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

Copyright © 1983, Small System Services, Inc. All rights reserved.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the United States Copyright Act without the permission of the copyright owner is unlawful.

Printed in the United States of America

ISBN 0-942386-11-6

10 9 8 7 6 5 4
Table of Contents

Preface .......................................................... v
Chapter 1: How To Use This Book ......................... 1
Chapter 2: The Fundamentals ............................. 7
Chapter 3: The Monitor ...................................... 23
Chapter 4: Addressing ....................................... 37
Chapter 5: Arithmetic ....................................... 53
Chapter 6: The Instruction Set ............................ 63
Chapter 7: Borrowing from BASIC ....................... 91
Chapter 8: Building A Program ........................... 97
Chapter 9: ML Equivalents Of BASIC Commands .......... 121
Appendices
A: Instruction Set ........................................... 149
B: Maps ....................................................... 167
C: Assembler Programs .................................... 223
D: Disassembler Programs ................................. 237
E: Number Charts ........................................... 243
F: Monitor Extensions ..................................... 253
G: The Wedge ............................................... 335
Index ............................................................ 339
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 1-1. 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 1-2. 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 1-3. 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 1-4. 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.

```
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 1-5. Apple Version.

```
100 FOR I = 770 TO 789: READ A: POKE I,A: NEXT
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 1-6. 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.
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/CBM. 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
How To Use This Book

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:

\[
X = \text{USR}(1536) \quad \text{(and hit RETURN)}
\]

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:

\[
\text{SYS 864} \quad \text{and you should see the "A" on the screen.}
\]

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.
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 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:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>00000001</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>00000010</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>00000011 (1 and 2)</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>00000100</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>00000101 (4 and 1)</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>00000110 (4 and 2)</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>00000111 (4 + 2 + 1)</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>00001000</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>00001001</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>00001010</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>00001011</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>00001100</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>00001101</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>00001110</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>00001111</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>00010000</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>00010001</td>
</tr>
</tbody>
</table>

(note new digits) ———> 0A 00001010
(note new column ———> 10 00010000
in the hex)
The Fundamentals

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$="0123456789ABCDEF"
2 PRINT"{CLEAR}{03 DOWN}PLEASE CHOOSE:
4 PRINT"{03 DOWN}{03 RIGHT}1-INPUT HEX & GET DECIMAL BACK.
5 REM NEW LINE HERE
6 PRINT"{02 DOWN} 2-INPUT DECIMAL TO GET HEX BACK.
7 GETK:IFK=0THEN7
9 PRINT"{CLEAR}" : ON KGOTO200,400
100 H$="":FORM=3 TO0STEP-1:N%=DE/(16^M):DE=DE-N%*16^M:H$=H$+MID$(HE$,N%+1,1)
101 RETURN
102 D=0:Q=3:FORM=1 TO4:FORW=0 TO15:IF MID$(H$,M,1)=MID$(HE$,W+1,1)THEN104
103 NEXTW
104 D1=W*(16^(Q)):D=D+D1:Q=Q-1:NEXTM
105 DE=INT(D):RETURN
200 INPUT"{02 DOWN}HEX";H$:GOSUBL02:PRINTS PC(11)"{UP}={REV}"DE"{LEFT}"
210 GOTO200
400 INPUT"{02 DOWN}DECIMAL";DE:GOSUBL00:PR
INTSPC(14)"{UP}={REV}"H$""
410GOTO400

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 BACK."  
160 PRINT"2- INPUT DECIMAL AND GET HEX BACK."  
170 PRINT:PRINT"=>":GET#1,K  
180 IFK<490R>50THEN170  
190 PRINTCHR$(K):ONK-48GOTO300,400  
300 H$="@ABCDEFGHIJJKL"  
310 PRINT"HEX:";INPUTN$:N=0  
320 FORI=1TOLEN(N$)  
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.
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:

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:

And three would be one and two:

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.**

![Binary Pattern](image)

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
```
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 FOR X=0 TO 255:PRINT X;
140 IF X AND 1 THEN K(C)=49:GOTO 160
150 K(C)=48
160 C=C+1:IF B AND X THEN K(C)=49:GOTO 180
170 K(C)=48
180 B=B*2:IF C>8 THEN 200
190 GOTO 160
200 FOR I=0 TO 7:PRINT STR$(K(L)-48);:L=L-1
210 NEXT
220 C=0:PRINT
260 L=8:B=2:C=1:NEXT X
```

