10110101
182	B6	46592	10110110
183	B7	46848	10110111
184	B8	47104	10111000
185	B9	47360	10111001
186	BA	47616	10111010
187	BB	47872	10111011
188	BC	48128	10111100
189	BD	48384	10111101
190	BE	48640	10111110
191	BF	48896	10111111
192	C0	49152	11000000
193	C1	49408	11000001
194	C2	49664	11000010
195	C3	49920	11000011
196	C4	50176	11000100
197	C5	50432	11000101
198	C6	50688	11000110
199	C7	50944	11000111
200	C8	51200	11001000
201	C9	51456	11001001
202	CA	51712	11001010
203	CB	51968	11001011
204	CC	52224	11001100

Decimal	Hex (LSB)	Hex (MSB)	Binary
205	CD	52480	11001101
206	CE	52736	11001110
207	CF	52992	11001111
208	D0	53248	11010000
209	D1	53504	11010001
210	D2	53760	11010010
211	D3	54016	11010011
212	D4	54272	11010100
213	D5	54528	11010101
214	D6	54784	11010110
215	D7	55040	11010111
216	D8	55296	11011000
217	D9	55552	11011001
218	DA	55808	11011010
219	DB	56064	11011011
220	DC	56320	11011100
221	DD	56576	11011101
222	DE	56832	11011110
223	DF	57088	11011111
224	E0	57344	11100000
225	E1	57600	11100001
226	E2	57856	11100010
227	E3	58112	11100011
228	E4	58368	11100100
229	E5	58624	11100101
230	E6	58880	11100110
231	E7	59136	11100111
232	E8	59392	11101000
233	E9	59648	11101001
234	EA	59904	11101010
235	EB	60160	11101011
236	EC	60416	11101100
237	ED	60672	11101101
238	EE	60928	11101110
239	EF	61184	11101111
240	F0	61440	11110000
241	F1	61696	11110001
242	F2	61952	11110010

Decimal	Hex (LSB)	Hex (MSB)	Binary
243	F3	62208	11110011
244	F4	62464	11110100
245	F5	62720	11110101
246	F6	62976	11110110
247	F7	63232	11110111
248	F8	63488	11111000
249	F9	63744	11111001
250	FA	64000	11111010
251	FB	64256	11111011
252	FC	64512	11111100
253	FD	64768	11111101
254	FE	65024	11111110
255	FF	65280	11111111

The following program will print copies of this number table. You might need to make some adjustments to the printout conventions of your computer's BASIC and your printer itself. This program is for Microsoft BASIC and will not work on the Atari.

Program E-1. Microsoft Table Printer.

```
10 OPEN4,4:REM OPEN CHANNEL TO PRINTER
100 HE$="0123456789ABCDEF"
110 FORX=1TO255
120 B=2:C=1
122 IFX<10THENPRINT#4,"  ";:GOTO130
124 IFX<100THENPRINT#4," ";
130 PRINT#4,X;" ";:DE=X:GOSUB240
135 REM CREATE BINARY
140 IFXAND1THENK$(C)="1":GOTO160
150 K$(C)="0"
160 C=C+1:IFBANDXTHENK$(C)="1":GOTO180
170 K$(C)="0"
180 B=B*2:IFC>8THEN200
190 GOTO160
200 FORI=8TO1STEP-1:PRINT#4,K$(I);:NEXTI
```

```
220 PRINT#4:NEXTX
230 END:REM TRANSFORM TO HEX
240 H$="":FORM=1TO0STEP-1:N%=DE/(16^M):DE=DE
    -N%*16^M
250 H$=H$+MID$(HE$,N%+1,1):NEXT
260 PRINT#4,H$" ";:DE=X*256
262 IFDE<1000THENPRINT#4,"  ";:GOTO270
264 IFDE<10000THENPRINT#4," ";
270 PRINT#4,DE" ";:RETURN
```

Appendix F
SUPERMON For PET

The following monitor extensions are the work of several programmers and were previously published in COMPUTE! Magazine. (See the copyright page for references.)

Here is the legendary Supermon — a version for Upgrade (3.0 or "New ROM") and 4.0 PETs, all keyboards, all memory sizes, 40 or 80 column screens. You need not yet know how to program in machine language (ML) to enter this program — or to use it. In fact, exploring with Supermon, you will find that the mysterious world of your computer's *own* language becomes gradually understandable. You will find yourself learning ML.

Many ML programmers with PET/CBM machines feel that Supermon is the essential tool for developing programs of short to medium length. All Upgrade and 4.0 machines have a "resident" monitor, a program within the computer's ROM which allows you to type SYS 1024 and see the registers, load and save and run ML programs, or see a memory dump (a list of numbers from the computer's memory cells). But to program or analyze ML easily, disassembler, assembler, hunt, and single-step functions are all practical necessities. Supermon provides these and more.

Even if you've never assembled a single instruction and don't know NOP from ROL, this appendix will lead you step-by-step through the entry and SAVE of Supermon.

How To Enter Supermon

1. Type in the BASIC program (Program 1). It is the same for all versions. Then save it normally by typing SAVE "CONTROL". This program will be used later to automatically find your memory size, transfer Supermon to the top, and report to you the SYS address you use to activate it.

2. Now the hard part: type SYS 1024 which enters you into the machine language monitor. You will see something like the following:

Figure 1.

```
B*
        PC   IRQ  SR AC XR YR SP
.;     0401 E455 32 04 5E 00 EE
```

Then type: M 0600 0648 and you will see something similar to this (the numbers will be different, but we are going to type over them which, after hitting RETURN on each line, will enter the new numbers into the computer's memory):

Figure 2.

```
.M Ø6ØØ Ø648
.:   Ø6ØØ 28 58 FF FF ØØ ØB Ø6 AD
.:   Ø6Ø8 FF FC ØØ 21 Ø6 Ø3 AD A9
.:   Ø61Ø CB 85 1F A9 ØC 85 2Ø A5
.:   Ø618 34 85 21 A5 35 85 22 AØ
.:   Ø62Ø ØØ 93 Ø6 Ø6 DØ 16 2Ø 38
.:   Ø628 Ø6 FØ 11 85 23 2Ø 38 Ø6
.:   Ø63Ø 18 65 34 AA A5 23 65 35
.:   Ø638 2Ø 43 Ø6 8A 2Ø 43 Ø6 2Ø
.:   Ø64Ø 5Ø Ø6 9Ø DB 6Ø EA EA A5
.:   Ø648 1F DØ Ø2 C6 2Ø C6 1F B1
.
```

We have divided Supermon into 21 blocks with 80 hexadecimal numbers per block to make typing easier. There is a final, shorter block with 64 numbers. Type right over the numbers on the screen so that line 0600 looks like it does in Program 2. Then hit RETURN and cursor over to the A5 on line 0608. (Set a TAB to this position if your keyboard has a TAB key.) Then type over the numbers in this line and so on. When you have finshed typing your RETURN on line 0648, type in: M 0650 0698 and the next block will appear for you to type over. Continue this way until you finish entering the new version of line 0CC8 at the end. (Hope that no lightning or fuses blow.)

3. If you have Upgrade ROMs, you will need to correct the lines listed in Program 3 at this point. To change line 06D0, simply type M 06D0 06D0 and it will appear so that you can type over it and RETURN as in step 2.

4. Now Supermon is in your memory and you must SAVE it. Hit RETURN so that you are on a new line and type: S''SUPERMON'', 01,0600,0CCC (to SAVE to tape) or type: S''0:SUPERMON'',08,0600,0CCC (to SAVE to disk drive 0).

5. Finally, you will want to use the Checksum program to see if you made any errors during the marathon. You probably did, but to make it as painless as possible, the Checksum program will flash through your Supermon and let you know which blocks need to be corrected. So, type in Program 4 (or if you have Upgrade ROMs, use the first three lines from Program 5). SAVE Checksum just in case. Then LOAD ''SUPERMON'' (an ordinary LOAD as with a BASIC program will slide it in starting at address 1536, above the end of Checksum). Then RUN. Incorrect blocks will be announced. When you know where the errors are, type SYS 1024 and then M XXXX XXXX for the starting and ending addresses of the bad block. Check

the numbers against Program 2 (or Program 3) and in all corrections. If, despite everything, you cannot find an error within a block, make sure that the corresponding number within the DATA statement of the Checksum program is correct. Then SAVE the good version "SUPERMON1" as in step 4.

6. Your reward is near. LOAD "CONTROL" and then LOAD SUPERMON1. Then type RUN and hold your breath. If all goes well, you should see:

Figure 3.

```
 SUPERMON4!

        DISSASSEMBLER   BY WOZNIAK/BAUM
          SINGLE STEP
BY JIM RUSSO
MOST OTHER STUFF ,BY BILL SEILER

TIDIED & WRAPPED BY JIM BUTTERFIELD


LINK TO MONITOR -- SYS 31283

SAVE WITH MLM:
.S "SUPERMON",Ø1,7A33,8ØØØ
READY.
```

And you should be able to use all the commands listed in the Supermon Summary. If some, or all, of the commands fail to function, check the last, short block of code to see if there are any errors.

After Supermon is relocated to the top of your memory, use a ML SAVE to save it in its final form. Instructions are on screen after RUN.

SUPERMON SUMMARY

```
COMMODORE MONITOR INSTRUCTIONS:
G GO RUN
L LOAD FROM TAPE OR DISK
M MEMORY DISPLAY
R REGISTER DISPLAY
S SAVE TO TAPE OR DISK
X EXIT TO BASIC
```

```
SUPERMON ADDITIONAL INSTRUCTIONS:
A SIMPLE ASSEMBLER
D DISASSEMBLER
F FILL MEMORY
H HUNT MEMORY
I SINGLE INSTRUCTION
P PRINTING DISASSEMBLER
T TRANSFER MEMORY
SUPERMON WILL LOAD ITSELF INTO THE
TOP OF MEMORY .. WHEREVER THAT HAPPENS
TO BE ON YOUR MACHINE.
   YOU MAY THEN SAVE THE MACHINE CODE
FOR FASTER LOADING IN THE FUTURE.
BE SURE TO NOTE THE SYS COMMAND WHICH
LINKS SUPERMON TO THE COMMODORE
MONITOR.


 SIMPLE ASSEMBLER
.A 2000 LDA #$12
.A 2002 STA $8000,X
.A 2005 (RETURN)
.
     IN THE ABOVE EXAMPLE THE USER
STARTED ASSEMBLY AT 2000 HEX.  THE
FIRST INSTRUCTION WAS LOAD A REGISTER
WITH IMMEDIATE 12 HEX.  IN THE SECOND
LINE THE USER DID NOT NEED TO TYPE THE
A AND ADDRESS.  THE SIMPLE ASSEMBLER
PROMPTS WITH THE NEXT ADDRESS.  TO EXIT
THE ASSEMBLER TYPE A RETURN AFTER THE
THE ADDRESS PROMPT.  SYNTAX IS THE SAME
AS THE DISASSEMBLER OUTPUT.

  DISASSEMBLER
.D 2000
(SCREEN CLEARS)
.,   2000 A9 12         LDA #$12
.,   2002 9D 00 80      STA $8000,X
.,   2005 AA            TAX
.,   2006 AA            TAX
(FULL PAGE OF INSTRUCTIONS)
     DISASSEMBLES 22 INSTRUCTIONS
STARTING AT 2000 HEX.  THE THREE BYTES
FOLLOWING THE ADDRESS MAY BE MODIFIED.
```

```
USE THE CRSR KEYS TO MOVE TO AND MODIFY
THE BYTES.  HIT RETURN AND THE BYTES
IN MEMORY WILL BE CHANGED.  SUPERMON
WILL THEN DISASSEMBLE THAT PAGE AGAIN.

 PRINTING DISASSEMBLER
.P 2000,2040
2000 A9 12         LDA #$12
2002 9D 00 80      STA $8000,XY.
2005 AA            TAX
   ....
203F A2 00         LDX #$00
TO ENGAGE PRINTER, SET UP BEFOREHAND:
     OPEN 4,4:CMD4
ON 4.0, ACCESS THE MONITOR VIA A CALL
SYS 54386 (*NOT* A BREAK) COMMAND

 SINGLE STEP
.I
     ALLOWS A MACHINE LANGUAGE PROGRAM
TO BE RUN STEP BY STEP.
CALL REGISTER DISPLAY WITH .R AND SET
THE PC ADDRESS TO THE DESIRED FIRST
INSTRUCTION FOR SINGLE STEPPING.
THE .I WILL CAUSE A SINGLE STEP TO
EXECUTE AND WILL DISASSEMBLE THE NEXT.
CONTROLS:
  < FOR SINGLE STEP;
  RVS FOR SLOW STEP;
  SPACE FOR FAST STEPPING;
  STOP TO RETURN TO MONITOR.
[ON BUSINESS KEYBOARDS--
      USE 8,←,6 AND STOP].
 FILL MEMORY
.F 1000 1100 FF
     FILLS THE MEMORY FROM 1000 HEX TO
1100 HEX WITH THE BYTE FF HEX.

 GO RUN
.G
     GO TO THE ADDRESS IN THE PC
REGISTER DISPLAY AND BEGIN RUN CODE.
ALL THE REGISTERS WILL BE REPLACED
WITH THE DISPLAYED VALUES.
.G 1000
```

```
     GO TO ADDRESS 1000 HEX AND BEGIN
RUNNING CODE.

 HUNT MEMORY
.H C000 D000 'READ
     HUNT THRU MEMORY FROM C000 HEX TO
D000 HEX FOR THE ASCII STRING READ AND
PRINT THE ADDRESS WHERE IT IS FOUND.  A
MAXIMUM OF 32 CHARACTERS MAY BE USED.
.H C000 D000 20 D2 FF
     HUNT MEMORY FROM C000 HEX TO D000
HEX FOR THE SEQUENCE OF BYTES 20 D2 FF
AND PRINT THE ADDRESS.  A MAXIMUM OF 32
BYTES MAY BE USED.

 LOAD
.L
   LOAD ANY PROGRAM FROM CASSETTE #1.
.L "RAM TEST"
   LOAD FROM CASSETTE #1 THE PROGRAM
NAMED RAM TEST.
.L "RAM TEST",08
 LOAD FROM DISK (DEVICE 8) THE PROGRAM
NAMED RAM TEST.
THIS COMMAND LEAVES BASIC POINTERS
UNCHANGED.

  MEMORY DISPLAY
 .M 0000 0080
 .:  0000 00 01 02 03 04 05 06 07
 .:  0008 08 09 0A 0B 0C 0D 0E 0F

      DISPLAY MEMORY FROM 0000 HEX TO
0080 HEX.  THE BYTES FOLLOWING THE .:
CAN BE ALTERED BY TYPING OVER THEM
THEN TYPING A RETURN.

 REGISTER DISPLAY
.R
     PC  IRQ  SR AC XR YR SP
.;  0000 E62E 01 02 03 04 05
     DISPLAYS THE REGISTER VALUES SAVED
WHEN SUPERMON WAS ENTERED.  THE VALUES
MAY BE CHANGED WITH THE EDIT FOLLOWED
BY A RETURN.
```

```
  USE THIS INSTRUCTION TO SET UP THE
PC VALUE BEFORE SINGLE STEPPING WITH
.I

 SAVE
.S "PROGRAM NAME",Ø1,Ø8ØØ,ØC8Ø
     SAVE TO CASSETTE #1 MEMORY FROM
Ø8ØØ HEX UP TO BUT NOT INCLUDING ØC8Ø
HEX AND NAME IT PROGRAM NAME.
.S "Ø:PROGRAM NAME",Ø8,12ØØ,1F5Ø
    SAVE TO DISK DRIVE #Ø MEMORY FROM
12ØØ HEX UP TO BUT NOT INCLUDING 1F5Ø
HEX AND NAME IT PROGRAM NAME.

 TRANSFER MEMORY
.T 1ØØØ 11ØØ 5ØØØ
     TRANSFER MEMORY IN THE RANGE 1ØØØ
HEX TO 11ØØ HEX AND START STORING IT AT
ADDRESS 5ØØØ HEX.

 EXIT TO BASIC
.X
     RETURN TO BASIC READY MODE.
THE STACK VALUE SAVED WHEN ENTERED WILL
BE RESTORED.  CARE SHOULD BE TAKEN THAT
THIS VALUE IS THE SAME AS WHEN THE
MONITOR WAS ENTERED.  A CLR IN
BASIC WILL FIX ANY STACK PROBLEMS.
```

Program I. CONTROL.

```
100 PRINT"{CLEAR}{02 DOWN}{REV} SUP
    ERMON!!"
110 PRINT"{DOWN}      DISSASSEMBLER ~
    {REV}D{OFF} BY WOZNIAK/BAU
    M
120 PRINT"        SINGLE STEP {REV}I
    {OFF} BY JIM RUSSO
130 PRINT"MOST OTHER STUFF {REV},HA
    LT{OFF} BY BILL SEILER
150 PRINT"{DOWN}TIDIED & WRAPPED BY
     JIM BUTTERFIELD"
```

```
170 L=PEEK(52)+PEEK(53)*256:SYS1536
    :M=PEEK(33):N=PEEK(34)
180 POKE52,M:POKE53,N:POKE48,M:POKE
    49,N:N=M+N*256
210 PRINT"{02 DOWN}LINK TO MONITOR ~
    -- SYS";N
220 PRINT:PRINT"SAVE WITH MLM:"
230 PRINT".S ";CHR$(34);"SUPERMON";
    CHR$(34);",01";:X=N/4096:G
    OSUB250
240 X=L/4096:GOSUB250:END
250 PRINT",";:FORJ=1TO4:X%=X:X=(X-X
    %)*16:IFX%>9THENX%=X%+7
260 PRINTCHR$(X%+48);:NEXTJ:RETURN
```