**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
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.**

```plaintext
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:
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 69 05 8D A0 0F 60
LDA #$02
ADC #$05
STA $0FA0
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.

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Object Code</th>
<th>Line Number</th>
<th>Disassembly</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000- A9 02</td>
<td>0100</td>
<td>0005</td>
<td>LDA #$02</td>
<td>LOAD A WITH 2</td>
</tr>
<tr>
<td>1002- 69 05</td>
<td>0110</td>
<td>0005</td>
<td>ADC #$05</td>
<td>ADD 5</td>
</tr>
<tr>
<td>1004- 8D A0 0F</td>
<td>0120</td>
<td>0005</td>
<td>STA $0FA0</td>
<td>STORE AT DECIMAL 4000</td>
</tr>
<tr>
<td>1007- 60</td>
<td>0130</td>
<td></td>
<td>RTS</td>
<td>RETURN</td>
</tr>
</tbody>
</table>

Source Code

Program 2-10. Labelled Assembly.

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Object Code</th>
<th>Line Number</th>
<th>Disassembly</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000- A9 02</td>
<td>0100</td>
<td>0005</td>
<td>LDA #$02</td>
<td>LOAD A WITH 2</td>
</tr>
<tr>
<td>1002- 69 05</td>
<td>0110</td>
<td>0005</td>
<td>ADC #$ADDER</td>
<td>ADD 5</td>
</tr>
<tr>
<td>1004- 8D A0 0F</td>
<td>0120</td>
<td>0005</td>
<td>STA STORAGE</td>
<td>STORE AT DECIMAL 4000</td>
</tr>
<tr>
<td>1007- 60</td>
<td>0130</td>
<td></td>
<td>RTS</td>
<td>RETURN</td>
</tr>
</tbody>
</table>

Comments

START ADDR $1000 (4096)
LOAD A WITH 2
ADD 5
STORE AT DECIMAL 4000
RETURN
END OF ASSEMBLY

--- LABEL FILE: ---

ADDER =0005
TWO =0002
START =1000
STORAGE =0FA0
### Program 2-11. The Source Code By Itself.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.BA $1000</td>
<td>START ADDR $1000 (4096)</td>
</tr>
<tr>
<td>.DE 2</td>
<td>DEFINE LABLE &quot;TWO&quot; AS 2.</td>
</tr>
<tr>
<td>.DE 5</td>
<td>DEFINE &quot;ADDER&quot; AS A 5.</td>
</tr>
<tr>
<td>.DE $0FA0</td>
<td>DEFINE STORAGE ADDR.</td>
</tr>
<tr>
<td></td>
<td>;DEFINE LABLE &quot;TWO&quot; AS 2.</td>
</tr>
<tr>
<td>LDA #TWO</td>
<td>LOAD A WITH 2</td>
</tr>
<tr>
<td>ADC #ADDER</td>
<td>ADD 5</td>
</tr>
<tr>
<td>STA STORAGE</td>
<td>STORE AT DECIMAL 4000</td>
</tr>
<tr>
<td>RTS</td>
<td>RETURN</td>
</tr>
<tr>
<td>.EN</td>
<td>END OF ASSEMBLY</td>
</tr>
</tbody>
</table>

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
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 Superman (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
3 The Monitor

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:


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):

<table>
<thead>
<tr>
<th>Address Field</th>
<th>Instruction Field</th>
<th>Argument (Operand) Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>LDY</td>
<td>#10</td>
</tr>
<tr>
<td>2002</td>
<td>LDA</td>
<td>#33</td>
</tr>
<tr>
<td>2004</td>
<td>STA</td>
<td>(88), Y</td>
</tr>
</tbody>
</table>

(The screen location is remembered by the Atari in addresses 88 and 89.)

<table>
<thead>
<tr>
<th>Address Field</th>
<th>Instruction Field</th>
<th>Argument (Operand) Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007</td>
<td>DEY</td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>BNE</td>
<td>2004</td>
</tr>
<tr>
<td>2010</td>
<td>RTS (or BRK)</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Line #</th>
<th>Label</th>
<th>Instruction</th>
<th>Argument</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>START</td>
<td>LDY</td>
<td>#10</td>
<td>Set up counter for loop</td>
</tr>
<tr>
<td>110</td>
<td></td>
<td>LDA</td>
<td>#33</td>
<td>“A” in ATASCII</td>
</tr>
<tr>
<td>120</td>
<td>LOOP</td>
<td>STA</td>
<td>(88), Y</td>
<td></td>
</tr>
<tr>
<td>130</td>
<td></td>
<td>DEY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td></td>
<td>BNE</td>
<td>LOOP</td>
<td>Loop until zero</td>
</tr>
</tbody>
</table>

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
3 The Monitor

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):

<table>
<thead>
<tr>
<th>PC</th>
<th>IRQ</th>
<th>SR</th>
<th>AC</th>
<th>XR</th>
<th>YR</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>E62E</td>
<td>30</td>
<td>00</td>
<td>05</td>
<td>FF</td>
<td>FE</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>ASCII</th>
<th>ATARI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1500 80 P</td>
<td>48 30 36</td>
</tr>
<tr>
<td>1501 82 R</td>
<td>50 38 32</td>
</tr>
<tr>
<td>1502 69 E</td>
<td>37 35 25</td>
</tr>
<tr>
<td>1503 83 S</td>
<td>51 31 23</td>
</tr>
<tr>
<td>1504 83 S</td>
<td>51 39 33</td>
</tr>
<tr>
<td>1505 32</td>
<td>0</td>
</tr>
<tr>
<td>1506 65 A</td>
<td>33 21</td>
</tr>
<tr>
<td>1507 78 N</td>
<td>46 26</td>
</tr>
<tr>
<td>1508 89 Y</td>
<td>57 39</td>
</tr>
<tr>
<td>1509 32</td>
<td>0</td>
</tr>
<tr>
<td>150A 75 K</td>
<td>43 3B</td>
</tr>
<tr>
<td>150B 69 E</td>
<td>37 25</td>
</tr>
<tr>
<td>150C 89 Y</td>
<td>57 39</td>
</tr>
<tr>
<td>150D 00</td>
<td>255</td>
</tr>
</tbody>
</table>

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.
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.**

<table>
<thead>
<tr>
<th>PC</th>
<th>IRQ</th>
<th>SR</th>
<th>AC</th>
<th>XR</th>
<th>YR</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0005</td>
<td>E455</td>
<td>30</td>
<td>00</td>
<td>5E</td>
<td>04</td>
<td>F8</td>
</tr>
</tbody>
</table>

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  
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.
4 Addressing

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.

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:

$$5000 \text{ STA } (90,X)$$

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:

- **Pointer #1**
  - $0090 $00
  - (it points to $0400)
- **Pointer #2**
  - $0091 $04
  - ($7005)
- **Pointer #3**
  - $0092 $05
  - ($80EA)
- **Pointer #4**
  - $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.
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 $5 \times 4$, 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.