Program 2. SUPERMON 4.0

```
.
.:   0600 A9 CB 85 1F A9 0C 85 20
.:   0608 A5 34 85 21 A5 35 85 22
.:   0610 A0 00 20 38 06 D0 16 20
.:   0618 38 06 F0 11 85 23 20 38
.:   0620 06 18 65 34 AA A5 23 65
.:   0628 35 20 43 06 8A 20 43 06
.:   0630 20 50 06 90 DB 60 EA EA
.:   0638 A5 1F D0 02 C6 20 C6 1F
.:   0640 B1 1F 60 48 A5 21 D0 02
.:   0648 C6 22 C6 21 68 91 21 60
.
.:   0650 A9 80 C5 1F A9 06 E5 20
.:   0658 60 AA AA AA AA AA AA AA
.:   0660 AA AA AA AA AA AA AA AA
.:   0668 AA AA AA AA AA AA AA AA
.:   0670 AA AA AA AA AA AA AA AA
.:   0678 AA AA AA AA AA AA AA AA
.:   0680 AD FE FF 00 85 34 AD FF
.:   0688 FF 00 85 35 AD FC FF 00
.:   0690 8D FA 03 AD FD FF 00 8D
.:   0698 FB 03 00 00 A2 08 DD DE
.
.:   06A0 FF 00 D0 0E 86 B4 8A 0A
```

```
.:   Ø6A8  AA  BD  E9  FF  ØØ  48  BD  E8
.:   Ø6BØ  FF  ØØ  48  6Ø  CA  1Ø  EA  4C
.:   Ø6B8  9A  FA  ØØ  A2  Ø2  2C  A2  ØØ
.:   Ø6CØ  ØØ  B4  FB  DØ  Ø8  B4  FC  DØ
.:   Ø6C8  Ø2  E6  DE  D6  FC  D6  FB  6Ø
.:   Ø6DØ  2Ø  98  D7  C9  2Ø  FØ  F9  6Ø
.:   Ø6D8  A9  ØØ  ØØ  8D  ØØ  ØØ  Ø1  2Ø
.:   Ø6EØ  79  FA  ØØ  2Ø  6B  D7  2Ø  57
.:   Ø6E8  D7  9Ø  Ø9  6Ø  2Ø  98  D7  2Ø
.
.:   Ø6FØ  54  D7  BØ  DE  AE  Ø6  Ø2  9A
.:   Ø6F8  4C  A4  D7  2Ø  31  D5  CA  DØ
.:   Ø7ØØ  FA  6Ø  E6  FD  DØ  Ø2  E6  FE
.:   Ø7Ø8  6Ø  A2  Ø2  B5  FA  48  BD  ØA
.:   Ø71Ø  Ø2  95  FA  68  9D  ØA  Ø2  CA
.:   Ø718  DØ  F1  6Ø  AD  ØB  Ø2  AC  ØC
.:   Ø72Ø  Ø2  4C  CE  FA  ØØ  A5  FD  A4
.:   Ø728  FE  38  E5  FB  8D  1B  Ø2  98
.:   Ø73Ø  E5  FC  A8  ØD  1B  Ø2  6Ø  2Ø
.:   Ø738  81  FA  ØØ  2Ø  44  D7  2Ø  92
.
.:   Ø74Ø  FA  ØØ  2Ø  AF  FA  ØØ  2Ø  92
.:   Ø748  FA  ØØ  2Ø  CA  FA  ØØ  2Ø  44
.:   Ø75Ø  D7  9Ø  15  A6  DE  DØ  65  2Ø
.:   Ø758  C1  FA  ØØ  9Ø  6Ø  A1  FB  81
.:   Ø76Ø  FD  2Ø  A8  FA  ØØ  2Ø  39  D5
.:   Ø768  DØ  EB  2Ø  C1  FA  ØØ  18  AD
.:   Ø77Ø  1B  Ø2  65  FD  85  FD  98  65
.:   Ø778  FE  85  FE  2Ø  AF  FA  ØØ  A6
.:   Ø78Ø  DE  DØ  3D  A1  FB  81  FD  2Ø
.:   Ø788  C1  FA  ØØ  BØ  34  2Ø  65  FA
.
.:   Ø79Ø  ØØ  2Ø  68  FA  ØØ  4C  1B  FB
.:   Ø798  ØØ  2Ø  31  FA  ØØ  2Ø  44  D7
.:   Ø7AØ  2Ø  92  FA  ØØ  2Ø  44  D7  2Ø
.:   Ø7A8  98  D7  2Ø  63  D7  9Ø  14  85
.:   Ø7BØ  B5  A6  DE  DØ  11  2Ø  CA  FA
.:   Ø7B8  ØØ  9Ø  ØC  A5  B5  81  FB  2Ø
.:   Ø7CØ  39  D5  DØ  EE  4C  9A  FA  ØØ
.:   Ø7C8  4C  BA  D4  2Ø  81  FA  ØØ  2Ø
.:   Ø7DØ  44  D7  2Ø  92  FA  ØØ  2Ø  44
.:   Ø7D8  D7  2Ø  98  D7  A2  ØØ  ØØ  2Ø
```

```
.
.:  Ø7EØ 98 D7 C9 27 DØ 14 2Ø 98
.:  Ø7E8 D7 9D 1Ø Ø2 E8 2Ø CF FF
.:  Ø7FØ C9 ØD FØ 22 EØ 2Ø DØ F1
.:  Ø7F8 FØ 1C 8E ØØ ØØ Ø1 2Ø 6B
.:  Ø8ØØ D7 9Ø C6 9D 1Ø Ø2 E8 2Ø
.:  Ø8Ø8 CF FF C9 ØD FØ Ø9 2Ø 63
.:  Ø81Ø D7 9Ø B6 EØ 2Ø DØ EC 86
.:  Ø818 B4 2Ø 34 D5 A2 ØØ ØØ AØ
.:  Ø82Ø ØØ ØØ B1 FB DD 1Ø Ø2 DØ
.:  Ø828 ØC C8 E8 E4 B4 DØ F3 2Ø
.
.:  Ø83Ø 17 D7 2Ø 31 D5 2Ø 39 D5
.:  Ø838 A6 DE DØ 92 2Ø CA FA ØØ
.:  Ø84Ø BØ DD 4C BA D4 2Ø 81 FA
.:  Ø848 ØØ 8D ØD Ø2 A5 FC 8D ØE
.:  Ø85Ø Ø2 A9 Ø4 A2 ØØ ØØ 8D Ø9
.:  Ø858 Ø2 8E ØA Ø2 A9 93 2Ø D2
.:  Ø86Ø FF A9 16 85 B5 2Ø Ø6 FC
.:  Ø868 ØØ 2Ø 64 FC ØØ 85 FB 84
.:  Ø87Ø FC C6 B5 DØ F2 A9 91 2Ø
.:  Ø878 D2 FF 4C BA D4 AØ 2C 2Ø
.
.:  Ø88Ø 79 D5 2Ø 17 D7 2Ø 31 D5
.:  Ø888 A2 ØØ ØØ A1 FB 2Ø 74 FC
.:  Ø89Ø ØØ 48 2Ø BB FC ØØ 68 2Ø
.:  Ø898 D3 FC ØØ A2 Ø6 EØ Ø3 DØ
.:  Ø8AØ 13 AC 1C Ø2 FØ ØE A5 FF
.:  Ø8A8 C9 E8 B1 FB BØ 1C 2Ø 5C
.:  Ø8BØ FC ØØ 88 DØ F2 Ø6 FF 9Ø
.:  Ø8B8 ØE BD 51 FF ØØ 2Ø 45 FD
.:  Ø8CØ ØØ BD 57 FF ØØ FØ Ø3 2Ø
.:  Ø8C8 45 FD ØØ CA DØ D4 6Ø 2Ø
.
.:  Ø8DØ 68 FC ØØ AA E8 DØ Ø1 C8
.:  Ø8D8 98 2Ø 5C FC ØØ 8A 86 B4
.:  Ø8EØ 2Ø 22 D7 A6 B4 6Ø AD 1C
.:  Ø8E8 Ø2 38 A4 FC AA 1Ø Ø1 88
.:  Ø8FØ 65 FB 9Ø Ø1 C8 6Ø A8 4A
.:  Ø8F8 9Ø ØB 4A BØ 17 C9 22 FØ
.:  Ø9ØØ 13 29 Ø7 Ø9 8Ø 4A AA BD
.:  Ø9Ø8 ØØ FF ØØ BØ Ø4 4A 4A 4A
```

```
.:  0910 4A 29 0F D0 04 A0 80 A9
.:  0918 00 00 AA BD 44 FF 00 85
.
.:  0920 FF 29 03 8D 1C 02 98 29
.:  0928 8F AA 98 A0 03 E0 8A F0
.:  0930 0B 4A 90 08 4A 4A 09 20
.:  0938 88 D0 FA C8 88 D0 F2 60
.:  0940 B1 FB 20 5C FC 00 A2 01
.:  0948 20 A1 FA 00 CC 1C 02 C8
.:  0950 90 F0 A2 03 CC 09 02 90
.:  0958 F0 60 A8 B9 5E FF 00 8D
.:  0960 0B 02 B9 9E FF 00 8D 0C
.:  0968 02 A9 00 00 A0 05 0E 0C
.
.:  0970 02 2E 0B 02 2A 88 D0 F6
.:  0978 69 3F 20 D2 FF CA D0 EA
.:  0980 4C 31 D5 20 81 FA 00 20
.:  0988 44 D7 20 92 FA 00 20 44
.:  0990 D7 A9 04 A2 00 00 8D 09
.:  0998 02 8E 0A 02 20 34 D5 20
.:  09A0 0B FC 00 20 64 FC 00 85
.:  09A8 FB 84 FC 20 35 F3 F0 05
.:  09B0 20 CA FA 00 B0 E9 4C BA
.:  09B8 D4 20 81 FA 00 A9 03 85
.
.:  09C0 B5 20 98 D7 20 0B D5 D0
.:  09C8 F8 AD 0D 02 85 FB AD 0E
.:  09D0 02 85 FC 4C E7 FB 00 CD
.:  09D8 0A 02 F0 03 20 D2 FF 60
.:  09E0 A9 03 A2 24 8D 09 02 8E
.:  09E8 0A 02 20 34 D5 78 AD FA
.:  09F0 FF 00 85 90 AD FB FF 00
.:  09F8 85 91 A9 A0 8D 4E E8 CE
.:  0A00 13 E8 A9 2E 8D 48 E8 A9
.:  0A08 00 00 8D 49 E8 AE 06 02
.
.:  0A10 9A 4C 55 D6 20 C0 FC 68
.:  0A18 8D 05 02 68 8D 04 02 68
.:  0A20 8D 03 02 68 8D 02 02 68
.:  0A28 8D 01 02 68 8D 00 00 02
.:  0A30 BA 8E 06 02 58 20 34 D5
```

```
.:  ØA38 2Ø 23 D5 85 B5 AØ ØØ ØØ
.:  ØA4Ø 2Ø FE D4 2Ø 31 D5 AD ØØ
.:  ØA48 ØØ Ø2 85 FC AD Ø1 Ø2 85
.:  ØA5Ø FB 2Ø 17 D7 2Ø ØE FC ØØ
.:  ØA58 2Ø 35 F3 C9 F7 FØ F9 2Ø
.
.:  ØA6Ø 35 F3 DØ Ø3 4C BA D4 C9
.:  ØA68 FF FØ F4 4C 5B FD ØØ 2Ø
.:  ØA7Ø 81 FA ØØ 2Ø 44 D7 8E 11
.:  ØA78 Ø2 A2 Ø3 2Ø 79 FA ØØ 48
.:  ØA8Ø CA DØ F9 A2 Ø3 68 38 E9
.:  ØA88 3F AØ Ø5 4A 6E 11 Ø2 6E
.:  ØA9Ø 1Ø Ø2 88 DØ F6 CA DØ ED
.:  ØA98 A2 Ø2 2Ø CF FF C9 ØD FØ
.:  ØAAØ 1E C9 2Ø FØ F5 2Ø F7 FE
.:  ØAA8 ØØ BØ ØF 2Ø 78 D7 A4 FB
.
.:  ØABØ 84 FC 85 FB A9 3Ø 9D 1Ø
.:  ØAB8 Ø2 E8 9D 1Ø Ø2 E8 DØ DB
.:  ØACØ 8E ØB Ø2 A2 ØØ ØØ 86 DE
.:  ØAC8 FØ Ø4 E6 DE FØ 7B A2 ØØ
.:  ØADØ ØØ 86 B5 A5 DE 2Ø 74 FC
.:  ØAD8 ØØ A6 FF 8E ØC Ø2 AA BC
.:  ØAEØ 5E FF ØØ BD 9E FF ØØ 2Ø
.:  ØAE8 EØ FE ØØ DØ E2 A2 Ø6 EØ
.:  ØAFØ Ø3 DØ 1A AC 1C Ø2 FØ 15
.:  ØAF8 A5 FF C9 E8 A9 3Ø BØ 21
.
.:  ØBØØ 2Ø E6 FE ØØ DØ CA 2Ø E8
.:  ØBØ8 FE ØØ DØ C5 88 DØ EB Ø6
.:  ØB1Ø FF 9Ø ØB BC 57 FF ØØ BD
.:  ØB18 51 FF ØØ 2Ø EØ FE ØØ DØ
.:  ØB2Ø B3 CA DØ DØ FØ ØA 2Ø DF
.:  ØB28 FE ØØ DØ A9 2Ø DF FE ØØ
.:  ØB3Ø DØ A4 AD ØB Ø2 C5 B5 DØ
.:  ØB38 9D 2Ø 44 D7 AC 1C Ø2 FØ
.:  ØB4Ø 2F AD ØC Ø2 C9 9D DØ 2Ø
.:  ØB48 2Ø CA FA ØØ 9Ø ØB 98 DØ
.
.:  ØB5Ø Ø5 AE 1B Ø2 1Ø ØB 4C 9A
.:  ØB58 FA ØØ C8 DØ FA AE 1B Ø2
```

```
.:   ØB6Ø  1Ø  F5  CA  CA  8A  AC  1C  Ø2
.:   ØB68  DØ  Ø3  B9  FC  ØØ  ØØ  91  FB
.:   ØB7Ø  88  DØ  F8  A5  DE  91  FB  2Ø
.:   ØB78  64  FC  ØØ  85  FB  84  FC  AØ
.:   ØB8Ø  41  2Ø  79  D5  2Ø  17  D7  2Ø
.:   ØB88  31  D5  4C  D8  FD  ØØ  A8  2Ø
.:   ØB9Ø  E6  FE  ØØ  DØ  11  98  FØ  ØE
.:   ØB98  86  B4  A6  B5  DD  1Ø  Ø2  Ø8
.
.:   ØBAØ  E8  86  B5  A6  B4  28  6Ø  C9
.:   ØBA8  3Ø  9Ø  Ø3  C9  47  6Ø  38  6Ø
.:   ØBBØ  4Ø  Ø2  45  Ø3  DØ  Ø8  4Ø  Ø9
.:   ØBB8  3Ø  22  45  33  DØ  Ø8  4Ø  Ø9
.:   ØBCØ  4Ø  Ø2  45  33  DØ  Ø8  4Ø  Ø9
.:   ØBC8  4Ø  Ø2  45  B3  DØ  Ø8  4Ø  Ø9
.:   ØBDØ  ØØ  ØØ  22  44  33  DØ  8C  44
.:   ØBD8  ØØ  ØØ  11  22  44  33  DØ  8C
.:   ØBEØ  44  9A  1Ø  22  44  33  DØ  Ø8
.:   ØBE8  4Ø  Ø9  1Ø  22  44  33  DØ  Ø8
.
.:   ØBFØ  4Ø  Ø9  62  13  78  A9  ØØ  ØØ
.:   ØBF8  21  81  82  ØØ  ØØ  ØØ  ØØ  59
.:   ØCØØ  4D  91  92  86  4A  85  9D  2C
.:   ØCØ8  29  2C  23  28  24  59  ØØ  ØØ
.:   ØC1Ø  58  24  24  ØØ  ØØ  1C  8A  1C
.:   ØC18  23  5D  8B  1B  A1  9D  8A  1D
.:   ØC2Ø  23  9D  8B  1D  A1  ØØ  ØØ  29
.:   ØC28  19  AE  69  A8  19  23  24  53
.:   ØC3Ø  1B  23  24  53  19  A1  ØØ  ØØ
.:   ØC38  1A  5B  5B  A5  69  24  24  AE
.
.:   ØC4Ø  AE  A8  AD  29  ØØ  ØØ  7C  ØØ
.:   ØC48  ØØ  15  9C  6D  9C  A5  69  29
.:   ØC5Ø  53  84  13  34  11  A5  69  23
.:   ØC58  AØ  D8  62  5A  48  26  62  94
.:   ØC6Ø  88  54  44  C8  54  68  44  E8
.:   ØC68  94  ØØ  ØØ  B4  Ø8  84  74  B4
.:   ØC7Ø  28  6E  74  F4  CC  4A  72  F2
.:   ØC78  A4  8A  ØØ  ØØ  AA  A2  A2  74
.:   ØC8Ø  74  74  72  44  68  B2  32  B2
.:   ØC88  ØØ  ØØ  22  ØØ  ØØ  1A  1A  26
```

```
.
.:   ØC9Ø 26 72 72 88 C8 C4 CA 26
.:   ØC98 48 44 44 A2 C8 54 46 48
.:   ØCAØ 44 5Ø 2C 41 49 4E ØØ ØØ
.:   ØCA8 DB FA ØØ 3Ø FB ØØ 5E FB
.:   ØCBØ ØØ D1 FB ØØ F8 FC ØØ 28
.:   ØCB8 FD ØØ D4 FD ØØ 4D FD ØØ
.:   ØCCØ B9 D4 7F FD ØØ 4A FA ØØ
.:   ØCC8 33 FA ØØ AA AA AA AA AA
```

Program 3. Changes For SUPERMON 3.0.

```
.:   Ø6DØ 2Ø EB E7 C9 2Ø FØ F9 6Ø
.
.:   Ø6EØ 79 FA ØØ 2Ø BE E7 2Ø AA
.:   Ø6E8 E7 9Ø Ø9 6Ø 2Ø EB E7 2Ø
.:   Ø6FØ A7 E7 BØ DE AE Ø6 Ø2 9A
.:   Ø6F8 4C F7 E7 2Ø CD FD CA DØ
.
.:   Ø738 81 FA ØØ 2Ø 97 E7 2Ø 92
.
.:   Ø748 FA ØØ 2Ø CA FA ØØ 2Ø 97
.:   Ø75Ø E7 9Ø 15 A6 DE DØ 65 2Ø
.
.:   Ø76Ø FD 2Ø A8 FA ØØ 2Ø D5 FD
.
.:   Ø798 ØØ 2Ø 81 FA ØØ 2Ø 97 E7
.:   Ø7AØ 2Ø 92 FA ØØ 2Ø 97 E7 2Ø
.:   Ø7A8 EB E7 2Ø B6 E7 9Ø 14 85
.
.:   Ø7CØ D5 FD DØ EE 4C 9A FA ØØ
.:   Ø7C8 4C 56 FD 2Ø 81 FA ØØ 2Ø
.:   Ø7DØ 97 E7 2Ø 92 FA ØØ 2Ø 97
.:   Ø7D8 E7 2Ø EB E7 A2 ØØ ØØ 2Ø
.:   Ø7EØ EB E7 C9 27 DØ 14 2Ø EB
.:   Ø7E8 E7 9D 1Ø Ø2 E8 2Ø CF FF
.
.:   Ø7F8 FØ 1C 8E ØØ ØØ Ø1 2Ø BE
.:   Ø8ØØ E7 9Ø C6 9D 1Ø Ø2 E8 2Ø
.:   Ø8Ø8 CF FF C9 ØD FØ Ø9 2Ø B6
.:   Ø81Ø E7 9Ø B6 EØ 2Ø DØ EC 86
.:   Ø818 B4 2Ø DØ FD A2 ØØ ØØ AØ
.
.:   Ø83Ø 6A E7 2Ø CD FD 2Ø D5 FD
.
.:   Ø84Ø BØ DD 4C 56 FD 2Ø 81 FA
```

```
.
.:  Ø878 D2 FF 4C 56 FD AØ 2C 2Ø
.:  Ø88Ø 15 FE 2Ø 6A E7 2Ø CD FD
.:  Ø8EØ 2Ø 75 E7 A6 B4 6Ø AD 1C
.
.:  Ø98Ø 4C CD FD 2Ø 81 FA ØØ 2Ø
.:  Ø988 97 E7 2Ø 92 FA ØØ 2Ø 97
.:  Ø99Ø E7 A9 Ø4 A2 ØØ ØØ 8D Ø9
.:  Ø998 Ø2 8E ØA Ø2 2Ø DØ FD 2Ø
.
.:  Ø9A8 FB 84 FC 2Ø Ø1 F3 FØ Ø5
.:  Ø9BØ 2Ø CA FA ØØ BØ E9 4C 56
.:  Ø9B8 FD 2Ø 81 FA ØØ A9 Ø3 85
.:  Ø9CØ B5 2Ø EB E7 2Ø A7 FD DØ
.
.:  Ø9E8 ØA Ø2 2Ø DØ FD 78 AD FA
.
.:  ØA1Ø 9A 4C F1 FE 2Ø 7B FC 68
.
.:  ØA3Ø BA 8E Ø6 Ø2 58 2Ø DØ FD
.:  ØA38 2Ø BF FD 85 B5 AØ ØØ ØØ
.:  ØA4Ø 2Ø 9A FD 2Ø CD FD AD ØØ
.
.:  ØA5Ø FB 2Ø 6A E7 2Ø ØE FC ØØ
.:  ØA58 2Ø Ø1 F3 C9 F7 FØ F9 2Ø
.:  ØA6Ø Ø1 F3 DØ Ø3 4C 56 FD C9
.
.:  ØA7Ø 81 FA ØØ 2Ø 97 E7 8E 11
.
.:  ØAA8 ØØ BØ ØF 2Ø CB E7 A4 FB
.
.:  ØB38 9D 2Ø 97 E7 AC 1C Ø2 FØ
.
.:  ØB8Ø 41 2Ø 15 FE 2Ø 6A E7 2Ø
.:  ØB88 CD FD 4C D8 FD ØØ A8 2Ø
.
.:  ØCCØ 55 FD 7F FD ØØ 4A FA ØØ
```

Program 4. SUPERMON 4.0 Checksum.

```
1ØØ REM SUPERMON 4 CHECKSUM
11Ø DATA7331,12186,1ØØ71,1Ø387,1Ø82
    9,9175,1Ø314,9823,9715,871
    4,8852
12Ø DATA885Ø,9748,7754,1Ø247,1Ø423,
    1Ø948,1ØØ75,6Ø93,5492,78Ø5
    :S=1536
```

```
130 FORB=1TO21:READX:FORI=STOS+79:N
    =PEEK(I):Y=Y+N
140 NEXTI:IFY<>XTHENPRINT"ERROR IN ~
    BLOCK #"B:GOTO160
150 PRINT"BLOCK #"B" IS CORRECT"
160 S=I:Y=0:NEXTB:PRINT"CHECK THE F
    INAL, SHORT BLOCK BY HAND"
```

Program 5. Changes For SUPERMON 3.0 Checksum.

```
100 REM SUPERMON 3 CHECKSUM
110 DATA7331,12186,10467,10880,1112
    4,10005,10906,10196,9951,8
    813
120 DATA8852,9329,10239,8457,10334,
    10423,11047,10311,6093,549
    2,7805:S=1536
```

PET MICROMON

An Enhanced Machine Language Monitor

Micromon is for Upgrade and 4.0 BASICs, all memory sizes, all keyboards and is in the public domain. If you have enough memory, you can add the additional commands of ''Micromon Plus'' as well. ''Plus'' is from $5B00 to $5F48 and you will want to move Micromon from $1000 up to $6000.

There is quite a bit of typing here so there are two checksum programs which will find and flag any errors. See the instructions for typing in Supermon.

Micromon Instructions

SIMPLE ASSEMBLER

```
.A 2000 LDA #$12
.A 2002 STA $8000,X
.A 2005 DEX:GARBAGE
```

In the above example, the user started assembly at 2000 hex. The first instruction was load a register with immediate 12 hex. In the second line the user did not need to type the A and address. The simple assembler retypes the last entered line and prompts with the next address. To exit the assembler, type a return after the address prompt. Syntax is the same as the Disassembler output. A colon (:) can be used to terminate a line.

BREAK SET

```
.B 1000 00FF
```

The example sets a break at 1000 hex on the FF hex occurrence of the instruction at 1000. Break set is used with the QUICK TRACE command. A BREAK SET with count blank stops at the first occurrence of the break address.

COMPARE MEMORY

```
.C 1000 2000 C000
```

Compares memory from hex 1000 to hex 2000 to memory beginning at hex C000. Compare will print the locations of the unequal bytes.

DISASSEMBLER

```
.D 2000 3000
., 2000 A9 12    LDA #$12
., 2002 9D 00 80 STA $8000,X
., 2005 AA       TAX
```

Disassembles from 2000 to 3000. The three bytes following the address may be modified. Use the CRSR KEYS to move to and modify the bytes. Hit return and the bytes in memory will be changed. Micromon will then disassemble that line again.

Disassembly can be done under the control of the cursor. To disassemble one at a time from $1000.

```
.D 1000
```

If the cursor is on the last line, one instruction can be disassembled for each pressing of the cursor down key. If it is held down, the key will repeat and continuous disassembly will occur. Disassembly can even be in reverse! If the screen is full of a disassembly listing, place the cursor at the top line of the screen and press the cursor up key.

EXIT MICROMON

```
.E
```

Combine the killing of Micromon and exit to BASIC.

FILL MEMORY

```
.F 1000 1100 FF
```

Fills the memory from 1000 hex to 1100 hex with the byte FF hex.

GO RUN

```
.G
```

Go to the address in the PC Register display and begin run code. All the registers will be replaced with the displayed values.

```
.G 1000
```

Go to address 1000 hex and begin running code.

HUNT MEMORY

```
.H C000 D000 'READ
```

Hunt through memory from C000 hex to D000 hex for the ASCII string "read" and print the address where it is found. Maximum of 32 characters may be used.

```
.H C000 D000 20 D2 FF
```

Hunt memory from C000 hex to D000 hex for the sequence of bytes 20 D2 FF and print the address. A maximum of 32 bytes may be used. Hunt can be stopped with the STOP key.

KILL MICROMON

```
.K
```

Restore the Break vector and IRQ that was saved before Micromon was called and break into the TIM monitor. A return to Micromon can be done with a Go to the value in the PC register.

LOAD

```
.L "RAM TEST",08
```

Load the program named RAM TEST from the disk. *Note for cassette users:* To load or save to cassette. Kill Micromon with the K command to return to the TIM monitor. Then use the TIM monitor L and S commands to load and save to the cassettes. This has to be done because of the repeat keys of Micromon. BASIC 4.0 users then can return to Micromon with a Go command to the PC value, but BASIC 2.0 users should return to BASIC, then SYS to Micromon because the TIM overwrites the IRQ value for loads and saves with a filename.

MEMORY DISPLAY

```
.M 0000 0008
.: 0000 30 31 32 33 34 35 36 37 1234567
.: 0008 38 41 42 43 44 45 46 47 89ABCDE
```

Display memory from 0000 hex to 0008 in hex and ASCII. The bytes following the address may be modified by editing and then typing a RETURN.

Memory display can also be done with the cursor control keys.

NEW LOCATER

```
.N 1000 17FF 6000 1000 1FFF
.N 1FB0 1FFF 6000 1000 1FFF W
```

The first line fixes all three byte instructions in the range 1000 hex to 1FFF hex by adding 6000 hex offset to the bytes following the instruction. New Locater will not adjust any instruction outside of the 1000 hex to 1FFF hex range. The second line adjusts Word values in the same range as the first line. New Locater stops and disassembles on any bad op code.

CALCULATE BRANCH OFFSET

```
.O 033A 033A FE
```

Calculate the offset for branch instructions. The first address is the starting address and the second address is the target address. The offset is then displayed.

QUICK TRACE

```
.Q
.Q 1000
```

The first example begins trace at the address in the PC of the register display. The second begins at 1000 hex. Each instruction is executed as in the WALK command, but no disassembly is shown. The Break Address is checked for the break on Nth occurrence. The execution may be stopped by pressing the STOP and = (left arrow on business) keys at the same time.

REGISTER DISPLAY

```
.R
   PC  IRQ  SR AC XR YR SP
.: 0000 E455 01 02 03 04 05
```

Displays the register values saved when Micromon was entered. The values may be changed with the edit followed by a RETURN.

SAVE

```
.S "1:PROGRAM NAME",08,0800,0C80
```

Save to disk drive #1 memory from 0800 hex up to, *but not including*, 0C80 hex and name it PROGRAM NAME. See note in LOAD command for cassette users.

TRANSFER MEMORY

```
.T 1000 1100 5000
```

Transfer memory in the range 1000 hex to 1100 hex and start storing it at address 5000 hex.

WALK CODE

```
.W
```

Single step starting at address in register PC.

```
.W 1000
```

Single step starting at address 1000 hex. Walk will cause a single step to execute and will disassemble the next instruction. Stop key stops walking. The J key finishes a subroutine that is walking, then continues with the walk.

EXIT TO BASIC

```
.X
```

Return to BASIC READY mode. The stack value saved when entered will be restored. Care should be taken that this value is the same as when the monitor was entered. A CLR in BASIC will fix any stack problems. Do not X to BASIC then return to Micromon via a SYS to the cold start address. Return via a SYS to BRK (SYS 1024) or SYS to the Warm start of Micromon (Warm start=Cold start+3). An X and cold start will write over the TIM break vector that was saved.

CHANGE CHARACTER SETS

```
.Z
```

Change from uppercase/graphics to lower/uppercase mode or vice versa.

HEX CONVERSION

```
.$4142 16706 A B 0100 0001 0100 0010
```

A hex number is input and the decimal value, the ASCII for the two bytes, and the binary values are returned. The ASCII control values are returned in reverse.

Hex conversion can also be scrolled with the cursor control keys.