```
0005 .BA $1010
0010 ; -------- STORAGE AREAS --------
0020 TESTED .DE $1000
0030 SECOND .DE $1002
0040 TEMP .DE $1008
0050 ;
0060 ; -------- LANDING PLACES --------
0070 LOWER .DE $1004
0080 EQUAL .DE $1005
0090 HIGHER .DE $1006
0100 ;
1010- 38 0110 START SEC
1011- AD 00 10 0120 LDA TESTED ; COMPARE THE LOW BYTES
1014- ED 02 10 0130 SBC SECOND
1017- 8D 08 10 0140 STA TEMP
101A- AD 01 10 0150 LDA TESTED+1 ; COMPARE THE HIGH BYTES
101D- ED 03 10 0160 SBC SECOND+1
1020- 0D 08 10 0170 ORA TEMP
1023- F0 E0 0180 BEQ EQUAL ; TESTED = SECOND
1025- 90 DD 0190 BCC LOWER ; TESTED < SECOND
1027- B0 DD 0200 BCS HIGHER ; TESTED > SECOND
0210 .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).
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.

I. 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)
**LOA and LOY** work the same.

Be sure you understand what is happening here. **LOX $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 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 are sometimes called "equates.")

<table>
<thead>
<tr>
<th></th>
<th>TESTBYTE</th>
<th>NEWBYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004</td>
<td>LDA</td>
<td>TESTBYTE</td>
</tr>
<tr>
<td>2006</td>
<td>CMP</td>
<td>NEWBYTE</td>
</tr>
<tr>
<td>2008</td>
<td>BNE</td>
<td>BRANCHTARGET</td>
</tr>
<tr>
<td>200A</td>
<td>JR</td>
<td>SPECIALSUBROUTINE</td>
</tr>
</tbody>
</table>

"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 ADDRESS)
2006- C5 99 0060 CMP *NEWBYTE ; (ZERO PAGE ADDRESSING)
2008- D0 03 0070 BNE BRANCHTARGET ; (RELATIVE ADDRESS)
200A- 20 10 20 0080 JSR SPECIALSUBROUTINE
200C- AD 00 04 0090 BRANCHTARGET LDA $400 ; YOU CAN FREELY MIX
0100 ; LABELS 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

The Instruction Set 6
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

\[
\text{FOR } 1 = 1 \ \text{TO 10: NEXT I,}
\]

the value of the variable I goes up by one each time the loop cycles around. The same effect is created by:

\[
\begin{align*}
2000 & \text{ LDX } #10 \\
2002 & \text{ DEX } \quad ("\text{DEcrement"} \ or \ "\text{DEcrease X"} \ by \ 1) \\
2003 & \text{ BNE } 2002 \quad (\text{Branch if Not Equal [to zero] back up to address } 2002)
\end{align*}
\]

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, INcerease 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 range! Time nearly up!” (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.)

JMP 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
The Instruction Set

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
The Instruction Set

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 $N00!

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:

<table>
<thead>
<tr>
<th></th>
<th>AND</th>
<th>OR</th>
<th>EOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 AND 0 = 0</td>
<td>0 OR 0 = 0</td>
<td>0 EOR 0 = 0</td>
</tr>
<tr>
<td>0</td>
<td>0 AND 1 = 0</td>
<td>0 OR 1 = 1</td>
<td>0 EOR 1 = 1</td>
</tr>
<tr>
<td>1</td>
<td>1 AND 0 = 0</td>
<td>1 OR 0 = 1</td>
<td>1 EOR 0 = 1</td>
</tr>
<tr>
<td>1</td>
<td>1 AND 1 = 1</td>
<td>1 OR 1 = 1</td>
<td>1 EOR 1 = 0</td>
</tr>
</tbody>
</table>

**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?
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.

Kernels 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:

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFCF</td>
<td>4C 15</td>
<td>F2 (INPUT one byte)</td>
</tr>
<tr>
<td>FFD2</td>
<td>4C 66</td>
<td>F2 (OUTPUT one byte)</td>
</tr>
<tr>
<td>FFD5</td>
<td>4C 01</td>
<td>F4 (LOAD something)</td>
</tr>
<tr>
<td>FFD8</td>
<td>4C DD</td>
<td>F6 (SAVE something)</td>
</tr>
</tbody>
</table>

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:

\[
\begin{align*}
2000 & \text{ LDA } \# 65 \quad \text{(put 'a' into the accumulator)} \\
2002 & \text{ JSR } \$FBFD \quad \text{(print it)}
\end{align*}
\]

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

\[
\begin{align*}
2000 & \text{ JSR } \$FFE4 \quad \text{(GET whatever key is being pressed)} \\
2003 & \text{ BEQ } 2000 \quad \text{(if no key is pressed, a zero is in the accumulator, so you BEQ back and try for a character again)} \\
2005 & \text{ RTS} \quad \text{(the character's value is in the accumulator)}
\end{align*}
\]

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.

0010 ; COMMODORE & APPLE VERSION
0020 .BA $2000
0030 .OS ; (OUTPUT SOURCE CODE)
0040 COUNTER .DE $55 ; (WILL HOLD INDEX)
0050 STRING .BY 'SUPERDUPER' ; STORE THIS TEXT STRING

2000- 53 55 50
2003- 45 52 44
2006- 55 50 45
2009- 52

0060 LENGTH .DE 11 ; STRING IS 10 CHARs. LONG
0070 ;
0080 PRINTIT .DE $FFD2 ; (COMMODORE)
0090 ;
0100 ; (FOR APPLE USE $FDED)
0110 ;

200A- A0 00 0120 START LDY #$00
200C- B9 00 20 0130 LOOP LDA STRING,Y
200F- 20 D2 FF 0140 JSR PRINTIT
2012- C8 0150 INY
2013- C0 0B 0160 CPY #$LENGTH ; (NOTE LENGTH IS PLUS ONE.
2015- D0 F5 0170 BNE LOOP
2017- 60 0180 RTS
0190 .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
060A- A9 00
060C- 85 55
060E- A0 55
0610- B9 00 06
0613- 20 A4 F6
0616- E6 55
0618- A9 0B
061A- C5 55
061C- D0 F0
061E- 60
0060 LENGTH .DE 11 ; STRING IS 10 CHARS. LONG
0070 ;
0080 PRINTIT .DE $F6A4 ; (ATARI)
0090 ;
0100 START LDA #00 ; (ANY FREE ZERO PAGE)
0110 STA *COUNTER
0120 LOOP LDY #COUNTER
0130 LDA STRING,Y
0140 JSR PRINTIT
0150 INC *COUNTER
0160 LDA #LENGTH
0170 CMP *COUNTER
0180 BNE LOOP
0190 RTS
0200 .EN
```

--- LABEL FILE: ---

<table>
<thead>
<tr>
<th>Counter</th>
<th>Length</th>
<th>Loop</th>
<th>PRINTIT</th>
<th>Start</th>
<th>STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>0055</td>
<td>000B</td>
<td>060E</td>
<td>F6A4</td>
<td>060A</td>
<td>0600</td>
</tr>
</tbody>
</table>
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
Building A Program 8

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 ;
0150 LDA BASIC+1 ;GET ADDR OF NEXT
0160 STA *L1L ;BASIC LINE
0170 LDA BASIC+2
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 README LDY #$00
036C- B1 BA 0200 LDA (L1L),Y ; NOT END OF LINE
036E- D0 06 0210 BNE GO.ON
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 ; 0- 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
CLC
ADC #$04
STA *LIL

MOVE FORWARD TO 1ST PART OF BASIC TEXT

(STARTING 1ST LINE # AND OF NEXT LINE)

0441

;-------------------------------

; O-- SUBROUTINE TO CHECK FOR ZERO (LINE IS FINISHED?)

; AND THEN CHECK 1ST CHARACTER IN BASIC LINE AGAINST

; 1ST CHARACTER IN SAMPLE STRING AT LINE 0:. IF THE

; 1ST CHARACTERS MATCH, WE MOVE TO A FULL STRING

; COMPARISON IN THE SUBROUTINE CALLED "SAME." IF 1ST

; CHAR. DON'T MATCH, WE RAISE THE "Y" COUNTER AND

; CHECK FOR A MATCH IN THE 2ND CHAR. OF THE CURRENT

; BASIC LINE'S TEXT.

LDY #$00
LDA (LIL), Y
BEQ STOPLINE
CMP BASIC+6
BEQ SAME
INY
JMP LOOP

=J
E:
Q:
::l
OQ
»
OQ
»
1
3

; O-- SUBROUTINE TO LOOK AT EACH CHARACTER IN BOTH THE

; SAMPLE (LINE 0) AND THE TARGET (CURRENT LINE) TO

; SEE IF THERE IS A PERFECT MATCH. Y KEEPS TRACK OF

; TARGET. X INDEXES SAMPLE. IF WE FIND A MISMATCH

; BEFORE A LINE-END ZERO, WE FALL THROUGH TO LINE

; 590 AND JUMP BACK UP TO 460 WHERE WE CONTINUE ON
Looking for 1st char matches in the current line.

0519

03A7 - A2 00
03A9 - E8
03AA - C8
03AB - BD 06 04
03AE - F0 07
03B0 - D1 BA
03B2 - F0 F5
03B4 - 4C 9A 03
03B7 - 20 C5 03
03BA - A5 BC
03BC - 85 BA
03BE - A5 BD
03C0 - 85 BB
03C2 - 4C 6A 03

0520 SAME
0530 COMPARE
0540 INY
0550 LDA BASIC+6,X
0560 BEQ PERFECT
0570 CMP (L1L),Y
0580 BEQ COMPARE
0590 JMP LOOP
0600 PERFECT
0610 STOPLINE
0620 STA *L1L
0630 LDA *L2L+1
0640 STA *L1L+1
0650 JMP READLINE

0518 ;
0519 ;
0520 SAME LDX #$00 ; COMPARING SAMPLE TO
0530 COMPARE INX ; TARGET
0540 INY
0550 LDA BASIC+6,X
0560 BEQ PERFECT ; LINE ENDS SO PRINT
0570 CMP (L1L),Y
0580 BEQ COMPARE ; CONTINUE COMPARING
0590 JMP LOOP ; NO MATCH