DECIMAL CONVERSION

```
.#16706 4142 A B 0100 0001 0100 0010
```

A decimal number is input and the hex value, the ASCII for the two bytes, and the binary values are returned.

BINARY CONVERSION

```
.%0100000101000010 4142 16706 A B
```

A binary number is input and the hex value, the decimal number, and the ASCII values are returned.

ASCII CONVERSION

```
."A 41 65 0100 0001
```

An ASCII character is input and the hex value, decimal value, and binary values are returned. Because of the quote, *the control characters can be determined also.*

ADDITION

```
.+ 1111 2222  3333
```

The two hex numbers input are added, and the sum displayed.

SUBTRACTION

```
.-3333 1111 2222
```

The second number is subtracted from the first number and the difference displayed.

CHECKSUM

```
.& A000 AFFF 67E2
```

The checksum between the two addresses is calculated and displayed.

MICROMON INSTRUCTIONS:

A	SIMPLE ASSEMBLE
B	BREAK SET
C	COMPARE MEMORY
D	DISASSEMBLER
E	EXIT MICROMON
F	FILL MEMORY
G	GO RUN
H	HUNT MEMORY
K	KILL MICROMON
L	LOAD
M	MEMORY DISPLAY
N	NEW LOCATER
O	CALCULATE BRANCH
Q	QUICK TRACE
R	REGISTER DISPLAY
S	SAVE
T	TRANSFER MEMORY
W	WALK CODE
X	EXIT TO BASIC
Z	CHANGE CHARACTER SETS
$	HEX CONVERSION
#	DECIMAL CONVERSION

- % BINARY CONVERSION
- " ASCII CONVERSION
- + ADDITION
- – SUBTRACTION
- & CHECKSUM

Micromon also has repeat for all keys.

Micromon is executed by the following: SYS 4096 as listed in Program 2, where it resides in $1000 to $1FFF.

For 8032, make the following changes for Micromon operation. In location the X stands for the start of Micromon. Values in hex.

Location	Old Value	New Value
X3E7	08	10 To display 16 instead
X3EC	08	10 of 8 bytes.
X3F6	08	10
X427	08	10
XD18	08	10
XDA3	08	10
XCFC	28	50 To fix scroll.
XD7B	28	50
XE16	83	87
XE20	28	50
XE24	C0	80
XE26	04	08
XE37	27	4F
XE46	28	50
X681	24	00 To print all characters in Walk command.

Micromon Plus Instructions

PRINTING DISASSEMBLER

.(Shift) D 1000 1FFF

The same as the Disassembler but no ., printed before each line. Also the ASCII values for the bytes are output at the end of the line.

FORM FEED SET

.I

Sets a form feed for printout. Gives 57 printed lines per page. Works with the Shift D and Shift M commands.

.I "Heading"

Sets form feed with a message to be printed at the top of each page.

.I X

Cancels form feed.

PRINT LOAD ADDRESS

.J "File name"

Read the load address of the file and print it in hex. Device number 8 is used.

KILL MICROMON ADDITIONS

.(Shift) K

Kill Micromon and its additions and BRK to the TIM monitor. This is the same as the unshifted K command except now a G command will reinitialize Micromon and the additions.

LOAD FROM DISK

.(Shift) L "filename"

This is the same as the normal load command except that the disk (device #8) is used as the default, not the cassette.

PRINTING MEMORY DUMP

.(Shift) M F000 F100

The same as the normal Memory dump, but does not print the .: and prints out 16 hex bytes and the ASCII for them.

PRINT SWITCHER

.P

If the output is to the CRT then switch the output to the printer (device #4). If the output is not to the CRT then clear the output device and restore the output to the CRT.

.P 06

Make device #6 the output device if the current output is to the CRT.

SEND TO PROM PROGRAMMER

.U 06 7000 7FFF

This command will send out bytes to a PROM programmer on the IEEE bus. The first byte is the device number and the two addresses are the range of memory to output. A CHR$(2) is sent first to start the programmer. This is followed by the memory bytes as ASCII characters separated by spaces. After all bytes have been sent, a CHR$(3) is sent to stop the programmer. Micromon then does a checksum on the range to compare against the programmer checksum. Although this is for a particular programmer, it could be modified for others.

SPECIFY LOAD ADDRESS

.Y 7000 "Filename"

This command allows a file to be loaded starting at the address you specify and not the load address it would normally load into. The disk (device #8) is used for loading.

TEXT FLIP FOR 8032 & FAT 40's

.(Shift) Z

This is for 8032 and Fat 40's to go from Text to Graphics mode or vice versa.

DOS SUPPORT

.@ or .>

This reads the error channel from disk device number 8.

.@ disk command or .> disk command

This sends the disk command to disk device number 8.

.@$0 or .>$0

This reads the directory from disk device number 8. The SPACE BAR will hold the display, any other key will start it again, and the STOP key will return to command mode.

CONTROL CHARACTERS

.(Up arrow)g

This command will print the control character of the ASCII character input.

Examples of controls:

g	Ring bell
i	Tab set and clear
M	Insert line
n	Text mode
N	Graphics mode
q	Cursor down
Q	Cursor up
s	Home cursor
S	Clear screen
u	Delete line
v	Erase end
V	Erase begin

MICROMON PLUS INSTRUCTIONS

(Shift) D	PRINTING DISASSEMBLER
I	HEADING AND FORM FEED CONTROL
J	PRINT LOAD ADDRESS
(Shift) K	KILL MICROMON ADDITIONS
(Shift) L	LOAD FROM DISK
(Shift) M	PRINT MEMORY DISPLAY
P	PRINTER SWITCHING
U	SEND TO PROM PROGRAMMER
Y	SPECIFY LOAD ADDRESS
(Shift) Z	TEXT/GRAPHICS FLIP
>	DOS SUPPORT COMMANDS
@	DOS SUPPORT COMMANDS
(Up arrow)	CONTROL CHARACTERS

Program 1. Checksum For Micromon.

```
10 DATA 15463,14894,1429Ø,11897,12
    453,13919,14116,11715,1257
    5,14571
2Ø DATA 13693,11853,129Ø3,14513,12
    137,15ØØ6,12654,13291,1243
    6,13899
3Ø DATA 15366,9999,11834,13512,128
    92,14475,15149,14896,15782
    ,9511
4Ø DATA 12171,8985
1ØØ Q=4Ø96
11Ø FOR BLOCK=1TO32
12Ø FOR BYTE=ØTO127
13Ø X=PEEK(Q+BYTE):CK=CK+X
14Ø NEXT BYTE
15Ø READ SUM
16Ø IF SUM <> CK THEN PRINT" ERROR
    IN BLOCK #"BLOCK:GOTO17Ø
165 PRINT"                   BLOCK"
    BLOCK" IS CORRECT
17Ø CK=Ø:Q=Q+128
18Ø NEXT BLOCK
```

Program 2. Micromon.

```
1ØØØ 4C ØC 1Ø 4C 6F 1Ø 4C CF
1ØØ8 FF 4C D2 FF 78 A5 92 A6
1Ø1Ø 93 8D E5 Ø2 8E E6 Ø2 AD
1Ø18 F6 1F AE F7 1F 8D E3 Ø2
1Ø2Ø 8E E4 Ø2 AD FØ 1F AE F1
1Ø28 1F 85 92 86 93 A5 9Ø A6
1Ø3Ø 91 CD EE 1F DØ Ø5 EC EF
1Ø38 1F FØ 1Ø 8D 9E Ø2 8E 9F
1Ø4Ø Ø2 AD EE 1F AE EF 1F 85
1Ø48 9Ø 86 91 AD EC 1F AE ED
1Ø5Ø 1F EØ 8Ø BØ Ø8 85 34 86
1Ø58 35 85 3Ø 86 31 A9 1Ø 8D
1Ø6Ø 84 Ø2 8D 85 Ø2 A9 ØØ 8D
```

```
1Ø68 86 Ø2 8D A2 Ø2 58 ØØ 38
1Ø7Ø AD 7B Ø2 E9 Ø1 8D 7B Ø2
1Ø78 AD 7A Ø2 E9 ØØ 8D 7A Ø2

1Ø8Ø 2Ø 55 19 A2 42 A9 2A 2Ø
1Ø88 29 18 A9 52 DØ 23 A9 3F
1Ø9Ø 2Ø Ø9 1Ø 2Ø 55 19 A9 2E
1Ø98 2Ø Ø9 1Ø A9 ØØ 8D 94 Ø2
1ØAØ 8D A2 Ø2 A2 FF 9A 2Ø A4
1ØA8 18 C9 2E FØ F9 C9 2Ø FØ
1ØBØ F5 A2 1D DD 92 1F DØ 13
1ØB8 8D 87 Ø2 8A ØA AA BD BØ
1ØCØ 1F 85 FB BD B1 1F 85 FC
1ØC8 6C FB ØØ CA 1Ø E5 6C E3
1ØDØ Ø2 A2 Ø2 DØ Ø2 A2 ØØ B4
1ØD8 FB DØ Ø9 B4 FC DØ Ø3 EE
1ØEØ 94 Ø2 D6 FC D6 FB 6Ø A9
1ØE8 ØØ 8D 8C Ø2 2Ø 4F 12 A2
1ØFØ Ø9 2Ø 52 19 CA DØ FA 6Ø
1ØF8 A2 Ø2 B5 FA 48 BD 91 Ø2

11ØØ 95 FA 68 9D 91 Ø2 CA DØ
11Ø8 F1 6Ø AD 92 Ø2 AC 93 Ø2
111Ø 4C 17 11 A5 FD A4 FE 38
1118 E5 FB 8D 91 Ø2 98 E5 FC
112Ø A8 ØD 91 Ø2 6Ø A9 ØØ FØ
1128 Ø2 A9 Ø1 8D 95 Ø2 2Ø E6
113Ø 17 2Ø 55 19 2Ø 13 11 2Ø
1138 3C 18 9Ø 1B 2Ø ØA 11 BØ
114Ø Ø3 4C C5 11 2Ø 7F 11 E6
1148 FD DØ Ø2 E6 FE 2Ø 3B 19
115Ø AC 94 Ø2 DØ 45 FØ E5 2Ø
1158 ØA 11 18 AD 91 Ø2 65 FD
116Ø 85 FD 98 65 FE 85 FE 2Ø
1168 F8 1Ø 2Ø 7F 11 2Ø ØA 11
117Ø BØ 53 2Ø D1 1Ø 2Ø D5 1Ø
1178 AC 94 Ø2 DØ 1D FØ EB A2
```

```
118Ø  ØØ  A1  FB  AC  95  Ø2  FØ  Ø2
1188  81  FD  C1  FD  FØ  ØB  2Ø  13
119Ø  18  2Ø  52  19  2Ø  AE  18  FØ
1198  Ø1  6Ø  4C  93  1Ø  2Ø  Ø1  18
11AØ  2Ø  ØB  18  2Ø  A4  18  2Ø  6F
11A8  18  9Ø  17  8D  89  Ø2  AE  94
11BØ  Ø2  DØ  12  2Ø  13  11  9Ø  ØD
11B8  AD  89  Ø2  81  FB  2Ø  3B  19
11CØ  DØ  EC  4C  8E  1Ø  4C  93  1Ø
11C8  2Ø  Ø1  18  2Ø  ØB  18  2Ø  A4
11DØ  18  A2  ØØ  2Ø  A4  18  C9  27
11D8  DØ  14  2Ø  A4  18  9D  A3  Ø2
11EØ  E8  2Ø  Ø6  1Ø  C9  ØD  FØ  22
11E8  EØ  2Ø  DØ  F1  FØ  1C  8E  97
11FØ  Ø2  2Ø  77  18  9Ø  CC  9D  A3
11F8  Ø2  E8  2Ø  Ø6  1Ø  C9  ØD  FØ

12ØØ  Ø9  2Ø  6F  18  9Ø  BC  EØ  2Ø
12Ø8  DØ  EC  8E  88  Ø2  2Ø  55  19
121Ø  A2  ØØ  AØ  ØØ  B1  FB  DD  A3
1218  Ø2  DØ  ØA  C8  E8  EC  88  Ø2
122Ø  DØ  F2  2Ø  8E  11  2Ø  3B  19
1228  AC  94  Ø2  DØ  Ø5  2Ø  13  11
123Ø  BØ  DE  4C  93  1Ø  2Ø  39  14
1238  2Ø  13  11  9Ø  ØD  AØ  2C  2Ø
124Ø  E7  1Ø  2Ø  AB  12  2Ø  AE  18
1248  DØ  EE  2Ø  B3  15  DØ  E3  2Ø
125Ø  47  19  2Ø  13  18  2Ø  52  19
1258  2Ø  ØE  1E  48  2Ø  ØB  13  68
126Ø  2Ø  22  13  A2  Ø6  EØ  Ø3  DØ
1268  14  AC  8B  Ø2  FØ  ØF  AD  96
127Ø  Ø2  C9  E8  B1  FB  BØ  1D  2Ø
1278  A1  12  88  DØ  F1  ØE  96  Ø2

128Ø  9Ø  ØE  BD  E9  1E  2Ø  AD  15
1288  BD  EF  1E  FØ  Ø3  2Ø  AD  15
129Ø  CA  DØ  D2  6Ø  2Ø  B7  12  AA
1298  E8  DØ  Ø1  C8  98  2Ø  A1  12
12AØ  8A  8E  88  Ø2  2Ø  1A  18  AE
12A8  88  Ø2  6Ø  AD  8B  Ø2  2Ø  B6
12BØ  12  85  FB  84  FC  6Ø  38  A4
```

```
12B8 FC AA 10 01 88 65 FB 90
12C0 01 C8 60 A8 4A 90 0B 4A
12C8 B0 17 C9 22 F0 13 29 07
12D0 09 80 4A AA BD 98 1E B0
12D8 04 4A 4A 4A 4A 29 0F D0
12E0 04 A0 80 A9 00 AA BD DC
12E8 1E 8D 96 02 29 03 8D 8B
12F0 02 98 29 8F AA 98 A0 03
12F8 E0 8A F0 0B 4A 90 08 4A

1300 4A 09 20 88 D0 FA C8 88
1308 D0 F2 60 B1 FB 20 A1 12
1310 A2 01 20 F1 10 CC 8B 02
1318 C8 90 F0 A2 03 C0 03 90
1320 F1 60 A8 B9 F6 1E 8D 92
1328 02 B9 36 1F 8D 93 02 A9
1330 00 A0 05 0E 93 02 2E 92
1338 02 2A 88 D0 F6 69 3F 20
1340 09 10 CA D0 EA 4C 52 19
1348 20 01 18 A9 03 20 AC 13
1350 A0 2C 4C 50 15 BD 05 01
1358 CD F8 1F D0 0B BD 06 01
1360 CD F9 1F D0 03 20 D7 18
1368 A5 97 CD 83 02 F0 0A 8D
1370 83 02 A9 10 8D 84 02 D0
1378 24 C9 FF F0 20 AD 84 02

1380 F0 05 CE 84 02 D0 16 CE
1388 85 02 D0 11 A9 02 8D 85
1390 02 A5 9E D0 08 A9 00 85
1398 97 A9 02 85 A8 AD F3 1F
13A0 48 AD F2 1F 48 08 48 48
13A8 48 6C 9E 02 8D 89 02 48
13B0 20 A4 18 20 19 19 D0 F8
13B8 68 49 FF 4C AE 12 20 39
13C0 14 AE 94 02 D0 0D 20 13
13C8 11 90 08 20 D6 13 20 AE
13D0 18 D0 EE 4C 4A 12 20 55
13D8 19 A2 2E A9 3A 20 29 18
13E0 20 52 19 20 13 18 A9 08
```

```
13E8 2Ø Ø3 19 A9 Ø8 2Ø B9 13
13FØ A9 12 2Ø Ø9 1Ø AØ Ø8 A2
13F8 ØØ A1 FB 29 7F C9 2Ø BØ

14ØØ Ø2 A9 2E 2Ø Ø9 1Ø C9 22
14Ø8 FØ Ø4 C9 62 DØ ØA A9 14
141Ø 2Ø Ø9 1Ø A9 22 2Ø Ø9 1Ø
1418 2Ø 3B 19 88 DØ DB A9 92
142Ø 4C Ø9 1Ø 2Ø Ø1 18 A9 Ø8
1428 2Ø AC 13 2Ø B3 15 2Ø D6
143Ø 13 A9 3A 8D 6F Ø2 4C 5C
1438 15 2Ø Ø1 18 85 FD 86 FE
144Ø 2Ø Ø6 1Ø C9 ØD FØ Ø3 2Ø
1448 Ø6 18 4C 55 19 2Ø 4C 18
145Ø 85 FD 86 FE A2 ØØ 8E A4
1458 Ø2 2Ø A4 18 C9 2Ø FØ F4
146Ø 9D 8D Ø2 E8 EØ Ø3 DØ F1
1468 CA 3Ø 14 BD 8D Ø2 38 E9
147Ø 3F AØ Ø5 4A 6E A4 Ø2 6E
1478 A3 Ø2 88 DØ F6 FØ E9 A2

148Ø Ø2 2Ø Ø6 1Ø C9 ØD FØ 22
1488 C9 3A FØ 1E C9 2Ø FØ F1
149Ø 2Ø A4 15 BØ ØF 2Ø 84 18
1498 A4 FB 84 FC 85 FB A9 3Ø
14AØ 9D A3 Ø2 E8 9D A3 Ø2 E8
14A8 DØ D7 8E 92 Ø2 A2 ØØ 8E
14BØ 94 Ø2 A2 ØØ 8E 89 Ø2 AD
14B8 94 Ø2 2Ø C3 12 AE 96 Ø2
14CØ 8E 93 Ø2 AA BD 36 1F 2Ø
14C8 84 15 BD F6 1E 2Ø 84 15
14DØ A2 Ø6 EØ Ø3 DØ 14 AC 8B
14D8 Ø2 FØ ØF AD 96 Ø2 C9 E8
14EØ A9 3Ø BØ 1E 2Ø 81 15 88
14E8 DØ F1 ØE 96 Ø2 9Ø ØE BD
14FØ E9 1E 2Ø 84 15 BD EF 1E
14F8 FØ Ø3 2Ø 84 15 CA DØ D2

15ØØ FØ Ø6 2Ø 81 15 2Ø 81 15
15Ø8 AD 92 Ø2 CD 89 Ø2 FØ Ø3
```

```
1510 4C 91 15 20 3C 18 AC 8B
1518 02 F0 2E AD 93 02 C9 9D
1520 D0 1F 20 13 11 90 0A 98
1528 D0 6F AE 91 02 30 6A 10
1530 08 C8 D0 65 AE 91 02 10
1538 60 CA CA 8A AC 8B 02 D0
1540 03 B9 FC 00 91 FB 88 D0
1548 F8 AD 94 02 91 FB A0 41
1550 8C 6F 02 20 B3 15 20 E7
1558 10 20 AB 12 A9 20 8D 70
1560 02 8D 75 02 A5 FC 20 B8
1568 15 8E 71 02 8D 72 02 A5
1570 FB 20 B8 15 8E 73 02 8D
1578 74 02 A9 07 85 9E 4C 93

1580 10 20 84 15 8E 88 02 AE
1588 89 02 DD A3 02 F0 0D 68
1590 68 EE 94 02 F0 03 4C B2
1598 14 4C 8E 10 E8 8E 89 02
15A0 AE 88 02 60 C9 30 90 03
15A8 C9 47 60 38 60 CD 8C 02
15B0 D0 03 60 A9 91 4C 09 10
15B8 48 4A 4A 4A 4A 20 32 18
15C0 AA 68 29 0F 4C 32 18 8D
15C8 7D 02 08 68 29 EF 8D 7C
15D0 02 8E 7E 02 8C 7F 02 68
15D8 18 69 01 8D 7B 02 68 69
15E0 00 8D 7A 02 A9 80 8D 86
15E8 02 D0 21 AD 13 E8 10 03
15F0 4C 55 13 D8 68 8D 7F 02
15F8 68 8D 7E 02 68 8D 7D 02

1600 68 8D 7C 02 68 8D 7B 02
1608 68 8D 7A 02 A5 90 8D 82
1610 02 A5 91 8D 81 02 BA 8E
1618 80 02 20 D7 18 AD 12 E8
1620 58 AD 7C 02 29 10 F0 03
1628 4C 6F 10 2C 86 02 50 1F
1630 AD 7A 02 CD 99 02 D0 6D
1638 AD 7B 02 CD 98 02 D0 65
```

```
164Ø AD 9C Ø2 DØ 5D AD 9D Ø2
1648 DØ 55 A9 8Ø 8D 86 Ø2 3Ø
165Ø 14 4E 86 Ø2 9Ø D2 AE 8Ø
1658 Ø2 9A AD F5 1F 48 AD F4
166Ø 1F 48 4C 1F 17 2Ø 55 19
1668 2Ø 3Ø 19 8D 89 Ø2 AØ ØØ
167Ø 2Ø ØB 19 AD 7B Ø2 AE 7A
1678 Ø2 85 FB 86 FC 2Ø 52 19

168Ø A9 24 8D 8C Ø2 2Ø 52 12
1688 2Ø E4 FF FØ FB C9 Ø3 DØ
169Ø Ø3 4C 93 1Ø C9 4A DØ 56
1698 A9 Ø1 8D 86 Ø2 DØ 4F CE
16AØ 9D Ø2 CE 9C Ø2 AD 12 E8
16A8 C9 EE FØ Ø4 C9 6F DØ 3E
16BØ A2 53 4C 85 1Ø A9 ØØ FØ
16B8 12 AD 9A Ø2 AE 9B Ø2 8D
16CØ 9C Ø2 8E 9D Ø2 A9 4Ø DØ
16C8 Ø2 A9 8Ø 8D 86 Ø2 2Ø Ø6
16DØ 1Ø C9 ØD FØ 11 C9 2Ø DØ
16D8 5C 2Ø 6Ø 18 2Ø FC 18 2Ø
16EØ Ø6 1Ø C9 ØD DØ 4F 2Ø 55
16E8 19 AD 86 Ø2 FØ 22 78 A9
16FØ AØ 8D 4E E8 CE 13 E8 2C
16F8 12 E8 AD FØ 1F AE F1 1F

17ØØ 8D 82 Ø2 8E 81 Ø2 A9 3B
17Ø8 A2 ØØ 8D 48 E8 8E 49 E8
171Ø AE 8Ø Ø2 9A 78 AD 81 Ø2
1718 85 91 AD 82 Ø2 85 9Ø AD
172Ø 7A Ø2 48 AD 7B Ø2 48 AD
1728 7C Ø2 48 AD 7D Ø2 AE 7E
173Ø Ø2 AC 7F Ø2 4Ø 4C 8E 1Ø
1738 2Ø 4C 18 8D 98 Ø2 8E 99
174Ø Ø2 A9 ØØ 8D 9A Ø2 8D 9B
1748 Ø2 2Ø 5D 18 8D 9A Ø2 8E
175Ø 9B Ø2 4C 93 1Ø 2Ø E6 17
1758 8D AØ Ø2 8E A1 Ø2 2Ø 5D
176Ø 18 8D 8D Ø2 8E 8E Ø2 2Ø
1768 5D 18 8D 8F Ø2 8E 9Ø Ø2
```

```
1770 20 Ø6 1Ø C9 ØD FØ ØA 20
1778 Ø6 1Ø C9 57 DØ Ø3 EE 8C

1780 Ø2 2Ø 3C 18 AE 94 Ø2 DØ
1788 18 2Ø ØA 11 9Ø 13 AC 8C
1790 Ø2 DØ 1A B1 FB 2Ø C3 12
1798 AA BD F6 1E DØ Ø6 2Ø E7
17AØ 1Ø 4C 93 1Ø AC 8B Ø2 CØ
17A8 Ø2 DØ 33 FØ Ø3 8C 8B Ø2
17BØ 88 38 B1 FB AA ED 8D Ø2
17B8 C8 B1 FB ED 8E Ø2 9Ø 1E
17CØ 88 AD 8F Ø2 F1 FB C8 AD
17C8 9Ø Ø2 F1 FB 9Ø 1Ø 88 18
17DØ 8A 6D AØ Ø2 91 FB C8 B1
17D8 FB 6D A1 Ø2 91 FB 2Ø 3B
17EØ 19 88 1Ø FA 3Ø 9E 2Ø 4C
17E8 18 85 FD 86 FE 2Ø 5D 18
17FØ 8D 92 Ø2 8E 93 Ø2 2Ø A4
17F8 18 2Ø 6Ø 18 85 FB 86 FC

18ØØ 6Ø 2Ø 4C 18 BØ F6 2Ø 6Ø
18Ø8 18 BØ Ø3 2Ø 5D 18 85 FD
181Ø 86 FE 6Ø A5 FC 2Ø 1A 18
1818 A5 FB 48 4A 4A 4A 4A 2Ø
182Ø 32 18 AA 68 29 ØF 2Ø 32
1828 18 48 8A 2Ø Ø9 1Ø 68 4C
183Ø Ø9 1Ø 18 69 F6 9Ø Ø2 69
1838 Ø6 69 3A 6Ø A2 Ø2 B5 FA
184Ø 48 B5 FC 95 FA 68 95 FC
1848 CA DØ F3 6Ø A9 ØØ 8D 97
185Ø Ø2 2Ø A4 18 C9 2Ø FØ F9
1858 2Ø 84 18 BØ Ø8 2Ø A4 18
186Ø 2Ø 6F 18 9Ø Ø7 AA 2Ø 6F
1868 18 9Ø Ø1 6Ø 4C 8E 1Ø A9
187Ø ØØ 8D 97 Ø2 2Ø A4 18 C9
1878 2Ø DØ Ø9 2Ø A4 18 C9 2Ø

188Ø DØ ØF 18 6Ø 2Ø 99 18 ØA
1888 ØA ØA ØA 8D 97 Ø2 2Ø A4
```

```
189Ø 18 2Ø 99 18 ØD 97 Ø2 38
1898 6Ø C9 3A Ø8 29 ØF 28 9Ø
18AØ Ø2 69 Ø8 6Ø 2Ø Ø6 1Ø C9
18A8 ØD DØ F8 4C 93 1Ø A5 9B
18BØ C9 EF DØ Ø7 Ø8 2Ø CC FF
18B8 85 9E 28 6Ø 2Ø C6 18 AD
18CØ 13 E8 6A 9Ø F7 6Ø 2Ø AE
18C8 18 DØ ØB 2Ø D7 18 A9 Ø3
18DØ 85 BØ A9 ØØ 85 AF 6Ø Ø8
18D8 78 AD 4Ø E8 Ø9 1Ø 8D 4Ø
18EØ E8 A9 7F 8D 4E E8 A9 3C
18E8 8D 11 E8 A9 3D 8D 13 E8
18FØ AD EE 1F 85 9Ø AD EF 1F
18F8 85 91 28 6Ø 8D 7B Ø2 8E

19ØØ 7A Ø2 6Ø 8D 89 Ø2 AØ ØØ
19Ø8 2Ø 52 19 B1 FB 2Ø 1A 18
191Ø 2Ø 3B 19 CE 89 Ø2 DØ FØ
1918 6Ø 2Ø 6F 18 9Ø ØB A2 ØØ
192Ø 81 FB C1 FB FØ Ø3 4C 8E
1928 1Ø 2Ø 3B 19 CE 89 Ø2 6Ø
193Ø A9 7C 85 FB A9 Ø2 85 FC
1938 A9 Ø5 6Ø E6 FB DØ Ø7 E6
194Ø FC DØ Ø3 EE 94 Ø2 6Ø 98
1948 48 2Ø 55 19 68 A2 2E 2Ø
195Ø 29 18 A9 2Ø 2C A9 ØD 4C
1958 Ø9 1Ø A2 ØØ BD 76 1F 2Ø
196Ø Ø9 1Ø E8 EØ 1C DØ F5 AØ
1968 3B 2Ø 47 19 AD 7A Ø2 2Ø
197Ø 1A 18 AD 7B Ø2 2Ø 1A 18
1978 2Ø 52 19 AD 81 Ø2 2Ø 1A

198Ø 18 AD 82 Ø2 2Ø 1A 18 2Ø
1988 3Ø 19 2Ø Ø3 19 4C 93 1Ø
199Ø 4C 8E 1Ø 2Ø 4C 18 2Ø FC
1998 18 2Ø 5D 18 8D 82 Ø2 8E
19AØ 81 Ø2 2Ø 3Ø 19 8D 89 Ø2
19A8 2Ø A4 18 2Ø 19 19 DØ F8
19BØ FØ DB 2Ø 6Ø 1C AE 8Ø Ø2
19B8 9A 6C 94 ØØ 4C 8E 1Ø AØ
```

```
19CØ  Ø1  84  D4  88  84  D1  84  96
19C8  84  9D  A9  Ø2  85  DB  A9  A3
19DØ  85  DA  2Ø  Ø6  1Ø  C9  2Ø  FØ
19D8  F9  C9  ØD  FØ  1A  C9  22  DØ
19EØ  DB  2Ø  Ø6  1Ø  C9  22  FØ  36
19E8  C9  ØD  FØ  ØB  91  DA  E6  D1
19FØ  C8  CØ  1Ø  FØ  C7  DØ  EA  AD
19F8  87  Ø2  C9  4C  DØ  E1  AD  ØØ

1AØØ  CØ  C9  4Ø  DØ  Ø6  2Ø  22  F3
1AØ8  4C  12  1A  C9  4C  DØ  AD  2Ø
1A1Ø  56  F3  2Ø  BC  18  A5  96  29
1A18  1Ø  DØ  E1  4C  93  1Ø  2Ø  Ø6
1A2Ø  1Ø  C9  ØD  FØ  D2  C9  2C  DØ
1A28  FØ  2Ø  6F  18  29  ØF  FØ  C3
1A3Ø  C9  Ø3  FØ  FA  85  D4  2Ø  Ø6
1A38  1Ø  C9  ØD  FØ  BA  C9  2C  DØ
1A4Ø  E6  2Ø  F9  17  2Ø  Ø6  1Ø  C9
1A48  2C  DØ  F4  2Ø  6Ø  18  85  C9
1A5Ø  86  CA  2Ø  Ø6  1Ø  C9  2Ø  FØ
1A58  F9  C9  ØD  DØ  EC  AD  87  Ø2
1A6Ø  C9  53  DØ  F7  AD  ØØ  CØ  C9
1A68  4Ø  DØ  Ø6  2Ø  A4  F6  4C  93
1A7Ø  1Ø  C9  4C  DØ  D4  2Ø  E3  F6
1A78  4C  93  1Ø  2Ø  Ø1  18  2Ø  3B

1A8Ø  19  2Ø  3B  19  2Ø  ØB  18  2Ø
1A88  52  19  2Ø  13  11  9Ø  ØA  98
1A9Ø  DØ  15  AD  91  Ø2  3Ø  1Ø  1Ø
1A98  Ø8  C8  DØ  ØB  AD  91  Ø2  1Ø
1AAØ  Ø6  2Ø  1A  18  4C  93  1Ø  4C
1AA8  8E  1Ø  2Ø  Ø1  18  2Ø  CØ  1A
1ABØ  4C  93  1Ø  2Ø  55  19  A2  2E
1AB8  A9  24  2Ø  29  18  2Ø  13  18
1ACØ  2Ø  2F  1B  2Ø  E6  1A  2Ø  52
1AC8  19  2Ø  CC  1A  2Ø  CF  1A  2Ø
1ADØ  52  19  A2  Ø4  A9  3Ø  18  ØE
1AD8  92  Ø2  2E  93  Ø2  69  ØØ  2Ø
1AEØ  Ø9  1Ø  CA  DØ  EF  6Ø  A5  FC
1AE8  A6  FB  8D  93  Ø2  8E  92  Ø2
```

```
1AFØ 2Ø 52 19 A5 FC 2Ø FA 1A
1AF8 A5 FB AA 2Ø 52 19 8A 29

1BØØ 7F C9 2Ø Ø8 BØ ØA A9 12
1BØ8 2Ø Ø9 1Ø 8A 18 69 4Ø AA
1B1Ø 8A 2Ø Ø9 1Ø C9 22 FØ Ø4
1B18 C9 62 DØ ØA A9 14 2Ø Ø9
1B2Ø 1Ø A9 22 2Ø Ø9 1Ø 28 BØ
1B28 Ø5 A9 92 2Ø Ø9 1Ø 6Ø 2Ø
1B3Ø 52 19 A6 FB A5 FC AC ØØ
1B38 CØ CØ 4Ø DØ Ø3 4C D9 DC
1B4Ø CØ 4C DØ Ø3 4C 83 CF 4C
1B48 8E 1Ø 2Ø 5B 1B BØ F8 2Ø
1B5Ø 52 19 2Ø 13 18 2Ø C3 1A
1B58 4C 93 1Ø A2 Ø4 A9 ØØ 85
1B6Ø FC 2Ø 17 1C 2Ø 83 1B 85
1B68 FB 2Ø 78 1B 2Ø 92 1B CA
1B7Ø DØ F7 Ø8 2Ø 52 19 28 6Ø
1B78 2Ø Ø6 1Ø C9 ØD FØ ØF C9

1B8Ø 2Ø FØ ØB C9 3Ø 9Ø CØ C9
1B88 3A BØ BC 29 ØF 6Ø 68 68
1B9Ø 18 6Ø 85 FE A5 FC 48 A5
1B98 FB 48 Ø6 FB 26 FC Ø6 FB
1BAØ 26 FC 68 65 FB 85 FB 68
1BA8 65 FC 85 FC Ø6 FB 26 FC
1BBØ A5 FE 65 FB 85 FB A9 ØØ
1BB8 65 FC 85 FC 6Ø 2Ø 17 1C
1BCØ 8D 93 Ø2 48 48 2Ø 52 19
1BC8 2Ø 52 19 68 2Ø 1A 18 2Ø
1BDØ 52 19 68 AA A9 ØØ 2Ø 36
1BD8 1B 2Ø 52 19 2Ø CC 1A 4C
1BEØ 93 1Ø 2Ø F4 1B 2Ø 52 19
1BE8 2Ø 13 18 2Ø 2F 1B 2Ø E6
1BFØ 1A 4C 93 1Ø A2 ØF A9 ØØ
1BF8 85 FB 85 FC 2Ø 17 1C 2Ø

1CØØ 83 1B 2Ø 11 1C 2Ø 78 1B
1CØ8 2Ø 11 1C CA DØ F7 4C 52
```

```
1C1Ø 19 4A 26 FB 26 FC 6Ø 2Ø
1C18 A4 18 C9 2Ø FØ F9 6Ø A9
1C2Ø Ø2 4D 4C E8 8D 4C E8 4C
1C28 93 1Ø 2Ø ØB 18 4C F6 17
1C3Ø 2Ø 2A 1C 18 A5 FB 65 FD
1C38 85 FB A5 FC 65 FE 85 FC
1C4Ø 4C 5Ø 1C 2Ø 2A 1C 2Ø 13
1C48 11 84 FC AD 91 Ø2 85 FB
1C5Ø 2Ø 52 19 2Ø 13 18 4C 93
1C58 1Ø 2Ø 6Ø 1C ØØ 6C EC 1F
1C6Ø 78 AD E5 Ø2 AE E6 Ø2 85
1C68 92 86 93 AD 9E Ø2 AE 9F
1C7Ø Ø2 85 9Ø 86 91 58 6Ø 2Ø
1C78 2A 1C 2Ø 3C 18 2Ø 52 19

1C8Ø AØ ØØ 8C 92 Ø2 8C 93 Ø2
1C88 2Ø 13 11 9Ø 1D AD 94 Ø2
1C9Ø DØ 18 AØ ØØ 18 B1 FB 6D
1C98 92 Ø2 8D 92 Ø2 98 6D 93
1CAØ Ø2 8D 93 Ø2 2Ø 3B 19 4C
1CA8 88 1C AD 93 Ø2 2Ø 1A 18
1CBØ AD 92 Ø2 2Ø 1A 18 4C 93
1CB8 1Ø AD A2 Ø2 DØ Ø4 A5 9E
1CCØ DØ Ø6 68 A8 68 AA 68 4Ø
1CC8 AD 6F Ø2 C9 11 DØ 7D A5
1CDØ D8 C9 18 DØ ED A5 C4 85
1CD8 FD A5 C5 85 FE A9 19 8D
1CEØ 9C Ø2 AØ Ø1 2Ø 8C 1E C9
1CE8 3A FØ 1A C9 2C FØ 16 C9
1CFØ 24 FØ 12 CE 9C Ø2 FØ CA
1CF8 38 A5 FD E9 28 85 FD BØ

1DØØ E1 C6 FE DØ DD 8D 87 Ø2
1DØ8 2Ø 45 1E BØ B5 AD 87 Ø2
1D1Ø C9 3A DØ 11 18 A5 FB 69
1D18 Ø8 85 FB 9Ø Ø2 E6 FC 2Ø
1D2Ø D6 13 4C 39 1D C9 24 FØ
1D28 1A 2Ø ØE 1E 2Ø AB 12 A9
1D3Ø ØØ 8D 8C Ø2 AØ 2C 2Ø 4F
1D38 12 A9 ØØ 85 9E 4C 4A 12
```

```
1D4Ø 4C C2 1C 2Ø 3B 19 2Ø B3
1D48 1A 4C 39 1D C9 91 DØ FØ
1D5Ø A5 D8 DØ EC A5 C4 85 FD
1D58 A5 C5 85 FE A9 19 8D 9C
1D6Ø Ø2 AØ Ø1 2Ø 8C 1E C9 3A
1D68 FØ 1A C9 2C FØ 16 C9 24
1D7Ø FØ 12 CE 9C Ø2 FØ 15 18
1D78 A5 FD 69 28 85 FD 9Ø E1

1D8Ø E6 FE DØ DD 8D 87 Ø2 2Ø
1D88 45 1E 9Ø Ø3 4C C2 1C AD
1D9Ø 87 Ø2 C9 3A FØ Ø6 C9 24
1D98 FØ 1D DØ 27 2Ø 15 1E 38
1DAØ A5 FB E9 Ø8 85 FB BØ Ø2
1DA8 C6 FC 2Ø D9 13 A9 ØØ 85
1DBØ 9E 2Ø 4Ø 1E 4C 96 1Ø 2Ø
1DB8 15 1E 2Ø D5 1Ø 2Ø B6 1A
1DCØ 4C AD 1D 2Ø 15 1E A5 FB
1DC8 A6 FC 85 FD 86 FE A9 1Ø
1DDØ 8D 9C Ø2 38 A5 FD ED 9C
1DD8 Ø2 85 FB A5 FE E9 ØØ 85
1DEØ FC 2Ø ØE 1E 2Ø AB 12 2Ø
1DE8 13 11 FØ Ø7 BØ F3 CE 9C
1DFØ Ø2 DØ EØ EE 8B Ø2 AD 8B
1DF8 Ø2 2Ø B9 13 A2 ØØ A1 FB

1EØØ 8E 8C Ø2 A9 2C 2Ø 4D 19
1EØ8 2Ø 52 12 4C AD 1D A2 ØØ
1E1Ø A1 FB 4C C3 12 A9 83 85
1E18 C8 85 FE A9 ØØ 85 C7 A9
1E2Ø 28 85 FD AØ CØ A2 Ø4 88
1E28 B1 C7 91 FD 98 DØ F8 C6
1E3Ø C8 C6 FE CA DØ F1 A2 27
1E38 A9 2Ø 9D ØØ 8Ø CA 1Ø FA
1E4Ø A9 13 4C Ø9 1Ø CØ 28 DØ
1E48 Ø2 38 6Ø 2Ø 8C 1E C9 2Ø
1E5Ø FØ F3 88 2Ø 75 1E AA 2Ø
1E58 75 1E 85 FB 86 FC A9 FF
1E6Ø 8D A2 Ø2 85 A7 A5 AA FØ
1E68 ØA A5 A9 A4 C6 91 C4 A9
```

```
1E7Ø  ØØ 85 AA 18 6Ø 2Ø 8C 1E
1E78  2Ø 99 18 ØA ØA ØA ØA 8D

1E8Ø  97 Ø2 2Ø 8C 1E 2Ø 99 18
1E88  ØD 97 Ø2 6Ø B1 FD C8 29
1E9Ø  7F C9 2Ø BØ Ø2 Ø9 4Ø 6Ø
1E98  4Ø Ø2 45 Ø3 DØ Ø8 4Ø Ø9
1EAØ  3Ø 22 45 33 DØ Ø8 4Ø Ø9
1EA8  4Ø Ø2 45 33 DØ Ø8 4Ø Ø9
1EBØ  4Ø Ø2 45 B3 DØ Ø8 4Ø Ø9
1EB8  ØØ 22 44 33 DØ 8C 44 ØØ
1ECØ  11 22 44 33 DØ 8C 44 9A
1EC8  1Ø 22 44 33 DØ Ø8 4Ø Ø9
1EDØ  1Ø 22 44 33 DØ Ø8 4Ø Ø9
1ED8  62 13 78 A9 ØØ 21 81 82
1EEØ  ØØ ØØ 59 4D 91 92 86 4A
1EE8  85 9D 2C 29 2C 23 28 24
1EFØ  59 ØØ 58 24 24 ØØ 1C 8A
1EF8  1C 23 5D 8B 1B A1 9D 8A

1FØØ  1D 23 9D 8B 1D A1 ØØ 29
1FØ8  19 AE 69 A8 19 23 24 53
1F1Ø  1B 23 24 53 19 A1 ØØ 1A
1F18  5B 5B A5 69 24 24 AE AE
1F2Ø  A8 AD 29 ØØ 7C ØØ 15 9C
1F28  6D 9C A5 69 29 53 84 13
1F3Ø  34 11 A5 69 23 AØ D8 62
1F38  5A 48 26 62 94 88 54 44
1F4Ø  C8 54 68 44 E8 94 ØØ B4
1F48  Ø8 84 74 B4 28 6E 74 F4
1F5Ø  CC 4A 72 F2 A4 8A ØØ AA
1F58  A2 A2 74 74 74 72 44 68
1F6Ø  B2 32 B2 ØØ 22 ØØ 1A 1A
1F68  26 26 72 72 88 C8 C4 CA
1F7Ø  26 48 44 44 A2 C8 ØD 2Ø
1F78  2Ø 2Ø 2Ø 5Ø 43 2Ø 2Ø 49
```

1F8Ø	52	51	2Ø	2Ø	53	52	2Ø	41
1F88	43	2Ø	58	52	2Ø	59	52	2Ø
1F9Ø	53	5Ø	41	42	43	44	46	47
1F98	48	4C	4D	4E	51	52	53	54
1FAØ	57	58	2C	3A	3B	24	23	22
1FA8	2B	2D	4F	5A	4B	25	26	45
1FBØ	4D	14	38	17	25	11	35	12
1FB8	9D	11	B5	16	C8	11	BF	19
1FCØ	BE	13	55	17	B9	16	5A	19
1FC8	BF	19	29	11	C9	16	B5	19
1FDØ	48	13	23	14	93	19	AA	1A
1FD8	4A	1B	BD	1B	3Ø	1C	43	1C
1FEØ	7B	1A	1F	1C	59	1C	E2	1B
1FE8	77	1C	B2	19	ØØ	1Ø	55	13
1FFØ	EB	15	B9	1C	C6	15	8E	1Ø
1FF8	BC	18	3Ø	35	32	37	38	31

Program 3. Checksum For Micromon Plus.

```
1Ø DATA 15965,14778,13Ø59,14282,14
    416,17693,12979,129Ø3,1767
    6,2176Ø
2Ø DATA 14416,17693,12979,129Ø3
1ØØ Q=23296
11Ø FOR BLOCK=1TO8
12Ø FOR BYTE=ØTO127
13Ø X=PEEK(Q+BYTE):CK=CK+X
14Ø NEXT BYTE
15Ø READ SUM
16Ø IF SUM <> CK THEN PRINT" ERROR
    IN BLOCK #"BLOCK:GOTO17Ø
165 PRINT"                  BLOCK"
    BLOCK" IS CORRECT
17Ø CK=Ø:Q=Q+128
18Ø NEXT BLOCK
19Ø PRINT"ANY REMAINING PROBLEMS AR
    E EITHER WITHIN THE FINAL"

2ØØ PRINT"SHORT BLOCK OR WITHIN DAT
    A STATEMENTS IN THIS PROGR
    AM."
```