0600 PERFECT JSR PRINTOUT
0610 STOPLINE LDA *L2L ; TRANSFER NEXT LINE
0620 STA *L1L ; ADDRESS POINTER TO
0630 LDA *L2L+1 ; CURRENT LINE POINTER
0640 STA *L1L+1 ; TO GET READY TO READ
0650 JMP READLINE ; THE NEXT LINE.

0601 ; -------
0602 ; -0- 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 ;
0609 ;
0610 STOPLINE LDA *L2L ; TRANSFER NEXT LINE
0620 STA *L1L ; ADDRESS POINTER TO
0630 LDA *L2L+1 ; CURRENT LINE POINTER
0640 STA *L1L+1 ; TO GET READY TO READ
0650 JMP READLINE ; THE NEXT LINE.

0651 ; -------
0652 ; -0- 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 ; --- LABEL FILE: ---
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 04 0A 00 99 22 48 49 22 00 11 04 14 00 80 00 00 00
040B 04 0A 00 99 22 49 22 00 11 04 14 00 80 00 00 00
0411

LINE 10
LINE ? "HI"
LINE 20
LINE END
Figure 8-2. Atari BASIC Structure.

Single Statement Lines

START P

Points to first line

LO HI $88 $99

Line # 32768 Means End of Program

Line offset $16 Line #

Statement offset $16 Tokenized code

First basic line

Second basic line
10 PRINT "HI"
20 END

EXAMPLE

10 BYTES

<table>
<thead>
<tr>
<th>LINE</th>
<th>LINE/STMT</th>
<th>OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

6 BYTES

<table>
<thead>
<tr>
<th>LIT. STRING</th>
<th>H</th>
<th>I</th>
<th>EOL</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>0A, 00</td>
<td>0A</td>
<td>0A</td>
<td>20</td>
<td>0F</td>
</tr>
<tr>
<td>48 49 16</td>
<td>14</td>
<td>00</td>
<td>06</td>
<td>06</td>
</tr>
<tr>
<td>15 16 00, 80</td>
<td>32768</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MULTI-STATEMENT LINES

<table>
<thead>
<tr>
<th>LINE #</th>
<th>LINE OFFSET</th>
<th>STMT OFFSET</th>
<th>TOKENS 1ST STMT</th>
<th>2ND OFFSET</th>
<th>TOKENS 2ND STMT</th>
</tr>
</thead>
</table>

EOL
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 UL. 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  
RTS

**SUBROUTINES**

1  
2  
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:

\[
\text{POS} = \text{USR} (1536, \text{ADR}(A$), \text{LEN}(A$), \text{ADR}(B$), \text{LEN}(B$))
\]

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 \( \text{POS} - \text{ADR}(A$) + 1 \). 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.
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00CB</td>
<td>Variables in zero page for speed</td>
</tr>
<tr>
<td>$00CC</td>
<td>$00CD</td>
</tr>
<tr>
<td>$0190</td>
<td>$0200</td>
</tr>
<tr>
<td>$01A0</td>
<td>$01B0</td>
</tr>
</tbody>
</table>

Program 8.2.
00D8 0340 TADRH  =$D8
00D9 0350 ENDOLOOP =$D9
0360 ;
0370 ;Syntax documentation
0380 ;
0390 ;FIND:Find Text
0400 ;X=USR(FIND,A,B,C,D)
0410 ;FIND:Address of utility (1536)
0420 ;A: Where to start search
0430 ;B: Where to quit searching
0440 ;C: Search string address
0450 ;D: Length of search string
0460 ;X: Position found (=0 if no match)
0000 0470 *= $0600
0480 ;-------------------------------------
0490 ;This portion sets up the parameters
0500 ;for the search by pulling the values
0510 ;passed by BASIC off the stack
0520 ;
0530 FIND
0600 68 0540 PLA ;Count byte
0601 68 0550 PLA ;hi byte, Source start
0602 85CC 0560 STA SADRH
0604 68 0570 PLA ;lo byte, Source start
0605 85CB 0580 STA SADRL
0607 68 0590 PLA ;hi byte, Source end
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.
0617 A000 0850 LDY #0
0619 B1CF 0860 LDA (FNDL),Y ;Set up first
061B 85D2 0870 STA FIRSTCHAR ;comparison
0880 ;
061D A6CE 0890 LDX SLENH ;Less than 255
061F F018 08A0 BEQ SHORT ;bytes?