Program 4. Micromon Plus.

```
5BØØ 78 A5 9Ø A6 91 CD EE 6F
5BØ8 DØ Ø5 EC EF 6F FØ 3Ø 8D
5B1Ø 9E Ø2 8E 9F Ø2 AD EE 6F
5B18 AE EF 6F 85 9Ø 86 91 A5
5B2Ø 92 A6 93 8D E5 Ø2 8E E6
5B28 Ø2 AD 3C 5F AE 3D 5F 8D
5B3Ø E3 Ø2 8E E4 Ø2 AD FØ 6F
5B38 AE F1 6F 85 92 86 93 AD
5B4Ø 3E 5F AE 3F 5F EØ 8Ø BØ
5B48 Ø8 85 34 86 35 85 3Ø 86
5B5Ø 31 A9 1Ø 8D 84 Ø2 8D 85
5B58 Ø2 A9 ØØ 8D 86 Ø2 8D A2
5B6Ø Ø2 8D E7 Ø2 8D E8 Ø2 58
5B68 ØØ A2 ØC DD 15 5F DØ 13
5B7Ø 8D 87 Ø2 8A ØA AA BD 22
5B78 5F 85 FB BD 23 5F 85 FC

5B8Ø 6C FB ØØ CA 1Ø E5 4C 8E
5B88 6Ø 2Ø 39 64 2Ø 13 61 9Ø
5B9Ø 17 2Ø EF 6Ø 8E 8C Ø2 2Ø
5B98 52 62 2Ø AB 5B 2Ø AB 62
5BAØ 2Ø 93 5C 2Ø AE 68 DØ E4
5BA8 4C 9B 6Ø A2 1E 2Ø F1 6Ø
5BBØ AØ ØØ B1 FB 2Ø 6Ø 5C CC
5BB8 8B Ø2 C8 9Ø F5 6Ø A5 BØ
5BCØ C9 Ø3 DØ 19 2Ø Ø6 6Ø AA
5BC8 A9 Ø4 EØ ØD FØ Ø9 2Ø 6F
5BDØ 68 29 1F C9 Ø4 9Ø AF 2Ø
5BD8 E3 5B 4C 9B 6Ø 2Ø CC FF
5BEØ 4C 93 6Ø 85 BØ 85 D4 2Ø
5BE8 Ø9 5C AE ØØ CØ EØ 4Ø DØ
5BFØ ØB 2Ø BA FØ 2Ø 2D F1 A5
5BF8 96 DØ E2 6Ø EØ 4C DØ 5D

5CØØ 2Ø D5 FØ 2Ø 48 F1 4C F7
5CØ8 5B A9 ØØ 85 96 8D FC Ø3
5C1Ø 85 ØD 8D E8 Ø2 6Ø 2Ø 39
```

```
5C18 64 AE 94 Ø2 DØ 1Ø 2Ø 13
5C2Ø 61 9Ø ØB 2Ø 31 5C 2Ø 93
5C28 5C 2Ø AE 68 DØ EB 4C A8
5C3Ø 5B A2 Ø5 2Ø F1 6Ø 2Ø 13
5C38 68 A2 Ø2 2Ø F1 6Ø A9 1Ø
5C4Ø 2Ø Ø3 69 A9 1Ø 2Ø B9 63
5C48 A2 Ø4 2Ø F1 6Ø AØ 1Ø A2
5C5Ø ØØ A1 FB 2Ø 6Ø 5C 2Ø 3B
5C58 69 88 DØ F5 6Ø 4C 8E 6Ø
5C6Ø 29 7F C9 2Ø BØ Ø2 A9 2Ø
5C68 4C Ø9 6Ø 2Ø Ø6 6Ø C9 ØD
5C7Ø FØ 19 C9 2Ø DØ Ø3 2Ø 17
5C78 6C C9 58 FØ 5Ø 2Ø 71 5D

5C8Ø 8E E8 Ø2 A2 Ø2 2Ø A7 5C
5C88 4C 9B 6Ø A2 Ø4 2Ø C1 5C
5C9Ø 4C 9B 6Ø 2Ø 55 69 AE E7
5C98 Ø2 FØ 31 CE E7 Ø2 DØ 2C
5CAØ AE E8 Ø2 FØ 1A A2 Ø6 2Ø
5CA8 C1 5C A2 14 2Ø F1 6Ø BD
5CBØ A3 Ø2 2Ø Ø9 6Ø E8 EC E8
5CB8 Ø2 DØ F4 A2 Ø3 DØ Ø2 A2
5CCØ Ø9 2Ø 55 69 CA DØ FA A9
5CC8 39 8D E7 Ø2 6Ø A9 ØØ 8D
5CDØ E7 Ø2 8D E8 Ø2 4C 9B 6Ø
5CD8 2Ø Ø9 5C 2Ø CC FF 2Ø Ø6
5CEØ 6Ø C9 ØD FØ 16 C9 24 FØ
5CE8 24 48 2Ø 9E 5D 68 2Ø Ø9
5CFØ 6Ø 2Ø Ø6 6Ø C9 ØD DØ F6
5CF8 4C DD 5B 2Ø 52 69 2Ø C5

5DØØ 5D 2Ø Ø6 6Ø C9 ØD FØ FØ
5DØ8 2Ø Ø9 6Ø DØ F4 A2 ØØ 2Ø
5D1Ø 82 5D 2Ø 8B 5D 2Ø 55 69
5D18 2Ø 55 69 AØ Ø3 DØ Ø2 AØ
5D2Ø Ø2 84 D1 A9 Ø8 85 AF 2Ø
5D28 Ø6 6Ø AA A4 96 DØ 36 2Ø
5D3Ø Ø6 6Ø A4 96 DØ 2F C6 D1
5D38 DØ ED 2Ø 36 6B 2Ø 52 69
5D4Ø 2Ø Ø6 6Ø FØ Ø5 2Ø Ø9 6Ø
```

```
5D48 DØ F6 2Ø 55 69 A9 ØØ 85
5D5Ø AF 2Ø E4 FF FØ C9 DØ Ø5
5D58 2Ø E4 FF FØ FB C9 2Ø FØ
5D6Ø F7 C9 Ø3 DØ BA 2Ø 12 5E
5D68 2Ø 55 69 4C 93 6Ø 2Ø 17
5D7Ø 6C C9 22 DØ 7B A2 ØØ 2Ø
5D78 Ø6 6Ø C9 ØD FØ ØC C9 22

5D8Ø FØ Ø8 9D A3 Ø2 E8 EØ 4Ø
5D88 9Ø ED 6Ø 86 D1 A9 A3 85
5D9Ø DA A9 Ø2 85 DB 2Ø CC FF
5D98 2Ø F3 5D 4C C9 5D A9 Ø8
5DAØ 85 D4 85 BØ AC ØØ CØ CØ
5DA8 4Ø DØ ØB 2Ø BA FØ A9 6F
5DBØ 2Ø 28 F1 4C F7 5B CØ 4C
5DB8 DØ 36 2Ø D5 FØ A9 6F 2Ø
5DCØ 43 F1 4C F7 5B A9 6F 85
5DC8 D3 A9 Ø8 85 D4 85 AF AC
5DDØ ØØ CØ CØ 4Ø DØ ØB 2Ø B6
5DD8 FØ A5 D3 2Ø 64 F1 4C F7
5DEØ 5B CØ 4C DØ ØB 2Ø D2 FØ
5DE8 A5 D3 2Ø 93 F1 4C F7 5B
5DFØ 4C 8E 6Ø A9 Ø8 85 D4 A9
5DF8 6Ø 85 D3 AD ØØ CØ C9 4Ø

5EØØ DØ Ø6 2Ø 66 F4 4C F7 5B
5EØ8 C9 4C DØ E4 2Ø A5 F4 4C
5E1Ø F7 5B A9 ØØ 85 AF AD ØØ
5E18 CØ C9 4Ø DØ Ø3 4C 8F F3
5E2Ø C9 4C DØ CC 4C CE F3 A9
5E28 Ø2 2C 4C E8 Ø8 A9 ØE 28
5E3Ø FØ Ø2 Ø9 8Ø 2Ø Ø9 6Ø 4C
5E38 93 6Ø 2Ø Ø9 5C 2Ø 6E 5D
5E4Ø 2Ø 8B 5D 2Ø Ø6 6Ø 8D FB
5E48 ØØ 2Ø Ø6 6Ø 8D FC ØØ 2Ø
5E5Ø 12 5E 2Ø 52 69 A9 24 A2
5E58 2Ø 2Ø 29 68 2Ø 13 68 4C
5E6Ø 93 6Ø 2Ø 6Ø 6C ØØ 6C 3E
5E68 5F AØ Ø8 84 D4 AØ 4C 8C
```

```
5E70 87 02 A0 00 4C C4 69 20
5E78 17 6C 29 9F 4C 34 5E 4C

5E80 8E 60 20 A4 68 20 6F 68
5E88 29 1F C9 04 90 F1 85 D4
5E90 20 2A 6C A5 FD A6 FE 8D
5E98 92 02 8E 93 02 20 3C 68
5EA0 A5 D4 20 E3 5B A9 02 20
5EA8 09 60 20 52 69 20 13 61
5EB0 90 0F AE 94 02 D0 0A A1
5EB8 FB 20 1A 68 20 3B 69 D0
5EC0 E9 A9 03 20 09 60 20 EF
5EC8 60 20 CC FF 20 F8 60 4C
5ED0 7D 6C 20 09 5C 20 01 68
5ED8 20 6E 5D 86 D1 20 04 5F
5EE0 20 8D 5D 20 06 60 20 06
5EE8 60 A9 00 85 AF AD 00 C0
5EF0 C9 40 D0 06 20 52 F3 4C
5EF8 01 5F C9 4C D0 81 20 8C

5F00 F3 4C 12 6A AD 00 C0 C9
5F08 40 D0 03 4C 0A F4 C9 4C
5F10 D0 EA 4C 49 F4 50 C4 49
5F18 CD 40 3E DA 4A CB CC 5E
5F20 55 59 BE 5B 89 5B 6B 5C
5F28 16 5C D8 5C D8 5C 27 5E
5F30 3A 5E 62 5E 69 5E 77 5E
5F38 82 5E D2 5E 69 5B 00 5B
5F40 31 30 32 31 38 31 AA AA
```

VIC Micromon

VIC machine language programmers: here's one of the most valuable tools there is for working in machine language. Thirty-four commands are at your disposal including single-step, hex conversion, search, EPROM routines, and a relocator. If you aren't yet working with machine language, the instructions for entering and using this program are easy to follow. As presented, this program takes up 4K of memory from $4000 (16384 decimal) to $4FFF (20479), but there are instructions for locating it elsewhere in RAM memory. To enter Micromon directly, see the Tiny PEEKer/POKEr program with Supermon 64 (in this Appendix). The commands for VIC Micromon are the same as the PET/CBM version except as noted below.

VIC Micromon Instructions

Initialize Memory And Screen Pointers

.I 1000 1E00 1E

Define low memory as $1000 and high memory as $1E00 regardless of the memory present. The screen is defined to start at the $1E page of memory. The screen memory should always be on an even page within the range of $1000 to $1E00. Odd page values result in incorrect setup and operation of the VIC display. Although 3K of RAM can be added at $400 to $FFF, this memory is not accessible for use as screen memory.

Memory pages at $000 and $200 are accessible, but are not usable since they are used for BASIC and kernal storage, working buffers, and stack area. If the screen page is within the low to high memory range specified, there can be usage conflict of the screen memory pages. If the "I" command is used and exit is made to BASIC, the NEW command must be invoked in the BASIC environment to clean up the memory pointers used by BASIC.

Jump To Micromon Subroutine

.J 2000

The subroutine at $2000 is called while remaining in the VIC Micromon environment. The assembly language subroutine should exit by using a RTS instruction, which causes a return to the command input section of VIC Micromon. The machine image as shown by the Register display command is not used, nor is it disturbed when the subroutine returns to the VIC Micromon.

Load

.L 2000 "TEST FILE" 01

Search for and, if found, load into memory the data file on device #1 named TEST FILE. If the name is not specified, the first file found is loaded. The data is loaded into memory starting at location $2000. The last address loaded is determined by the length of the binary data file. If the device number is not specified, it defaults to device #1, which is the VIC cassette tape. The original memory addresses and name of the last file read can be inspected by doing a Memory display of the tape buffer which is at $375 for VIC Micromon.

Print Switcher

.P

If the output is to the screen, then switch the ouput to the RS-232 channel (device #2). If the output is not to the screen, restore the output to the screen with the RS-232 channel left active until the RS-232 output buffer is drained. Note that opening the RS-232 channel grabs 512 bytes for I/O buffering from the top of memory.

.P 0000

Regardless of the output, clear the RS-232 channel and set output to the screen.

.P CCBB

If the output is to the screen, set CC into the RS-232 command register at location $294 and BB into the RS-232 control register at location $293. Output is then switched to the RS-232 channel. This command is invalid if output is not currently to the screen.

Command Register Format

Field	Use	Value	Description
7,6,5	Parity Options	--0	Parity disabled
		0 0 1	Odd parity
		0 1 1	Even parity
		1 0 1	Mark transmitted
		1 1 1	Space transmitted
4	Duplex	0	Full duplex
		1	Half duplex
3,2,1	Unused		
0	Handshake	0	3 line
		1	x line

Control Register Format

Field	Use	Value	Description
7	Stop Bits	0	1 stop bit
		1	2 stop bits
6,5	Word Length	0 0	8 bits
		0 1	7 bits
		1 0	6 bits
		1 1	5 bits
4	Unused		
3,2,1,0	Baud Rate	0 0 0 0	User rate
		0 0 0 1	50 Baud
		0 0 1 0	75
		0 0 1 1	110
		0 1 0 0	134.5
		0 1 0 1	150
		0 1 1 0	300
		0 1 1 1	600
		1 0 0 0	1200
		1 0 0 1	1800
		1 0 1 0	2400

Save

```
.S 2000 3000 ''TEST FILE'' 01
```

Save memory from $2000 up to, but not including, $3000 onto device #1, which is the VIC cassette tape. If the device number is not specified, it defaults to device #1. The name *TEST FILE* is placed in the file header for the file saved.

Verify

```
.V 2000 ''TEST FILE'' 01
```

Search for and verify, if found, the data file on device #1 named ''TEST FILE.'' If the name is not specified, the first file found is verified. The data is verified by reading the file and comparing it to the data in memory starting at location $2000. If not specified, the device defaults to device #1. If there is a mismatch, the message ERROR is output to the screen at the end of the file verification.

Command End Tone

```
.(
```

Enable the command end tone. A continuous tone will be generated at the end of execution of the next command. The tone can be turned off but still be enabled by just hitting the carriage return. No tone is generated if there is a syntax error while inputting the next command.

```
.)
```

Disable the command end tone.

Program EPROM

```
.π 2800 2FFF 00
```

Program the 2716 type EPROM via the EPROM programmer on the VIC User I/O port with data read from memory starting at location $2800 and ending at location $2FFF. The last input parameter specifies in hex the starting 256 byte page offset on the EPROM. If the low order byte of the starting memory address is zero and the offset is zero, then the programming starts with the first byte of the EPROM. For example, to program only the last byte of the 2K EPROM with a data byte from location $2FFF in memory, the command would be:

```
.π 2FFF 2FFF 07
```

During programming, a compare of EPROM to memory is done for each data byte just after it is written to the EPROM. Any mismatch due to failure to program the EPROM results in output to the screen of the mismatched memory location. If programming must be terminated early, just hit the STOP key. No other means should be used to abort EPROM programming. *A warm restart or power down while programming can damage the EPROM.*

Read EPROM

```
.£ 2000 27FF 00
```

Load memory starting at location $2000 and ending at location $27FF with data read from the EPROM via the EPROM programmer on the VIC User I/O port. The last input parameter specifies in hex the starting 256 byte page offset on the EPROM. If the low order byte of the starting memory address is zero and the offset is zero, then reading starts with the first byte of the EPROM. For example, to read only the last byte of the 2K EPROM and load that byte into memory at location $10FF, the command would be:

```
.£ 10FF 10FF 07
```

During memory load, a compare of EPROM to memory is done for each data byte just after it is written to memory. Any mismatch because of failure to write the memory with data from the EPROM results in output to the screen of the mismatched memory location. The STOP key can be used to terminate the command early.

Compare EPROM

```
.=3000 37FF 00
```

Compare memory starting at location $3000 and ending at location $37FF with data read from the EPROM via the EPROM programmer on the VIC User I/O port. The last input parameter specifies in hex the starting 256 byte page offset on the EPROM. If the low order byte of the starting memory address is zero and the offset is zero, then the reading starts with the first byte of the EPROM. For example, to read only the last byte of the 2K EPROM and compare that with the data byte in memory at location $37FF, the command would be:

```
.=37FF 37FF 07
```

Any mismatch between the EPROM and corresponding memory data results in output to the screen of the mismatched memory location. The STOP key can be used to terminate the command early.

Commands for VIC Micromon

VIC Micromon Instruction	Command
SIMPLE ASSEMBLER	A
BREAK SET	B
COMPARE MEMORY	C
DISASSEMBLER	D
EXIT VIC MICROMON	E
FILL MEMORY	F
GO RUN	G
HUNT MEMORY	H
INITIAL MEMORY & SCREEN PTRS	I
JUMP TO SUBROUTINE	J
LOAD MEMORY FROM DEVICE	L
MEMORY DISPLAY	M
NEW LOCATER	N
OFFSET OR BRANCH CALCULATE	O
PRINT SWITCHER	P
QUICK TRACE	Q
REGISTER DISPLAY	R
SAVE MEMORY TO DEVICE	S
TRANSFER MEMORY	T
VERIFY MEMORY FROM DEVICE	V
WALK CODE	W
EXIT TO BASIC	X
ASCII CONVERSION	"
DECIMAL CONVERSION	#
HEXADECIMAL CONVERSION	$
BINARY CONVERSION	%
CHECKSUM MEMORY	&
COMMAND END TONE ENABLE	(
COMMAND END TONE DISABLE	)
ADDITION	+
SUBTRACTION	−
LOAD MEMORY FROM EPROM	£
PROGRAM EPROM FROM MEMORY	π
COMPARE EPROM TO MEMORY	=

Of the set of commands available on the PET version of Micromon, only two were removed in the conversion to the VIC. These were the K (Kill Micromon) and Z (change character sets) commands. The K command is not necessary since the VIC doesn't have the TIM monitor. The function of the Z command, which is to change character sets, is already provided for on the VIC by pressing the VIC shift and Commodore keys at the same time. The rest of the commands described for the PET Micromon (see elsewhere in this appendix) all apply identically to the commands for VIC Micromon, with the exception of the LOAD and SAVE commands, which have different formats.

VIC Micromon is always entered from VIC BASIC by a SYS 16384 when it resides at $4000 to $4FFF. Either the E (Exit VIC Micromon) or the X (Exit to BASIC) command would be used to exit VIC Micromon and return to the BASIC environment. The difference between these two commands is that the X command leaves the VIC Micromon vectors in the IRQ and BRK interrupt vector locations while in the BASIC environment. Also, the tape buffer is left defined as beginning at $375. Thus, certain IRQ interrupt conditions such as the moving of the cursor to the top or bottom of the screen with output from a D, M, or $ command displayed will cause scrolling and reentry into VIC Micromon. Also, if a BRK instruction is executed, VIC Micromon will be reentered via its BRK interrupt handler.

The E command restores the IRQ and BRK interrupt vectors and resets the tape buffer pointer to a value of $33C prior to exit to the VIC BASIC environment. Thus all active linkages and vectors to VIC Micromon are removed, and the VIC behaves as if VIC Micromon never existed. In particular, the E command should be used to exit VIC Micromon when the normal VIC cassette tape LOAD, SAVE, and VERIFY commands are to be used in the BASIC environment. Otherwise, invalid results are likely to occur with some tape operations.

Both the E and X commands expect the stack pointer value (as shown for SP by the Register display command) to be the same as when VIC Micromon was first entered via the BASIC SYS command. If the value of SP or the part of the stack pointed to by SP is overwritten, such as by the execution of faulty code, a clean exit to BASIC by the E and X commands is unlikely. However, both the E and X commands do check if BASIC has been initialized, and if not, exit to BASIC is via an indirect jump to the address given at location $C000. The address given in location $C000 is $E378, which is the entry to initialize BASIC. In this case, the value of SP and the contents of the stack aren't important. Once in BASIC and regardless of how the exit from VIC Micromon was made, any subsequent access to VIC Micromon at $4000 is always by a SYS16384.

VIC Micromon as given here is located from $4000 to $4FFF. It can be relocated to any 256 byte page boundary by making the changes, as shown in the following example, which relocate VIC Micromon from $4000 to $6000.

The example begins with VIC Micromon at $4000 and ends with a relocated VIC Micromon in RAM at $6000 as well as the original at $4000.

```
.T 4000 4FFF 6000
.N 6000 6003 2000 4000 4FFF
.N 6012 6E6D 2000 4000 4FFF
.N 6FB5 6FFE 2000 4000 4FFF W
```

Location	Old Value	New Value
6018	45	65
602A	43	63
6392	4C	6C
6650	45	65
66E7	45	65
6897	43	63

In order to access the relocated VIC Micromon at $6000, exit using the E command and then from BASIC use SYS24576.

Cartridge And Checksum

The VIC-20 treats cartridge programs located at $A000 in a special way. On power-up, a test is made for the existence of the $A000 cartridge program, and if one exists, an indirect jump is made to the address specified at location $A000. This jump is made after the stack pointer is initialized, but before anything else is done. Because kernal initialization has not occurred, any cartridge program using kernal I/O routines must do kernal initialization before using those routines.

VIC Micromon as presented here has the kernal initialization calls built in so that it can easily be relocated and used as a cartridge program at $A000. Besides making the changes to relocate it to $A000, the only additional changes are to the first four bytes of VIC Micromon.

Location	Contents
A000	09
A001	A0
A002	C7
A003	FE

Power-up with VIC Micromon installed as a cartridge at $A000 will result in immediate entry into VIC Micromon. Because BASIC is not initialized when the E or X command is used after power-up, the exit to BASIC will be via an indirect jump to the address given in location $C000, which is the entry to initialization of BASIC. Once in BASIC, subsequent access of VIC Micromon at $A000 must be made to location $A012, which is done via a SYS40978.

There is one last point, or rather one last byte, in VIC Micromon which is not used for anything other than to make the 4K byte checksum of VIC Micromon come out to a rounded up page value. For example, the VIC Micromon from $4000 to $4FFF has a data byte value of $E6 at location $4FFF that results in a checksum of $BF00. This provides an easy way to verify the integrity of VIC Micromon without having to memorize or look up a checksum.

Three Notes On VIC Micromon

Using the VIC Micromon tape commands L, S, and V on a VIC-20 with 3K of RAM installed at $400 to $FFF will result in overwrite of $400 to $438 with file header characters (blanks). This is due to the tape buffer being relocated to $375 while in VIC Micromon from the normal $33C. The normal VIC cassette commands will work properly and not overwrite this area when you EXIT from VIC Micromon. This is because VIC Micromon restores the tape buffer pointer value to $33C when an EXIT is done. This problem does not occur if the 3K RAM at $400 to $FFF is not installed.

If the I (Initialize memory and screen pointers) command was used in VIC Micromon and you EXIT, then the RUN/STOP plus RESTORE should be used in addition to the NEW command to clean up the BASIC environment.

Any binary image saved on cassette tape with the VIC Micromon ''S'' command can be loaded in the normal VIC-20 BASIC environment by using the command: LOAD'''',1,1 which looks for the next program on tape and LOADs it into the same part of memory that it came from (see page 9 of VIC-20 *Programmer's Reference Guide).*

Checksum

There's a good amount of typing to do to enter the VIC Micromon program. Use the following BASIC program (after you've SAVEd a copy of your efforts) to locate any errors you might have made.

Program 1. Micromon Checksum.

```
1 IFPEEK(20478)=67ANDPEEK(20479)=73THENRUN10
2 PRINT"VIC20 MICROMON LOAD &":PRINT"VERIFIC
   ATION PROGRAM.":PRINT
```

```
3 PRINT:PRINT:PRINT"AT LEAST 4K BYTES OF":PR
   INT"RAM MUST BE INSTALLED"
4 PRINT"AT 16384 ($4000) ELSE":PRINT"LOAD WI
   LL FAIL.":PRINT
5 PRINT"IF LOADED & VERIFIED":PRINT"OK, MICR
   OMON WILL BE":PRINT"ENTERED AUTOMATIC
   ALLY."
6 LOAD"",1,1
10 DATA 13328,16867,15061,13732,14507,13829,1
   3267,12747,16288,13920
20 DATA 14355,11977,11877,13583,11338,15173,1
   2337,14852,14051,15713
30 DATA 13442,15242,14746,15059,13134,15848,1
   5858,17856,13327,8601
40 DATA 12171,10074
100 Q=16384
110 FOR BLOCK=1TO32
120 FOR BYTE=0TO127
130 X=PEEK(Q+BYTE):CK=CK+X
140 NEXT BYTE
150 READ SUM
160 IF SUM <> CK THEN PRINT"ERROR IN BLOCK #"B
    LOCK:ERR=1:GOTO170
165 PRINT"BLOCK #"BLOCK" OK"
170 CK=0:Q=Q+128
180 NEXT BLOCK
190 IFERR=1THENPRINT"LOAD FAILED":END
200 SYS16384
```

Program 2. VIC Micromon.

```
4000 78 4C 15 40 41 30 C3 C2
4008 CD 20 8D FD 20 52 FD 20
4010 18 E5 20 F9 FD A9 DF A2
4018 45 8D 16 03 8E 17 03 AD
4020 14 03 AE 15 03 C9 91 D0
4028 04 E0 43 F0 09 8D 60 03
4030 8E 61 03 20 94 48 A9 75
4038 85 B2 A9 80 8D 8A 02 85
4040 9D A2 D6 20 5D 4E 8E 48
4048 03 8E 64 03 58 00 CE 3D
4050 03 D0 03 CE 3C 03 20 AE
```

```
4Ø58 45 A2 42 A9 2A 4C 3D 49
4Ø6Ø A9 3F 2Ø D2 FF A9 ØØ 2C
4Ø68 A9 ØF 8D ØE 9Ø 2Ø AE 45
4Ø7Ø A9 2E 2Ø D2 FF A9 ØØ 8D
4Ø78 4E Ø3 8D 56 Ø3 8D 64 Ø3

4Ø8Ø A2 7F 9A 2Ø 8C 48 C9 2E
4Ø88 FØ F9 C9 2Ø FØ F5 A2 24
4Ø9Ø DD 9Ø 4F DØ 13 8D 49 Ø3
4Ø98 8A ØA AA BD B5 4F 85 FB
4ØAØ BD B6 4F 85 FC 6C FB ØØ
4ØA8 CA 1Ø E5 4C 6Ø 4Ø A2 Ø2
4ØBØ DØ Ø2 A2 ØØ B4 FB DØ Ø9
4ØB8 B4 FC DØ Ø3 EE 56 Ø3 D6
4ØCØ FC D6 FB 6Ø A9 ØØ 8D 4E
4ØC8 Ø3 2Ø 13 42 A2 Ø9 2Ø 38
4ØDØ 49 CA DØ FA 6Ø A2 Ø2 B5
4ØD8 FA 48 BD 53 Ø3 95 FA 68
4ØEØ 9D 53 Ø3 CA DØ F1 6Ø AD
4ØE8 54 Ø3 AC 55 Ø3 4C F4 4Ø
4ØFØ A5 FD A4 FE 38 E5 FB 8D
4ØF8 53 Ø3 98 E5 FC A8 ØD 53

41ØØ Ø3 6Ø A9 ØØ FØ Ø2 A9 Ø1
41Ø8 8D 57 Ø3 2Ø CB 47 2Ø AE
411Ø 45 2Ø FØ 4Ø 2Ø 21 48 9Ø
4118 18 2Ø E7 4Ø 9Ø 7F 2Ø 59
412Ø 41 E6 FD DØ Ø2 E6 FE 2Ø
4128 1F 49 AC 56 Ø3 DØ 6E FØ
413Ø E8 2Ø E7 4Ø 18 AD 53 Ø3
4138 65 FD 85 FD 98 65 FE 85
414Ø FE 2Ø D5 4Ø 2Ø 59 41 2Ø
4148 E7 4Ø BØ 51 2Ø AE 4Ø 2Ø
415Ø B2 4Ø AC 56 Ø3 DØ 46 FØ
4158 EB A2 ØØ A1 FB AC 57 Ø3
416Ø FØ Ø2 81 FD C1 FD FØ ØB
4168 2Ø F8 47 2Ø 38 49 2Ø E1
417Ø FF FØ 2A 6Ø 2Ø E6 47 2Ø
4178 A1 49 FØ 1E AE 56 Ø3 DØ
```

```
4180 1C 20 F0 40 90 17 60 20
4188 54 48 8D 4B 03 20 7C 41
4190 AD 4B 03 81 FB 20 1F 49
4198 D0 F3 4C 60 40 4C 68 40
41A0 20 74 41 20 8C 48 C9 27
41A8 D0 12 20 8C 48 9D 65 03
41B0 E8 20 A4 49 F0 20 E0 20
41B8 D0 F3 F0 1A 8E 59 03 20
41C0 5F 48 90 D6 9D 65 03 E8
41C8 20 A4 49 F0 09 20 57 48
41D0 90 C8 E0 20 D0 EE 8E 4A
41D8 03 20 AE 45 A2 00 A0 00
41E0 B1 FB DD 65 03 D0 0A C8
41E8 E8 EC 4A 03 D0 F2 20 68
41F0 41 20 1F 49 20 7C 41 B0
41F8 E3 20 2B 44 20 F0 40 90

4200 0D A0 2C 20 C4 40 20 6F
4208 42 20 E1 FF D0 EE 20 B6
4210 45 D0 8A 20 2D 49 20 F8
4218 47 20 38 49 20 C9 4D 48
4220 20 CF 42 68 20 E6 42 A2
4228 06 E0 03 D0 14 AC 4D 03
4230 F0 0F AD 58 03 C9 E8 B1
4238 FB B0 1D 20 65 42 88 D0
4240 F1 0E 58 03 90 0E BD E9
4248 4E 20 99 45 BD EF 4E F0
4250 03 20 99 45 CA D0 D2 60
4258 20 7B 42 AA E8 D0 01 C8
4260 98 20 65 42 8A 8E 4A 03
4268 20 FF 47 AE 4A 03 60 AD
4270 4D 03 20 7A 42 85 FB 84
4278 FC 60 38 A4 FC AA 10 01

4280 88 65 FB 90 01 C8 60 A8
4288 4A 90 0B 4A B0 17 C9 22
4290 F0 13 29 07 09 80 4A AA
4298 BD 98 4E B0 04 4A 4A 4A
42A0 4A 29 0F D0 04 A0 80 A9
42A8 00 AA BD DC 4E 8D 58 03
42B0 29 03 8D 4D 03 98 29 8F
```

```
42B8 AA 98 AØ Ø3 EØ 8A FØ ØB
42CØ 4A 9Ø Ø8 4A 4A Ø9 2Ø 88
42C8 DØ FA C8 88 DØ F2 6Ø B1
42DØ FB 2Ø 65 42 A2 Ø1 2Ø CE
42D8 4Ø CC 4D Ø3 C8 9Ø FØ A2
42EØ Ø3 CØ Ø3 9Ø F1 6Ø A8 B9
42E8 F6 4E 8D 54 Ø3 B9 36 4F
42FØ 8D 55 Ø3 A9 ØØ AØ Ø5 ØE
42F8 55 Ø3 2E 54 Ø3 2A 88 DØ

43ØØ F6 69 3F 2Ø D2 FF CA DØ
43Ø8 EA 4C 38 49 2Ø E6 47 A9
431Ø Ø3 2Ø 9E 43 AØ 2C 4C 3C
4318 45 ØØ ØØ ØØ A9 3C 8D 13
432Ø 91 2Ø 3A 43 A9 FF 8D 12
4328 91 A5 FB AØ 18 2Ø 34 43
433Ø A5 FF AØ 14 8D 1Ø 91 8C
4338 11 91 AØ 1C 8C 11 91 6Ø
434Ø 2Ø 54 48 85 FF 2Ø AE 45
4348 2Ø 6E 41 2Ø 7C 41 2Ø 1C
435Ø 43 AD 49 Ø3 ØA Ø8 9Ø 17
4358 A1 FB 8D 1Ø 91 78 A9 C4
436Ø 8D 19 91 A9 3C 8D 11 91
4368 A9 2Ø 2C 1D 91 FØ FB 2Ø
437Ø 3A 43 58 8E 12 91 A9 ØC
4378 8D 11 91 AD 1Ø 91 28 BØ

438Ø Ø4 1Ø Ø2 81 FB C1 FB FØ
4388 Ø3 2Ø 68 41 2Ø 1F 49 DØ
439Ø B7 A9 4C 48 A9 77 48 Ø8
4398 48 48 48 6C 6Ø Ø3 8D 4B
43AØ Ø3 48 2Ø 8C 48 2Ø ØØ 49
43A8 DØ F8 68 49 FF 4C 72 42
43BØ 2Ø 2B 44 AE 56 Ø3 DØ ØD
43B8 2Ø FØ 4Ø 9Ø Ø8 2Ø C8 43
43CØ 2Ø E1 FF DØ EE 4C ØE 42
43C8 2Ø AE 45 A2 2E A9 3A 2Ø
43DØ ØE 48 2Ø 38 49 2Ø F8 47
43D8 A9 Ø8 2Ø EA 48 A9 Ø8 2Ø
43EØ AB 43 2Ø 38 49 2Ø 38 49
43E8 A9 12 2Ø D2 FF AØ Ø8 A2
```

```
43F0 00 A1 FB 29 7F C9 20 B0
43F8 02 A9 2E 20 D2 FF A9 00

4400 85 D4 EA EA EA EA EA EA
4408 EA EA EA EA 20 1F 49 88
4410 D0 DF 4C DF 4A 20 E6 47
4418 A9 08 20 9E 43 20 B6 45
4420 20 C8 43 A9 3A 8D 77 02
4428 4C 48 45 20 E6 47 85 FD
4430 86 FE 20 A4 49 F0 03 20
4438 EB 47 4C AE 45 20 31 48
4440 85 FD 86 FE A2 00 8E 66
4448 03 20 8C 48 C9 20 F0 F4
4450 9D 4F 03 E8 E0 03 D0 F1
4458 CA 30 14 BD 4F 03 38 E9
4460 3F A0 05 4A 6E 66 03 6E
4468 65 03 88 D0 F6 F0 E9 A2
4470 02 20 A4 49 F0 22 C9 3A
4478 F0 1E C9 20 F0 F3 20 90

4480 45 B0 0F 20 6C 48 A4 FB
4488 84 FC 85 FB A9 30 9D 65
4490 03 E8 9D 65 03 E8 D0 D9
4498 8E 54 03 A2 00 8E 56 03
44A0 A2 00 8E 4B 03 AD 56 03
44A8 20 87 42 AE 58 03 8E 55
44B0 03 AA BD 36 4F 20 70 45
44B8 BD F6 4E 20 70 45 A2 06
44C0 E0 03 D0 14 AC 4D 03 F0
44C8 0F AD 58 03 C9 E8 A9 30
44D0 B0 1E 20 6D 45 88 D0 F1
44D8 0E 58 03 90 0E BD E9 4E
44E0 20 70 45 BD EF 4E F0 03
44E8 20 70 45 CA D0 D2 F0 06
44F0 20 6D 45 20 6D 45 AD 54
44F8 03 CD 4B 03 D0 7F 20 21

4500 48 AC 4D 03 F0 2F AD 55
4508 03 C9 9D D0 20 20 F0 40
4510 90 01 88 C8 D0 6F 98 2A
```

```
4518 AE 53 Ø3 EØ 82 A8 DØ Ø3
452Ø BØ Ø3 38 BØ 6Ø CA CA 8A
4528 AC 4D Ø3 DØ Ø3 B9 FC ØØ
453Ø 91 FB 88 DØ F8 AD 56 Ø3
4538 91 FB AØ 41 8C 77 Ø2 2Ø
454Ø B6 45 2Ø C4 4Ø 2Ø 6F 42
4548 A9 2Ø 8D 78 Ø2 8D 7D Ø2
455Ø A5 FC 2Ø 9F 45 8E 79 Ø2
4558 8D 7A Ø2 A5 FB 2Ø 9F 45
456Ø 8E 7B Ø2 8D 7C Ø2 A9 Ø7
4568 85 C6 4C 68 4Ø 2Ø 7Ø 45
457Ø 8E 4A Ø3 AE 4B Ø3 DD 65
4578 Ø3 FØ ØD 68 68 EE 56 Ø3

458Ø FØ Ø3 4C AØ 44 4C 6Ø 4Ø
4588 E8 8E 4B Ø3 AE 4A Ø3 6Ø
459Ø C9 3Ø 9Ø Ø3 C9 47 6Ø 38
4598 6Ø CD 4E Ø3 DØ 1A 6Ø 48
45AØ 4A 4A 4A 4A 2Ø 17 48 AA
45A8 68 29 ØF 4C 17 48 A9 ØD
45BØ 2Ø D2 FF A9 ØA 2C A9 91
45B8 4C D2 FF 8D 3F Ø3 Ø8 68
45CØ 29 EF 8D 3E Ø3 8E 4Ø Ø3
45C8 8C 41 Ø3 68 18 69 Ø1 8D
45DØ 3D Ø3 68 69 ØØ 8D 3C Ø3
45D8 A9 8Ø 8D 48 Ø3 DØ 26 A9
45EØ CØ 8D 2E 91 A9 3F 8D 2E
45E8 91 2Ø 94 48 D8 68 8D 41
45FØ Ø3 68 8D 4Ø Ø3 68 8D 3F
45F8 Ø3 68 8D 3E Ø3 68 8D 3D

46ØØ Ø3 68 8D 3C Ø3 AD 14 Ø3
46Ø8 8D 44 Ø3 AD 15 Ø3 8D 43
461Ø Ø3 BA 8E 42 Ø3 58 AD 3E
4618 Ø3 29 1Ø FØ Ø3 4C 4E 4Ø
462Ø 2C 48 Ø3 5Ø 1F AD 3C Ø3
4628 CD 5B Ø3 DØ 6B AD 3D Ø3
463Ø CD 5A Ø3 DØ 63 AD 5E Ø3
4638 DØ 5B AD 5F Ø3 DØ 53 A9
464Ø 8Ø 8D 48 Ø3 3Ø 12 4E 48
```

```
4648 Ø3 9Ø D2 AE 42 Ø3 9A A9
465Ø 45 48 A9 BA 48 4C Ø6 47
4658 2Ø AE 45 2Ø 14 49 8D 4B
466Ø Ø3 AØ ØØ 2Ø F2 48 AD 3D
4668 Ø3 AE 3C Ø3 85 FB 86 FC
467Ø 2Ø 38 49 A9 24 8D 4E Ø3
4678 2Ø 16 42 2Ø E4 FF FØ FB

468Ø C9 Ø3 DØ Ø3 4C 68 4Ø C9
4688 4A DØ 4E A9 Ø1 8D 48 Ø3
469Ø DØ 47 CE 5F Ø3 CE 5E Ø3
4698 AD 21 91 C9 FE DØ 3A A2
46AØ 53 4C 5B 4Ø A9 ØØ FØ 12
46A8 AD 5C Ø3 AE 5D Ø3 8D 5E
46BØ Ø3 8E 5F Ø3 A9 4Ø DØ Ø2
46B8 A9 8Ø 8D 48 Ø3 2Ø A4 49
46CØ FØ ØF C9 2Ø DØ 56 2Ø 45
46C8 48 2Ø E3 48 2Ø A4 49 DØ
46DØ 4B 2Ø AE 45 AD 48 Ø3 FØ
46D8 1F 78 A9 AØ 8D 2E 91 A9
46EØ 5F 8D 2E 91 A9 DF A2 45
46E8 8D 44 Ø3 8E 43 Ø3 A9 49
46FØ A2 ØØ 8D 28 91 8E 29 91
46F8 AE 42 Ø3 9A 78 AD 44 Ø3

47ØØ AE 43 Ø3 2Ø 98 48 AD 3C
47Ø8 Ø3 48 AD 3D Ø3 48 AD 3E
471Ø Ø3 48 AD 3F Ø3 AE 4Ø Ø3
4718 AC 41 Ø3 4Ø 4C 6Ø 4Ø 2Ø
472Ø 31 48 8D 5A Ø3 8E 5B Ø3
4728 A9 ØØ 8D 5C Ø3 8D 5D Ø3
473Ø 2Ø 42 48 8D 5C Ø3 8E 5D
4738 Ø3 4C 68 4Ø 2Ø CB 47 8D
474Ø 62 Ø3 8E 63 Ø3 2Ø 42 48
4748 8D 4F Ø3 8E 5Ø Ø3 2Ø 42
475Ø 48 8D 51 Ø3 8E 52 Ø3 2Ø
4758 A4 49 FØ ØA 2Ø CF FF C9
476Ø 57 DØ Ø3 EE 4E Ø3 2Ø 21
4768 48 AE 56 Ø3 DØ 18 2Ø E7
477Ø 4Ø 9Ø 13 AC 4E Ø3 DØ 1A
4778 B1 FB 2Ø 87 42 AA BD F6
```

```
4780 4E DØ Ø6 2Ø C4 4Ø 4C 68
4788 4Ø AC 4D Ø3 CØ Ø2 DØ 33
4790 FØ Ø3 8C 4D Ø3 88 38 B1
4798 FB AA ED 4F Ø3 C8 B1 FB
47AØ ED 5Ø Ø3 9Ø 1E 88 AD 51
47A8 Ø3 F1 FB C8 AD 52 Ø3 F1
47BØ FB 9Ø 1Ø 88 18 8A 6D 62
47B8 Ø3 91 FB C8 B1 FB 6D 63
47CØ Ø3 91 FB 2Ø 1F 49 88 1Ø
47C8 FA 3Ø 9E 2Ø 31 48 85 FD
47DØ 86 FE 2Ø 42 48 8D 54 Ø3
47D8 8E 55 Ø3 2Ø 8C 48 2Ø 45
47EØ 48 85 FB 86 FC 6Ø 2Ø 31
47E8 48 BØ F6 2Ø 45 48 BØ Ø3
47FØ 2Ø 42 48 85 FD 86 FE 6Ø
47F8 A5 FC 2Ø FF 47 A5 FB 48

48ØØ 4A 4A 4A 4A 2Ø 17 48 AA
48Ø8 68 29 ØF 2Ø 17 48 48 8A
481Ø 2Ø D2 FF 68 4C D2 FF 18
4818 69 F6 9Ø Ø2 69 Ø6 69 3A
482Ø 6Ø A2 Ø2 B5 FA 48 B5 FC
4828 95 FA 68 95 FC CA DØ F3
483Ø 6Ø A9 ØØ 8D 59 Ø3 2Ø 8C
4838 48 C9 2Ø FØ F9 2Ø 6C 48
484Ø BØ Ø8 2Ø 8C 48 2Ø 57 48
4848 9Ø Ø7 AA 2Ø 57 48 9Ø Ø1
485Ø 6Ø 4C 6Ø 4Ø 2Ø 74 41 A9
4858 ØØ 8D 59 Ø3 2Ø 8C 48 C9
486Ø 2Ø DØ Ø9 2Ø 8C 48 C9 2Ø
4868 DØ ØF 18 6Ø 2Ø 81 48 ØA
487Ø ØA ØA ØA 8D 59 Ø3 2Ø 8C
4878 48 2Ø 81 48 ØD 59 Ø3 38

488Ø 6Ø C9 3A Ø8 29 ØF 28 9Ø
4888 Ø2 69 Ø8 6Ø 2Ø A4 49 DØ
489Ø FA 4C 65 4Ø A9 91 A2 43
4898 8D 14 Ø3 8E 15 Ø3 6Ø 2Ø
48AØ A4 49 FØ 37 2Ø E6 47 A5
48A8 FB Ø5 FC FØ 22 A5 9A C9
48BØ Ø3 DØ 9E A5 FB 8D 93 Ø2
```

```
48B8 A5 FC 8D 94 Ø2 A9 Ø2 AA
48CØ A8 2Ø BA FF 2Ø CØ FF A2
48C8 Ø2 2Ø C9 FF 4C 75 4Ø A9
48DØ Ø2 2Ø C3 FF A9 Ø3 85 9A
48D8 4C 68 4Ø A5 9A C9 Ø3 FØ
48EØ DC DØ F1 8D 3D Ø3 8E 3C
48E8 Ø3 6Ø 8D 4B Ø3 AØ ØØ 2Ø
48FØ 38 49 B1 FB 2Ø FF 47 2Ø
48F8 1F 49 CE 4B Ø3 DØ FØ 6Ø

49ØØ 2Ø 57 48 9Ø Ø8 A2 ØØ 81
49Ø8 FB C1 FB DØ 69 2Ø 1F 49
491Ø CE 4B Ø3 6Ø A9 3E 85 FB
4918 A9 Ø3 85 FC A9 Ø5 6Ø E6
492Ø FB DØ Ø9 E6 FF E6 FC DØ
4928 Ø3 EE 56 Ø3 6Ø 98 48 2Ø
493Ø AE 45 68 A2 2E 2Ø ØE 48
4938 A9 2Ø 4C D2 FF 2Ø ØE 48
494Ø A2 ØØ BD 76 4F 2Ø D2 FF
4948 E8 EØ 1C DØ F5 AØ 3B 2Ø
495Ø 2D 49 AD 3C Ø3 2Ø FF 47
4958 AD 3D Ø3 2Ø FF 47 2Ø 38
496Ø 49 AD 43 Ø3 2Ø FF 47 AD
4968 44 Ø3 2Ø FF 47 2Ø 14 49
497Ø 2Ø EA 48 4C 68 4Ø 4C 6Ø
4978 4Ø 2Ø 31 48 2Ø E3 48 2Ø

498Ø 42 48 8D 44 Ø3 8E 43 Ø3
4988 2Ø 14 49 8D 4B Ø3 2Ø 8C
499Ø 48 2Ø ØØ 49 DØ F8 FØ DB
4998 2Ø CF FF C9 2Ø FØ F9 DØ
49AØ Ø6 2Ø FØ 47 2Ø CF FF C9
49A8 ØD 6Ø AØ Ø1 84 BA A9 ØØ
49BØ A2 65 AØ Ø3 2Ø BD FF A8
49B8 2Ø E6 47 AD 49 Ø3 C9 53
49CØ DØ Ø8 2Ø A4 49 FØ AF 2Ø
49C8 EB 47 2Ø 98 49 FØ 29 C9
49DØ 22 DØ A3 2Ø CF FF C9 22
49D8 FØ ØB 91 BB E6 B7 C8 CØ
49EØ 51 9Ø FØ BØ 91 2Ø A4 49
```

```
49E8 FØ ØE 2Ø 57 48 29 1F FØ
49FØ 85 85 BA 2Ø 98 49 DØ D9
49F8 A9 ØØ 85 B9 AD 49 Ø3 C9

4AØØ 53 DØ ØC A9 FB A6 FD A4
4AØ8 FE 2Ø D8 FF 4C 68 4Ø 49
4A1Ø 4C FØ Ø2 A9 Ø1 A6 FB A4
4A18 FC 2Ø D5 FF A5 9Ø 29 1Ø
4A2Ø FØ EA A9 69 AØ C3 2Ø 1E
4A28 CB 4C 6Ø 4Ø 2Ø E6 47 2Ø
4A3Ø A5 4Ø 4C 68 4Ø 2Ø E6 47
4A38 2Ø 1F 49 2Ø 1F 49 2Ø FØ
4A4Ø 47 2Ø 38 49 2Ø FØ 4Ø 9Ø
4A48 ØA 98 DØ 15 AD 53 Ø3 3Ø
4A5Ø 1Ø 1Ø Ø8 C8 DØ ØB AD 53
4A58 Ø3 1Ø Ø6 2Ø FF 47 4C 68
4A6Ø 4Ø 4C 6Ø 4Ø 2Ø E6 47 2Ø
4A68 7A 4A 4C 68 4Ø 2Ø AE 45
4A7Ø A2 2E A9 24 2Ø ØE 48 2Ø
4A78 F8 47 2Ø EA 4A 2Ø AØ 4A

4A8Ø 2Ø 38 49 2Ø 86 4A 2Ø 89
4A88 4A 2Ø 38 49 A2 Ø4 A9 3Ø
4A9Ø 18 ØE 54 Ø3 2E 55 Ø3 69
4A98 ØØ 2Ø D2 FF CA DØ EF 6Ø
4AAØ A5 FC A6 FB 8D 55 Ø3 8E
4AA8 54 Ø3 2Ø 38 49 A5 FC 2Ø
4ABØ B4 4A A5 FB AA 2Ø 38 49
4AB8 8A 29 7F C9 2Ø Ø8 BØ ØA
4ACØ A9 12 2Ø D2 FF 8A 18 69
4AC8 4Ø AA 8A 2Ø D2 FF A9 ØØ
4ADØ 85 D4 EA EA EA EA EA EA
4AD8 EA EA EA EA 28 BØ CØ A9
4AEØ 92 2C A9 14 2C A9 22 4C
4AE8 D2 FF 2Ø 38 49 A6 FB A5
4AFØ FC 4C CD DD 2Ø Ø5 4B BØ
4AF8 41 2Ø 38 49 2Ø F8 47 2Ø

4BØØ 7D 4A 4C 68 4Ø A2 Ø4 A9
4BØ8 ØØ 85 FC 2Ø C2 4B 2Ø 2B
```

```
4B1Ø 4B 85 FB 2Ø 22 4B 2Ø 3D
4B18 4B CA DØ F7 Ø8 2Ø 38 49
4B2Ø 28 6Ø 2Ø A4 49 FØ ØF C9
4B28 2Ø FØ ØB C9 3Ø 9Ø ØB C9
4B3Ø 3A BØ Ø7 29 ØF 6Ø 68 68
4B38 18 6Ø 4C 6Ø 4Ø 85 FE A5
4B4Ø FC 48 A5 FB 48 Ø6 FB 26
4B48 FC Ø6 FB 26 FC 68 65 FB
4B5Ø 85 FB 68 65 FC 85 FC Ø6
4B58 FB 26 FC A5 FE 65 FB 85
4B6Ø FB A9 ØØ 65 FC 85 FC 6Ø
4B68 2Ø C2 4B 8D 55 Ø3 48 48
4B7Ø 2Ø 38 49 2Ø 38 49 68 2Ø
4B78 FF 47 2Ø 38 49 68 AA A9

4B8Ø ØØ 2Ø F1 4A 2Ø 38 49 2Ø
4B88 86 4A 4C 68 4Ø 2Ø 9F 4B
4B9Ø 2Ø 38 49 2Ø F8 47 2Ø EA
4B98 4A 2Ø AØ 4A 4C 68 4Ø A2
4BAØ ØF A9 ØØ 85 FB 85 FC 2Ø
4BA8 C2 4B 2Ø 2B 4B 2Ø BC 4B
4BBØ 2Ø 22 4B 2Ø BC 4B CA DØ
4BB8 F7 4C 38 49 4A 26 FB 26
4BCØ FC 6Ø 2Ø 8C 48 C9 2Ø FØ
4BC8 F9 6Ø 2Ø 54 48 8D 88 Ø2
4BDØ A6 FB A4 FC 2Ø 8A FE A6
4BD8 FD A4 FE 2Ø 7B FE 2Ø 18
4BEØ E5 2Ø A4 E3 4C 68 4Ø 2Ø
4BE8 FØ 47 4C DB 47 2Ø E7 4B
4BFØ 18 A5 FB 65 FD 85 FB A5
4BF8 FC 65 FE 85 FC 4C ØD 4C

4CØØ 2Ø E7 4B 2Ø FØ 4Ø 84 FC
4CØ8 AD 53 Ø3 85 FB 2Ø 38 49
4C1Ø 2Ø F8 47 4C 68 4Ø A9 FØ
4C18 2C A9 ØØ 8D ØB 9Ø 4C 65
4C2Ø 4Ø 78 2Ø 52 FD 58 A9 3C
4C28 85 B2 AE 42 Ø3 9A A5 73
4C3Ø C9 E6 FØ 95 6C ØØ CØ 2Ø
4C38 E7 4B 2Ø 21 48 2Ø 38 49
4C4Ø AØ ØØ 8C 54 Ø3 8C 55 Ø3
```

```
4C48 20 F0 40 90 1B AC 56 03
4C50 D0 16 18 B1 FB 6D 54 03
4C58 8D 54 03 98 6D 55 03 8D
4C60 55 03 20 1F 49 4C 48 4C
4C68 AD 55 03 20 FF 47 AD 54
4C70 03 20 FF 47 4C 68 40 AD
4C78 64 03 D0 04 A5 C6 D0 03

4C80 4C 56 FF AD 77 02 C9 11
4C88 D0 7D A5 D6 C9 16 D0 F0
4C90 A5 D1 85 FD A5 D2 85 FE
4C98 A9 17 8D 5E 03 A0 01 20
4CA0 51 4E C9 3A F0 1A C9 2C
4CA8 F0 16 C9 24 F0 12 CE 5E
4CB0 03 F0 CD 38 A5 FD E9 16
4CB8 85 FD B0 E1 C6 FE D0 DD
4CC0 8D 49 03 20 0A 4E B0 B8
4CC8 AD 49 03 C9 3A D0 11 18
4CD0 A5 FB 69 08 85 FB 90 02
4CD8 E6 FC 20 C8 43 4C F4 4C
4CE0 C9 24 F0 1A 20 C9 4D 20
4CE8 6F 42 A9 00 8D 4E 03 A0
4CF0 2C 20 13 42 A9 00 85 C6
4CF8 4C 0E 42 4C 56 FF 20 1F

4D00 49 20 6D 4A 4C F4 4C C9
4D08 91 D0 F0 A5 D6 D0 EC A5
4D10 D1 85 FD A5 D2 85 FE A9
4D18 17 8D 5E 03 A0 01 20 51
4D20 4E C9 3A F0 1A C9 2C F0
4D28 16 C9 24 F0 12 CE 5E 03
4D30 F0 15 18 A5 FD 69 16 85
4D38 FD 90 E1 E6 FE D0 DD 8D
4D40 49 03 20 0A 4E 90 03 4C
4D48 56 FF AD 49 03 C9 3A F0
4D50 06 C9 24 F0 1D D0 27 20
4D58 D0 4D 38 A5 FB E9 08 85
4D60 FB B0 02 C6 FC 20 CB 43
4D68 A9 00 85 C6 20 05 4E 4C
4D70 70 40 20 D0 4D 20 B2 40
4D78 20 70 4A 4C 68 4D 20 D0
```

```
4D8Ø 4D A5 FB A6 FC 85 FD 86
4D88 FE A9 1Ø 8D 5E Ø3 38 A5
4D9Ø FD ED 5E Ø3 85 FB A5 FE
4D98 E9 ØØ 85 FC 2Ø C9 4D 2Ø
4DAØ 6F 42 2Ø FØ 4Ø FØ Ø7 BØ
4DA8 F3 CE 5E Ø3 DØ EØ EE 4D
4DBØ Ø3 AD 4D Ø3 2Ø AB 43 A2
4DB8 ØØ A1 FB 8E 4E Ø3 A9 2C
4DCØ 2Ø 33 49 2Ø 16 42 4C 68
4DC8 4D A2 ØØ A1 FB 4C 87 42
4DDØ A6 D2 2Ø D7 4D A6 F4 E8
4DD8 86 AD 86 FE A2 ØØ 86 AC
4DEØ A9 2C 85 FD AØ CE E8 88
4DE8 B1 AC 91 FD 98 DØ F8 C6
4DFØ AD C6 FE CA 1Ø F1 A9 2Ø
4DF8 A6 D2 86 FE 84 FD AØ 2B

4EØØ 91 FD 88 1Ø FB A9 13 4C
4EØ8 D2 FF CØ 16 DØ Ø2 38 6Ø
4E1Ø 2Ø 51 4E C9 2Ø FØ F3 88
4E18 2Ø 3A 4E AA 2Ø 3A 4E 85
4E2Ø FB 86 FC A9 FF 8D 64 Ø3
4E28 85 CC A5 CF FØ ØA A5 CE
4E3Ø A4 D3 91 D1 A9 ØØ 85 CF
4E38 18 6Ø 2Ø 51 4E 2Ø 81 48
4E4Ø ØA ØA ØA ØA 8D 59 Ø3 2Ø
4E48 51 4E 2Ø 81 48 ØD 59 Ø3
4E5Ø 6Ø B1 FD C8 29 7F C9 2Ø
4E58 BØ Ø2 Ø9 4Ø 6Ø BD 98 4D
4E6Ø 2Ø D2 FF E8 DØ F7 6Ø ØØ
4E68 ØØ ØØ ØØ ØØ ØØ ØØ ØD 56
4E7Ø 49 43 32 3Ø 2Ø 4D 49 43
4E78 52 4F 4D 4F 4E 2Ø 56 31

4E8Ø 2E 32 2Ø 2Ø 2Ø 42 49 4C
4E88 4C 2Ø 59 45 45 2Ø 32 32
4E9Ø 2Ø 4A 41 4E 2Ø 2Ø 38 33
4E98 4Ø Ø2 45 Ø3 DØ Ø8 4Ø Ø9
4EAØ 3Ø 22 45 33 DØ Ø8 4Ø Ø9
4EA8 4Ø Ø2 45 33 DØ Ø8 4Ø Ø9
4EBØ 4Ø Ø2 45 B3 DØ Ø8 4Ø Ø9
```

```
4EB8 ØØ 22 44 33 DØ 8C 44 ØØ
4ECØ 11 22 44 33 DØ 8C 44 9A
4EC8 1Ø 22 44 33 DØ Ø8 4Ø Ø9
4EDØ 1Ø 22 44 33 DØ Ø8 4Ø Ø9
4ED8 62 13 78 A9 ØØ 21 81 82
4EEØ ØØ ØØ 59 4D 91 92 86 4A
4EE8 85 9D 2C 29 2C 23 28 24
4EFØ 59 ØØ 58 24 24 ØØ 1C 8A
4EF8 1C 23 5D 8B 1B A1 9D 8A
```

```
4FØØ 1D 23 9D 8B 1D A1 ØØ 29
4FØ8 19 AE 69 A8 19 23 24 53
4F1Ø 1B 23 24 53 19 A1 ØØ 1A
4F18 5B 5B A5 69 24 24 AE AE
4F2Ø A8 AD 29 ØØ 7C ØØ 15 9C
4F28 6D 9C A5 69 29 53 84 13
4F3Ø 34 11 A5 69 23 AØ D8 62
4F38 5A 48 26 62 94 88 54 44
4F4Ø C8 54 68 44 E8 94 ØØ B4
4F48 Ø8 84 74 B4 28 6E 74 F4
4F5Ø CC 4A 72 F2 A4 8A ØØ AA
4F58 A2 A2 74 74 74 72 44 68
4F6Ø B2 32 B2 ØØ 22 ØØ 1A 1A
4F68 26 26 72 72 88 C8 C4 CA
4F7Ø 26 48 44 44 A2 C8 ØD 2Ø
4F78 2Ø 2Ø 2Ø 5Ø 43 2Ø 2Ø 49
```

```
4F8Ø 52 51 2Ø 2Ø 53 52 2Ø 41
4F88 43 2Ø 58 52 2Ø 59 52 2Ø
4F9Ø 53 5Ø 41 42 43 44 46 47
4F98 48 4C 4D 4E 51 52 28 54
4FAØ 57 58 2C 3A 3B 24 23 22
4FA8 2B 2D 4F 49 4A 25 26 45
4FBØ 56 29 3D 5C FF AA 49 9F
4FB8 48 3D 44 1F 47 Ø2 41 F9
4FCØ 41 87 41 A4 46 AØ 41 AA
```

```
4FC8 49 BØ 43 3C 47 A8 46 4Ø
4FDØ 49 16 4C Ø6 41 B8 46 2A
4FD8 4C ØC 43 15 44 79 49 64
4FEØ 4A F4 4A 68 4B ED 4B ØØ
4FE8 4C 35 4A CA 4B 2C 4A 8D
4FFØ 4B 37 4C 21 4C AA 49 19
4FF8 4C 4Ø 43 4Ø 43 4Ø 43 49
```

Supermon64

Supermon64 is your gateway to machine language programming on the Commodore 64. Supermon, in several versions, has been popular over the years as a major programming tool for Commodore users. Supermon64 itself is in machine language, but you can type it in without knowing what it means. Using the Tiny PEEKer/POKEr (Program 1), or via the built-in monitor of a PET, type it in and SAVE it. The fastest way to check for errors is to type in Program 3 on a regular PET. Then load Supermon64 into the PET. It will come in above your BASIC. Then RUN the checksum and it will report the location of any errors. Type POKE 8192,0 and hit RETURN. Then type POKE 44,32 followed by NEW.

Enter the following:

Program 1. Tiny PEEKer/POKEr.

```
100 PRINT "TINY PEEKER/POKER"
110 X$="*":INPUT X$:IF X$="*" THEN END
120 GOSUB 500
130 IF E GOTO 280
140 A=V
150 IF J>LEN(X$) GOTO 300
160 FOR I=0 TO 7
170 P=J:GOSUB 550
180 C(I)=V
190 IF E GOTO 280
200 NEXT I
210 T=0
220 FOR I=0 TO 7
230 POKE A+I,C(I)
240 T=T+C(I)
250 NEXT I
260 PRINT "CHECKSUM=";T
270 GOTO 110
280 PRINT MID$(X$,1,J);"??":GOTO 110
300 T=0
310 FOR I=0 TO 7
320 V=PEEK(A+I)
330 T=T+V
340 V=V/16
350 PRINT " ";
360 FOR J=1 TO 2
370 V%=V
```

```
380 V=(V-V%)*16
390 IF V%>9 THEN V%=V%+7
400 PRINT CHR$(V%+48);
410 NEXT J
420 NEXT I
430 PRINT "/";T
440 GOTO 110
500 P=1
510 L=4
520 GOTO 600
550 P=J
560 L=2
600 E=0
610 V=0
620 FOR J=P TO LEN(X$)
630 X=ASC(MID$(X$,J))
640 IF X=32 THEN NEXT J
650 IF J>LEN(X$) GOTO 790
660 P=J
670 FOR J=P TO LEN(X$)
680 X=ASC(MID$(X$,J))
690 IF X<>32 THEN NEXT J
700 IF J-P<>L GOTO 790
710 FOR K=P TO J-1
720 X=ASC(MID$(X$,K))
730 IF X<58 THEN X=X-48
740 IF X>64 THEN X=X-55
750 IF X<0 OR X>15 GOTO 790
760 V=V*16+X
770 NEXT K
780 RETURN
790 E=-1
800 RETURN
```

This program is a very tiny monitor. It will allow you to enter information into memory, eight bytes at a time. To do this: wait for the question mark, and then type in monitor-format the address and contents:

?0800 00 1A 08 64 00 99 22 93

The program will return a checksum value to you, which you can use to insure that you have entered the information correctly. To view memory, type in only the address: the contents will be displayed.

Completing The Job
When you have finished entering all that data, you can make Supermon64 happen quite easily. Three last POKE commands and a CLR:

```
POKE 44,8
POKE 45, 235
POKE 46,17
CLR
```

You have Supermon64. Save it with a conventional BASIC SAVE before you do anything else.

Now you may RUN it — and learn how to use it.

NOTE: Before entering the hex numbers with Tiny PEEKer/POKEr, type in the memory partitioning POKES: POKE 8192,0 and POKE 44,32, and then type NEW. When you've finished entering all the hex numbers, type: POKE 44,8: POKE 46,17: CLR. You can then SAVE Supermon64 in the ordinary, BASIC way, to tape or disk. It's ready now to LOAD or RUN. Note also that the checksum program on page 333 checks 129 bytes at a time. This can have the effect of attributing a typing error to the wrong block if the error occurs near the beginning or the end of a block.

- **Simple assembler**

```
.A 2000 LDA #$12
.A 2002 STA $8000,X
.A 2005 (RETURN)
```

In the above example the user started assembly at 2000 hex. The first instruction was load a register with immediate 12 hex. In the second line the user did not need to type the A and address. The simple assembler prompts with the next address. To exit the assembler type a return after the address prompt. Syntax is the same as the disassembler output.

- **Disassembler**

```
.D 2000
(SCREEN CLEARS)
2000 A9 12       LDA #$12
```

```
2ØØ2 9D ØØ 8Ø     STA $8ØØØ,X
2ØØ5 AA           TAX
2ØØ6 AA           TAX
```

(Full page of instructions)

Disassembles 22 instructions starting at 2000 hex. The three bytes following the address may be modified. Use the CRSR keys to move to and modify the bytes. Hit return and the bytes in memory will be changed. Supermon64 will then disassemble that page again.

- **Printing disassembler**

```
.P 2ØØØ,2Ø4Ø
2ØØØ A9 12          LDA #$12
2ØØ2 9D ØØ 8Ø       STA $8ØØØ,X
2ØØ5 AA             TAX
     ....
2Ø3F A2 ØØ          LDX #$ØØ
```

To engage printer, set up beforehand:

OPEN 4,4:CMD4

- **Fill memory**

```
.F 1ØØØ 11ØØ FF
```

Fills the memory from 1000 hex to 1100 hex with the byte FF hex.

- **Go run**

```
.G
```

Go to the address in the PC register display and begin RUN code. All the registers will be replaced with the displayed values.

```
.G 1ØØØ
```

Go to address 1000 hex and begin running code.

- **Hunt memory**

```
.H CØØØ DØØØ 'READ
```

Hunt through memory from C000 hex to D000 hex for the ASCII string read and print the address where it is found. A maximum of 32 characters may be used.

```
.H C000 D000 20 D2 FF
```

Hunt through memory from C000 hex to D000 hex for the sequence of bytes 20 D2 FF and print the address. A maximum of 32 bytes may be used.

• Load

```
.L
```

Load any program from cassette #1.

```
.L "RAM TEST"
```

Load from cassette #1 the program named RAM TEST.

```
.L "RAM TEST",08
```

Load from disk (device 8) the program named RAM TEST. This command leaves BASIC pointers unchanged.

• Memory display

```
.M 0000 0080
.: 0000 00 01 02 03 04 05 06 07
.: 0008 08 09 0A 0B 0C 0D 0E 0F
```

Display memory from 0000 hex to 0080 hex. The bytes following the .: can be altered by typing over them, then typing a return.

• Register display

```
.R

     PC   IRQ  SR AC XR YR SP
    0000 E62E 01 02 03 04 05
```

Displays the register values saved when Supermon64 was entered. The values may be changed with the edit followed by a return.

• Save

```
.S "PROGRAM NAME",01,0800,0C80
```

SAVE to cassette #1 memory from 0800 hex up to but not including 0C80 hex and name it PROGRAM NAME.

```
.S "Ø:PROGRAM NAME",Ø8,12ØØ,1F5Ø
```

SAVE to disk drive #0 memory from 1200 hex up to but not including 1F50 hex and name it PROGRAM NAME.

• **Transfer memory**

```
.T 1ØØØ 11ØØ 5ØØØ
```

Transfer memory in the range 1000 hex to 1100 hex and start storing it at address 5000 hex.

• **Exit to BASIC**

```
.X
```

Return to BASIC ready mode. The stack value SAVEd when entered will be restored. Care should be taken that this value is the same as when the monitor was entered. A CLR in BASIC will fix any stack problems.

Program 2. Supermon64.

```
Ø8ØØ ØØ 1A Ø4 64 ØØ 99 22 93
Ø8Ø8 12 1D 1D 1D 1D 53 55 5Ø
Ø81Ø 45 52 2Ø 36 34 2D 4D 4F
Ø818 4E ØØ 31 Ø4 6E ØØ 99 22
Ø82Ø 11 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
Ø828 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
Ø83Ø ØØ 4B Ø4 78 ØØ 99 22 11
Ø838 2Ø 2E 2E 4A 49 4D 2Ø 42
Ø84Ø 55 54 54 45 52 46 49 45
Ø848 4C 44 ØØ 66 Ø4 82 ØØ 9E
Ø85Ø 28 C2 28 34 33 29 AA 32
Ø858 35 36 AC C2 28 34 34 29
Ø86Ø AA 31 32 37 29 ØØ ØØ ØØ
Ø868 AA AA AA AA AA AA AA AA
Ø87Ø AA AA AA AA AA AA AA AA
Ø878 AA AA AA AA AA AA AA AA
```

```
Ø88Ø A5 2D 85 22 A5 2E 85 23
Ø888 A5 37 85 24 A5 38 85 25
Ø89Ø AØ ØØ A5 22 DØ Ø2 C6 23
Ø898 C6 22 B1 22 DØ 3C A5 22
Ø8AØ DØ Ø2 C6 23 C6 22 B1 22
Ø8A8 FØ 21 85 26 A5 22 DØ Ø2
Ø8BØ C6 23 C6 22 B1 22 18 65
Ø8B8 24 AA A5 26 65 25 48 A5
Ø8CØ 37 DØ Ø2 C6 38 C6 37 68
Ø8C8 91 37 8A 48 A5 37 DØ Ø2
Ø8DØ C6 38 C6 37 68 91 37 18
Ø8D8 9Ø B6 C9 4F DØ ED A5 37
Ø8EØ 85 33 A5 38 85 34 6C 37
Ø8E8 ØØ 4F 4F 4F 4F AD E6 FF
Ø8FØ ØØ 8D 16 Ø3 AD E7 FF ØØ
Ø8F8 8D 17 Ø3 A9 8Ø 2Ø 9Ø FF
```

```
Ø9ØØ ØØ ØØ D8 68 8D 3E Ø2 68
Ø9Ø8 8D 3D Ø2 68 8D 3C Ø2 68
Ø91Ø 8D 3B Ø2 68 AA 68 A8 38
Ø918 8A E9 Ø2 8D 3A Ø2 98 E9
Ø92Ø ØØ ØØ 8D 39 Ø2 BA 8E 3F
Ø928 Ø2 2Ø 57 FD ØØ A2 42 A9
Ø93Ø 2A 2Ø 57 FA ØØ A9 52 DØ
Ø938 34 E6 C1 DØ Ø6 E6 C2 DØ
Ø94Ø Ø2 E6 26 6Ø 2Ø CF FF C9
Ø948 ØD DØ F8 68 68 A9 9Ø 2Ø
Ø95Ø D2 FF A9 ØØ ØØ 85 26 A2
Ø958 ØD A9 2E 2Ø 57 FA ØØ A9
Ø96Ø Ø5 2Ø D2 FF 2Ø 3E F8 ØØ
Ø968 C9 2E FØ F9 C9 2Ø FØ F5
Ø97Ø A2 ØE DD B7 FF ØØ DØ ØC
Ø978 8A ØA AA BD C7 FF ØØ 48
```

```
Ø98Ø  BD C6 FF ØØ 48 6Ø CA 1Ø
Ø988  EC 4C ED FA ØØ A5 C1 8D
Ø99Ø  3A Ø2 A5 C2 8D 39 Ø2 6Ø
Ø998  A9 Ø8 85 1D AØ ØØ ØØ 2Ø
Ø9AØ  54 FD ØØ B1 C1 2Ø 48 FA
Ø9A8  ØØ 2Ø 33 F8 ØØ C6 1D DØ
Ø9BØ  F1 6Ø 2Ø 88 FA ØØ 9Ø ØB
Ø9B8  A2 ØØ ØØ 81 C1 C1 C1 FØ
Ø9CØ  Ø3 4C ED FA ØØ 2Ø 33 F8
Ø9C8  ØØ C6 1D 6Ø A9 3B 85 C1
Ø9DØ  A9 Ø2 85 C2 A9 Ø5 6Ø 98
Ø9D8  48 2Ø 57 FD ØØ 68 A2 2E
Ø9EØ  4C 57 FA ØØ A9 9Ø 2Ø D2
Ø9E8  FF A2 ØØ ØØ BD EA FF ØØ
Ø9FØ  2Ø D2 FF E8 EØ 16 DØ F5
Ø9F8  AØ 3B 2Ø C2 F8 ØØ AD 39
```

```
ØAØØ  Ø2 2Ø 48 FA ØØ AD 3A Ø2
ØAØ8  2Ø 48 FA ØØ 2Ø B7 F8 ØØ
ØA1Ø  2Ø 8D F8 ØØ FØ 5C 2Ø 3E
ØA18  F8 ØØ 2Ø 79 FA ØØ 9Ø 33
ØA2Ø  2Ø 69 FA ØØ 2Ø 3E F8 ØØ
ØA28  2Ø 79 FA ØØ 9Ø 28 2Ø 69
ØA3Ø  FA ØØ A9 9Ø 2Ø D2 FF 2Ø
ØA38  E1 FF FØ 3C A6 26 DØ 38
ØA4Ø  A5 C3 C5 C1 A5 C4 E5 C2
ØA48  9Ø 2E AØ 3A 2Ø C2 F8 ØØ
ØA5Ø  2Ø 41 FA ØØ 2Ø 8B F8 ØØ
ØA58  FØ EØ 4C ED FA ØØ 2Ø 79
ØA6Ø  FA ØØ 9Ø Ø3 2Ø 8Ø F8 ØØ
ØA68  2Ø B7 F8 ØØ DØ Ø7 2Ø 79
ØA7Ø  FA ØØ 9Ø EB A9 Ø8 85 1D
ØA78  2Ø 3E F8 ØØ 2Ø A1 F8 ØØ
```

```
ØA8Ø  DØ F8 4C 47 F8 ØØ 2Ø CF
ØA88  FF C9 ØD FØ ØC C9 2Ø DØ
ØA9Ø  D1 2Ø 79 FA ØØ 9Ø Ø3 2Ø
```

```
0A98 80 F8 00 A9 90 20 D2 FF
0AA0 AE 3F 02 9A 78 AD 39 02
0AA8 48 AD 3A 02 48 AD 3B 02
0AB0 48 AD 3C 02 AE 3D 02 AC
0AB8 3E 02 40 A9 90 20 D2 FF
0AC0 AE 3F 02 9A 6C 02 A0 A0
0AC8 01 84 BA 84 B9 88 84 B7
0AD0 84 90 84 93 A9 40 85 BB
0AD8 A9 02 85 BC 20 CF FF C9
0AE0 20 F0 F9 C9 0D F0 38 C9
0AE8 22 D0 14 20 CF FF C9 22
0AF0 F0 10 C9 0D F0 29 91 BB
0AF8 E6 B7 C8 C0 10 D0 EC 4C

0B00 ED FA 00 20 CF FF C9 0D
0B08 F0 16 C9 2C D0 DC 20 88
0B10 FA 00 29 0F F0 E9 C9 03
0B18 F0 E5 85 BA 20 CF FF C9
0B20 0D 60 6C 30 03 6C 32 03
0B28 20 96 F9 00 D0 D4 A9 90
0B30 20 D2 FF A9 00 00 20 EF
0B38 F9 00 A5 90 29 10 D0 C4
0B40 4C 47 F8 00 20 96 F9 00
0B48 C9 2C D0 BA 20 79 FA 00
0B50 20 69 FA 00 20 CF FF C9
0B58 2C D0 AD 20 79 FA 00 A5
0B60 C1 85 AE A5 C2 85 AF 20
0B68 69 FA 00 20 CF FF C9 0D
0B70 D0 98 A9 90 20 D2 FF 20
0B78 F2 F9 00 4C 47 F8 00 A5

0B80 C2 20 48 FA 00 A5 C1 48
0B88 4A 4A 4A 4A 20 60 FA 00
0B90 AA 68 29 0F 20 60 FA 00
0B98 48 8A 20 D2 FF 68 4C D2
0BA0 FF 09 30 C9 3A 90 02 69
0BA8 06 60 A2 02 B5 C0 48 B5
```

```
ØBBØ C2 95 CØ 68 95 C2 CA DØ
ØBB8 F3 6Ø 2Ø 88 FA ØØ 9Ø Ø2
ØBCØ 85 C2 2Ø 88 FA ØØ 9Ø Ø2
ØBC8 85 C1 6Ø A9 ØØ ØØ 85 2A
ØBDØ 2Ø 3E F8 ØØ C9 2Ø DØ Ø9
ØBD8 2Ø 3E F8 ØØ C9 2Ø DØ ØE
ØBEØ 18 6Ø 2Ø AF FA ØØ ØA ØA
ØBE8 ØA ØA 85 2A 2Ø 3E F8 ØØ
ØBFØ 2Ø AF FA ØØ Ø5 2A 38 6Ø
ØBF8 C9 3A 9Ø Ø2 69 Ø8 29 ØF

ØCØØ 6Ø A2 Ø2 2C A2 ØØ ØØ B4
ØCØ8 C1 DØ Ø8 B4 C2 DØ Ø2 E6
ØC1Ø 26 D6 C2 D6 C1 6Ø 2Ø 3E
ØC18 F8 ØØ C9 2Ø FØ F9 6Ø A9
ØC2Ø ØØ ØØ 8D ØØ ØØ Ø1 2Ø CC
ØC28 FA ØØ 2Ø 8F FA ØØ 2Ø 7C
ØC3Ø FA ØØ 9Ø Ø9 6Ø 2Ø 3E F8
ØC38 ØØ 2Ø 79 FA ØØ BØ DE AE
ØC4Ø 3F Ø2 9A A9 9Ø 2Ø D2 FF
ØC48 A9 3F 2Ø D2 FF 4C 47 F8
ØC5Ø ØØ 2Ø 54 FD ØØ CA DØ FA
ØC58 6Ø E6 C3 DØ Ø2 E6 C4 6Ø
ØC6Ø A2 Ø2 B5 CØ 48 B5 27 95
ØC68 CØ 68 95 27 CA DØ F3 6Ø
ØC7Ø A5 C3 A4 C4 38 E9 Ø2 BØ
ØC78 ØE 88 9Ø ØB A5 28 A4 29

ØC8Ø 4C 33 FB ØØ A5 C3 A4 C4
ØC88 38 E5 C1 85 1E 98 E5 C2
ØC9Ø A8 Ø5 1E 6Ø 2Ø D4 FA ØØ
ØC98 2Ø 69 FA ØØ 2Ø E5 FA ØØ
ØCAØ 2Ø ØC FB ØØ 2Ø E5 FA ØØ
ØCA8 2Ø 2F FB ØØ 2Ø 69 FA ØØ
ØCBØ 9Ø 15 A6 26 DØ 64 2Ø 28
ØCB8 FB ØØ 9Ø 5F A1 C1 81 C3
ØCCØ 2Ø Ø5 FB ØØ 2Ø 33 F8 ØØ
ØCC8 DØ EB 2Ø 28 FB ØØ 18 A5
ØCDØ 1E 65 C3 85 C3 98 65 C4
ØCD8 85 C4 2Ø ØC FB ØØ A6 26
ØCEØ DØ 3D A1 C1 81 C3 2Ø 28
```

```
ØCE8 FB ØØ BØ 34 2Ø B8 FA ØØ
ØCFØ 2Ø BB FA ØØ 4C 7D FB ØØ
ØCF8 2Ø D4 FA ØØ 2Ø 69 FA ØØ

ØDØØ 2Ø E5 FA ØØ 2Ø 69 FA ØØ
ØDØ8 2Ø 3E F8 ØØ 2Ø 88 FA ØØ
ØD1Ø 9Ø 14 85 1D A6 26 DØ 11
ØD18 2Ø 2F FB ØØ 9Ø ØC A5 1D
ØD2Ø 81 C1 2Ø 33 F8 ØØ DØ EE
ØD28 4C ED FA ØØ 4C 47 F8 ØØ
ØD3Ø 2Ø D4 FA ØØ 2Ø 69 FA ØØ
ØD38 2Ø E5 FA ØØ 2Ø 69 FA ØØ
ØD4Ø 2Ø 3E F8 ØØ A2 ØØ ØØ 2Ø
ØD48 3E F8 ØØ C9 27 DØ 14 2Ø
ØD5Ø 3E F8 ØØ 9D 1Ø Ø2 E8 2Ø
ØD58 CF FF C9 ØD FØ 22 EØ 2Ø
ØD6Ø DØ F1 FØ 1C 8E ØØ ØØ Ø1
ØD68 2Ø 8F FA ØØ 9Ø C6 9D 1Ø
ØD7Ø Ø2 E8 2Ø CF FF C9 ØD FØ
ØD78 Ø9 2Ø 88 FA ØØ 9Ø B6 EØ

ØD8Ø 2Ø DØ EC 86 1C A9 9Ø 2Ø
ØD88 D2 FF 2Ø 57 FD ØØ A2 ØØ
ØD9Ø ØØ AØ ØØ ØØ B1 C1 DD 1Ø
ØD98 Ø2 DØ ØC C8 E8 E4 1C DØ
ØDAØ F3 2Ø 41 FA ØØ 2Ø 54 FD
ØDA8 ØØ 2Ø 33 F8 ØØ A6 26 DØ
ØDBØ 8D 2Ø 2F FB ØØ BØ DD 4C
ØDB8 47 F8 ØØ 2Ø D4 FA ØØ 85
ØDCØ 2Ø A5 C2 85 21 A2 ØØ ØØ
ØDC8 86 28 A9 93 2Ø D2 FF A9
ØDDØ 9Ø 2Ø D2 FF A9 16 85 1D
ØDD8 2Ø 6A FC ØØ 2Ø CA FC ØØ
ØDEØ 85 C1 84 C2 C6 1D DØ F2
ØDE8 A9 91 2Ø D2 FF 4C 47 F8
ØDFØ ØØ AØ 2C 2Ø C2 F8 ØØ 2Ø
ØDF8 54 FD ØØ 2Ø 41 FA ØØ 2Ø

ØEØØ 54 FD ØØ A2 ØØ ØØ A1 C1
ØEØ8 2Ø D9 FC ØØ 48 2Ø 1F FD
```

```
ØE1Ø ØØ 68 2Ø 35 FD ØØ A2 Ø6
ØE18 EØ Ø3 DØ 12 A4 1F FØ ØE
ØE2Ø A5 2A C9 E8 B1 C1 BØ 1C
ØE28 2Ø C2 FC ØØ 88 DØ F2 Ø6
ØE3Ø 2A 9Ø ØE BD 2A FF ØØ 2Ø
ØE38 A5 FD ØØ BD 3Ø FF ØØ FØ
ØE4Ø Ø3 2Ø A5 FD ØØ CA DØ D5
ØE48 6Ø 2Ø CD FC ØØ AA E8 DØ
ØE5Ø Ø1 C8 98 2Ø C2 FC ØØ 8A
ØE58 86 1C 2Ø 48 FA ØØ A6 1C
ØE6Ø 6Ø A5 1F 38 A4 C2 AA 1Ø
ØE68 Ø1 88 65 C1 9Ø Ø1 C8 6Ø
ØE7Ø A8 4A 9Ø ØB 4A BØ 17 C9
ØE78 22 FØ 13 29 Ø7 Ø9 8Ø 4A

ØE8Ø AA BD D9 FE ØØ BØ Ø4 4A
ØE88 4A 4A 4A 29 ØF DØ Ø4 AØ
ØE9Ø 8Ø A9 ØØ ØØ AA BD 1D FF
ØE98 ØØ 85 2A 29 Ø3 85 1F 98
ØEAØ 29 8F AA 98 AØ Ø3 EØ 8A
ØEA8 FØ ØB 4A 9Ø Ø8 4A 4A Ø9
ØEBØ 2Ø 88 DØ FA C8 88 DØ F2
ØEB8 6Ø B1 C1 2Ø C2 FC ØØ A2
ØECØ Ø1 2Ø FE FA ØØ C4 1F C8
ØEC8 9Ø F1 A2 Ø3 CØ Ø4 9Ø F2
ØEDØ 6Ø A8 B9 37 FF ØØ 85 28
ØED8 B9 77 FF ØØ 85 29 A9 ØØ
ØEEØ ØØ AØ Ø5 Ø6 29 26 28 2A
ØEE8 88 DØ F8 69 3F 2Ø D2 FF
ØEFØ CA DØ EC A9 2Ø 2C A9 ØD
ØEF8 4C D2 FF 2Ø D4 FA ØØ 2Ø

ØFØØ 69 FA ØØ 2Ø E5 FA ØØ 2Ø
ØFØ8 69 FA ØØ A2 ØØ ØØ 86 28
ØF1Ø A9 9Ø 2Ø D2 FF 2Ø 57 FD
ØF18 ØØ 2Ø 72 FC ØØ 2Ø CA FC
ØF2Ø ØØ 85 C1 84 C2 2Ø E1 FF
ØF28 FØ Ø5 2Ø 2F FB ØØ BØ E9
ØF3Ø 4C 47 F8 ØØ 2Ø D4 FA ØØ
ØF38 A9 Ø3 85 1D 2Ø 3E F8 ØØ
```

```
ØF4Ø 2Ø A1 F8 ØØ DØ F8 A5 2Ø
ØF48 85 C1 A5 21 85 C2 4C 46
ØF5Ø FC ØØ C5 28 FØ Ø3 2Ø D2
ØF58 FF 6Ø 2Ø D4 FA ØØ 2Ø 69
ØF6Ø FA ØØ 8E 11 Ø2 A2 Ø3 2Ø
ØF68 CC FA ØØ 48 CA DØ F9 A2
ØF7Ø Ø3 68 38 E9 3F AØ Ø5 4A
ØF78 6E 11 Ø2 6E 1Ø Ø2 88 DØ

ØF8Ø F6 CA DØ ED A2 Ø2 2Ø CF
ØF88 FF C9 ØD FØ 1E C9 2Ø FØ
ØF9Ø F5 2Ø DØ FE ØØ BØ ØF 2Ø
ØF98 9C FA ØØ A4 C1 84 C2 85
ØFAØ C1 A9 3Ø 9D 1Ø Ø2 E8 9D
ØFA8 1Ø Ø2 E8 DØ DB 86 28 A2
ØFBØ ØØ ØØ 86 26 FØ Ø4 E6 26
ØFB8 FØ 75 A2 ØØ ØØ 86 1D A5
ØFCØ 26 2Ø D9 FC ØØ A6 2A 86
ØFC8 29 AA BC 37 FF ØØ BD 77
ØFDØ FF ØØ 2Ø B9 FE ØØ DØ E3
ØFD8 A2 Ø6 EØ Ø3 DØ 19 A4 1F
ØFEØ FØ 15 A5 2A C9 E8 A9 3Ø
ØFE8 BØ 21 2Ø BF FE ØØ DØ CC
ØFFØ 2Ø C1 FE ØØ DØ C7 88 DØ
ØFF8 EB Ø6 2A 9Ø ØB BC 3Ø FF

1ØØØ ØØ BD 2A FF ØØ 2Ø B9 FE
1ØØ8 ØØ DØ B5 CA DØ D1 FØ ØA
1Ø1Ø 2Ø B8 FE ØØ DØ AB 2Ø B8
1Ø18 FE ØØ DØ A6 A5 28 C5 1D
1Ø2Ø DØ AØ 2Ø 69 FA ØØ A4 1F
1Ø28 FØ 28 A5 29 C9 9D DØ 1A
1Ø3Ø 2Ø 1C FB ØØ 9Ø ØA 98 DØ
1Ø38 Ø4 A5 1E 1Ø ØA 4C ED FA
1Ø4Ø ØØ C8 DØ FA A5 1E 1Ø F6
1Ø48 A4 1F DØ Ø3 B9 C2 ØØ ØØ
1Ø5Ø 91 C1 88 DØ F8 A5 26 91
1Ø58 C1 2Ø CA FC ØØ 85 C1 84
1Ø6Ø C2 A9 9Ø 2Ø D2 FF AØ 41
1Ø68 2Ø C2 F8 ØØ 2Ø 54 FD ØØ
```

```
1070 20 41 FA 00 20 54 FD 00
1078 A9 05 20 D2 FF 4C B0 FD

1080 00 A8 20 BF FE 00 D0 11
1088 98 F0 0E 86 1C A6 1D DD
1090 10 02 08 E8 86 1D A6 1C
1098 28 60 C9 30 90 03 C9 47
10A0 60 38 60 40 02 45 03 D0
10A8 08 40 09 30 22 45 33 D0
10B0 08 40 09 40 02 45 33 D0
10B8 08 40 09 40 02 45 B3 D0
10C0 08 40 09 00 00 22 44 33
10C8 D0 8C 44 00 00 11 22 44
10D0 33 D0 8C 44 9A 10 22 44
10D8 33 D0 08 40 09 10 22 44
10E0 33 D0 08 40 09 62 13 78
10E8 A9 00 00 21 81 82 00 00
10F0 00 00 59 4D 91 92 86 4A
10F8 85 9D 2C 29 2C 23 28 24

1100 59 00 00 58 24 24 00 00
1108 1C 8A 1C 23 5D 8B 1B A1
1110 9D 8A 1D 23 9D 8B 1D A1
1118 00 00 29 19 AE 69 A8 19
1120 23 24 53 1B 23 24 53 19
1128 A1 00 00 1A 5B 5B A5 69
1130 24 24 AE AE A8 AD 29 00
1138 00 7C 00 00 15 9C 6D 9C
1140 A5 69 29 53 84 13 34 11
1148 A5 69 23 A0 D8 62 5A 48
1150 26 62 94 88 54 44 C8 54
1158 68 44 E8 94 00 00 B4 08
1160 84 74 B4 28 6E 74 F4 CC
1168 4A 72 F2 A4 8A 00 00 AA
1170 A2 A2 74 74 74 72 44 68
1178 B2 32 B2 00 00 22 00 00

1180 1A 1A 26 26 72 72 88 C8
1188 C4 CA 26 48 44 44 A2 C8
```

1190	3A	3B	52	4D	47	58	4C	53
1198	54	46	48	44	50	2C	41	42
11A0	F9	00	35	F9	00	CC	F8	00
11A8	F7	F8	00	56	F9	00	89	F9
11B0	00	F4	F9	00	0C	FA	00	3E
11B8	FB	00	92	FB	00	C0	FB	00
11C0	38	FC	00	5B	FD	00	8A	FD
11C8	00	AC	FD	00	46	F8	00	FF
11D0	F7	00	ED	F7	00	0D	20	20
11D8	20	50	43	20	20	53	52	20
11E0	41	43	20	58	52	20	59	52
11E8	20	53	50	AA	AA	AA	AA	AA

Program 3. Supermon64 Checksum.

```
100 REM SUPERMON64 CHECKSUM PROGRAM
110 DATA 10170,13676,15404,14997,15136,
    16221,16696,12816,16228,14554
120 DATA14677,15039,14551,15104,15522,
    16414,15914,8958,11945 :S=2048
130 FORB=1TO19:READX:FORI=STOS+128:N=P
    EEK(I):Y=Y+N
140 NEXTI:IFY<>XTHENPRINT"ERROR IN
    BLOCK #"B:GOTO160
150 PRINT"BLOCK #"B" IS CORRECT"
160 S=I:Y=0:NEXTB:REM CHECK LAST SHORT
    BLOCK BY HAND
```

Appendix G
The Wedge

One of the best reasons to learn machine language is that it can improve your BASIC programming significantly. There are two main ways that machine language can assist BASIC programming: adding commands to BASIC itself and replacing parts of a BASIC program with a high-velocity machine language subroutine. To add an ML subroutine to a BASIC *program*, you SYS, USR, or CALL (from Microsoft, Atari, or Apple BASICs respectively). That's fairly straightforward. To make changes to the BASIC *language* itself, however, we need to *wedge* into BASIC somehow.

You can make BASIC a *customized* language with a wedge. Do you want auto-numbering when writing a program in BASIC? Add it. Does your BASIC lack a RENUMBER facility? You can give it one. Do you want all your BASIC programs to contain a REM line with your name in it? This could be automatically put into each of your programs if you know machine language. Using a wedge to a machine language program, you can communicate directly to your machine, bypass BASIC's limitations, and do pretty much what you want to do.

How To Wedge In

Adding commands to BASIC is a matter of interrupting a loop. This is often referred to as adding a *wedge* into BASIC. Under the control of the BASIC language, the computer is looking to see if a BASIC word has been typed in, followed by a hit on the RETURN key. Or, during a RUN, the computer examines the program in memory to see what you want accomplished.

These, then, are the two contexts in which the computer analyzes a BASIC word: in a program or in "direct mode." In direct mode, you can type the word "LIST" onto the screen and hit the RETURN key. The computer looks up the meaning of "LIST" in a table of words which includes the addresses of the appropriate ML subroutines. It then JSR's (Jumps to a SubRoutine) somewhere in the vast ML of your computer's BASIC. This subroutine performs the actions necessary to provide you with a listing of the program in your computer's memory. If you could add some additional words to this table, you could add to BASIC. You could customize it.

Here's how. When you first turn on a computer which uses Microsoft BASIC, one of the first things that happens is that the operating system puts some important ML instructions into a zone in the first 256 memory locations (this area of RAM is called *zero page*). These instructions are put into zero page to handle the loop — often called the *CHRGET* loop (which means "character get") — where the operating system will forever after jump while power is on. This location is of great importance to BASIC; it is the "did they type any BASIC into the computer?" subroutine. It's where BASIC analyzes what it finds on screen or in a program, looking at something character by character to see what it adds up to.

If you type "LIST," this little zero page ML subroutine looks at the "L" then the "I" and so on. The exact location of CHRGET differs on the various computers:

PET (Original BASIC):	**decimal address**	**194-217**
PET/CBM (Upgrade & 4.0):		**112-135**
VIC:		**115-138**
64:		**115-138**
Apple:		**177-200**

The CHRGET ML program looks like this:

```
0070 E6 77       INC $77
0072 D0 02       BNE $0076
0074 E6 78       INC $78
0076 AD 03 02    LDA $0203
0079 C9 3A       CMP #$3A
007B B0 0A       BCS $0087
007D C9 20       CMP #$20
007F F0 EF       BEQ $0070
0081 38          SEC
0082 E9 30       SBC #$30
0084 38          SEC
0085 E9 D0       SBC #$D0
0087 60          RTS
```

This is put into your zero page RAM within the first few seconds after you turn on the computer. You can change it (RAM memory can be changed) to jump (JMP) to your own ML program by replacing the first three bytes of code. In our example above, we will replace the three bytes at hexadecimal location 0070 (the exact address will vary according to the CHRGET location as listed above for the different computers). Here is how the replacement looks in the example CHRGET routine:

```
0070 4C 00 75    JMP $7500
0073 02          ???
```

```
0074 E6 78      INC $78
0076 AD 02 02   LDA $0202
0079 C9 3A      CMP #$3A
007B B0 0A      BCS $0087
007D C9 20      CMP #$20
007F F0 EF      BEQ $0070
0081 38         SEC
0082 E9 30      SBC #$30
0084 38         SEC
0085 E9 D0      SBC #$D0
0087 60         RTS
```

The effect that this has is dramatic. Whenever the computer looks for a character in BASIC mode, it will jump first (because you forced it to) to your personal ML ''wedged'' routine located at $7500. The subroutine at $7500 could be anything you wanted it to be, anything you've put at address $7500. For an example, we've caused an ''A'' to appear on the PET/CBM screen:

```
7500 E6 77      INC $77
7502 D0 02      BNE $7506
7504 E6 78      INC $78
7506 A9 41      LDA #$41
7508 8D 00 80   STA $8000
750B 4C 76 00   JMP $0076
```

Notice that we had to first perform the actions that the CHRGET would have performed. Before we can start our LDA #$41 to put an ''A'' on screen, we had to replace the early part of CHRGET that we wrote over (see 7500 to 7505 in Example 3). And, after we're done with our custom routine, we jump back into CHRGET at 750B.

Adding a wedge to Atari BASIC is somewhat more involved. A clear and complete exposition of the techniques involved appears in an article by my colleague Charles Brannon, ''The Atari Wedge'' (*COMPUTE!* Magazine, November 1982).

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

COMPUTE! Books

P.O. Box 5406 Greensboro, NC 27403

Ask your retailer for these **COMPUTE! Books**. If he or she has sold out, order directly from **COMPUTE!**

For Fastest Service
Call Our **TOLL FREE US Order Line**
800-334-0868
In NC call 919-275-9809

Quantity	Title	Price	Total
______	The Beginner's Guide to Buying A Personal Computer	**$ 3.95****	______
______	COMPUTE!'s First Book of Atari	**$12.95***	______
______	Inside Atari DOS	**$19.95***	______
______	COMPUTE!'s First Book of PET/CBM	**$12.95***	______
______	Programming the PET/CBM	**$24.95*****	______
______	Every Kid's First Book of Robots and Computers	**$ 4.95****	______
______	COMPUTE!'s Second Book of Atari	**$12.95***	______
______	COMPUTE!'s First Book of VIC	**$12.95***	______
______	COMPUTE!'s First Book of Atari Graphics	**$12.95***	______
______	Mapping the Atari	**$14.95***	______
______	Home Energy Applications On Your Personal Computer	**$14.95***	______
______	Machine Language for Beginners	**$12.95***	______

* Add $2 shipping and handling. Outside US add $4 air mail; $2 surface mail.

** Add $1 shipping and handling. Outside US add $4 air mail; $2 surface mail.

*** Add $3 shipping and handling. Outside US add $9 air mail; $3 surface mail.

Please add shipping and handling for each book ordered. ______

Total enclosed or to be charged. ______

All orders must be prepaid (money order, check, or charge). All payments must be in US funds. NC residents add 4% sales tax.

☐ Payment enclosed Please charge my: ☐ VISA ☐ MasterCard
☐ American Express Acc't. No. ______________ Expires ___/___

Name ______________________________

Address ______________________________

City ______________ State ______________ Zip ______________

Country ______________________________

Allow 4-5 weeks for delivery.

Machine Language For Beginners

"Most books about machine language assume a considerable familiarity with both the details of microprocessor chips and with programming techniques. This book only assumes a working knowledge of BASIC. It was designed to speak directly to the amateur programmer, the part-time computerist. It should help you make the transition from BASIC to machine language with relative ease."

— From The Introduction

Contains everything you need to learn 6502 machine language including:

- A dictionary of all major BASIC words and their machine language equivalents. This section contains many sample programs and illustrations of how all the familiar BASIC programming techniques are accomplished in machine language.

- A complete Assembler program which supports pseudo-ops, forward branches, two number systems, and number conversions. It can easily be customized following the step-by-step instructions to make it perform any functions you want to add.

- A Disassembler program with graphic illustrations of jumps and subroutine boundaries.

- An easy-to-use number chart for quick conversions.

- Memory maps, monitor extensions, and all 6502 commands arranged for easy reference.

- Many clear, understandable examples and comparisons to already familiar BASIC programming methods.

ISBN 0-942386-11-6

$14.95