0910 NXTSRCH
0621 A9FF 08B0 LDA #255 ;Select end
0920 SEARCH2
0623 85D9 08C0 STA ENDMETHOD
0625 A000 08D0 LDY #0
0930 SEARCHLOOP
0627 B1CB 08E0 LDA (SADRL),Y
0629 C5D2 08F0 CMP FIRSTCHAR ;Found a match?
062B F017 0900 BEQ FOUND1 ;yes
1000 NOTFOUND
062D C8 1010 INY ;no
062E C4D9 1020 CPY ENDMETHOD
0630 D0F5 1030 BNE SEARCHLOOP ;continue
1040 ;
0632 E6CC 1050 INC SADRH ;Next block
0634 CA 1060 DEX ;Done?
0635 3006 1070 BMI EXIT ;yes
0637 D0E8 1080 BNE NXTSRCH ;nope
1090 SHORT
0639 A5CD 1100 LDA SLENL ;Set up last
063B D0E6 1110 BNE SEARCH2 ; scan
1120 EXIT

063D A900 1130 LDA #0 ; return
063F 85D4 1140 STA FR0 ; = 0
0641 85D5 1150 STA FR0+1 ; no string
0643 60 1160 RTS ; found
1170 ;
1180 ; ---------------------------------
1190 ; Here is where we check for a full match, starting with the second character of the search string
1200 ; We have to use two "pseudo" registers in memory, since the same Y register is needed to access both areas of memory
1210 ; (search space and search string)
1220 ;
1230 ;
1240 ;
1250 ;
1260 ;
1270 FOUND1

0644 84D4 1280 STY FR0 ; Save Y
0646 84D3 1290 STY SINDEX ; Source index
0648 A001 1300 LDY #1
064A 84D6 1310 STY FINDEX ; Find index
1320 ;
1330 ; We use a temporary address, since we don't want to change the address in SADR (so we can continue the search if no match found)
1340 ;
1350 ;
1360 ;
As long as each character matches, we continue to compare until we get a failed comparison or reach the end of the search string, which indicates a match.

If the comparison fails, we hit page bound, increment the TADR and try again. If the comparison is successful, we update the SINDEX and continue searching.
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 TADR
=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.


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 ADRES,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
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.**

<table>
<thead>
<tr>
<th>DATA</th>
<th>INITIALIZATION</th>
<th>MAIN LOOP</th>
<th>DATA</th>
</tr>
</thead>
</table>

---

**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.
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.

\begin{verbatim}
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)
\end{verbatim}

**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:

\begin{verbatim}
2000 LDX #200 (or $C8 hex)
2002 DEX
2003 BNE 2002
\end{verbatim}
FOR-NEXT-STEP

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
GOSUB

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 $NNNNN and you will go to a subroutine at address NNNN instead of a line number, as in BASIC. ("'NNNNN'" 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. Superman and Micromon are such extensions. The format for Moveit in Commodore machines is .T 2000 20065000 (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)

5015 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 databases 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 Assembler 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 FOR I = START TO FIN: GET#1,A: POKE I,A:NEXT I
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
ON GOSUB

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)
```
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 reading 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 x 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 control 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:

<table>
<thead>
<tr>
<th>Computer</th>
<th>You Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple II</td>
<td>G 2000</td>
</tr>
<tr>
<td>PET, VIC, 64</td>
<td>G 2000</td>
</tr>
<tr>
<td>Atari</td>
<td>G 2000</td>
</tr>
</tbody>
</table>
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:

\[BSAVE\ Name,A,L\ \ (A = \text{address, } L = \text{length})\]

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

\[.S \text{‘‘PROGRAM NAME’’},01,2000,2010\] (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:

\[.S \text{‘‘0:NAME’’},08,2000,2010\]

(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.
ASC

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.

CHRS

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 CHRS 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.

MIDS

To extract a substring which starts at the fourth character from within the string and is five characters long (as in MIDS(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

RIGHTS$ 

This, too, is complicated because normally we do not know the LENgth of a given string. To find RIGHTS$(X$,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)
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

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addressing Mode</strong></td>
<td><strong>Mnemonics</strong></td>
<td><strong>Opcode</strong></td>
<td><strong>Size In Bytes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediate</td>
<td>ADC #Arg</td>
<td>69</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero Page</td>
<td>ADC Arg</td>
<td>65</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>ADC Arg, X</td>
<td>75</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute</td>
<td>ADC Arg</td>
<td>6D</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute, X</td>
<td>ADC Arg, X</td>
<td>7D</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute, Y</td>
<td>ADC Arg, Y</td>
<td>79</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Indirect, X)</td>
<td>ADC (Arg, X)</td>
<td>61</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Indirect), Y</td>
<td>ADC (Arg), Y</td>
<td>71</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### AND “AND” Memory With Accumulator

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addressing Mode</strong></td>
<td><strong>Mnemonics</strong></td>
<td><strong>Opcode</strong></td>
<td><strong>Size In Bytes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediate</td>
<td>AND # Arg</td>
<td>29</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero Page</td>
<td>AND Arg</td>
<td>25</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>AND Arg, X</td>
<td>35</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute</td>
<td>AND Arg</td>
<td>2D</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute, X</td>
<td>AND Arg, X</td>
<td>3D</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute, Y</td>
<td>AND Arg, Y</td>
<td>39</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Indirect, X)</td>
<td>AND (Arg, X)</td>
<td>21</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Indirect), Y</td>
<td>AND (Arg), Y</td>
<td>31</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## ASL

Shift Left One Bit

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N Z C I D V</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Addressing Mode</strong></td>
<td><strong>Mnemonics</strong></td>
<td><strong>Opcode</strong></td>
<td></td>
</tr>
<tr>
<td>Accumulator</td>
<td>ASL A</td>
<td>0A</td>
<td>1</td>
</tr>
<tr>
<td>Zero Page</td>
<td>ASL Arg</td>
<td>06</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>ASL Arg, X</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>Absolute</td>
<td>ASL Arg</td>
<td>0E</td>
<td>3</td>
</tr>
<tr>
<td>Absolute, X</td>
<td>ASL Arg, X</td>
<td>1E</td>
<td>3</td>
</tr>
</tbody>
</table>

### BCC

Branch On Carry Clear

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N Z C I D V</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Addressing Mode</strong></td>
<td><strong>Mnemonics</strong></td>
<td><strong>Opcode</strong></td>
<td></td>
</tr>
<tr>
<td>Relative</td>
<td>BCC Arg</td>
<td>90</td>
<td>2</td>
</tr>
</tbody>
</table>

### BCS

Branch On Carry Set

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N Z C I D V</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Addressing Mode</strong></td>
<td><strong>Mnemonics</strong></td>
<td><strong>Opcode</strong></td>
<td></td>
</tr>
<tr>
<td>Relative</td>
<td>BCS Arg</td>
<td>B0</td>
<td>2</td>
</tr>
</tbody>
</table>
### BEQ
#### Branch On Zero

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Z C I D V</td>
<td>Relative</td>
<td>BEQ Arg</td>
<td>F0</td>
<td>2</td>
</tr>
</tbody>
</table>

### BIT
#### Test Bits In Memory Against Accumulator

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Z C I D V</td>
<td>Zero Page</td>
<td>BIT Arg</td>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Absolute</td>
<td>BIT Arg</td>
<td>2C</td>
<td>3</td>
</tr>
</tbody>
</table>

### BMI
#### Branch On Minus

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Z C I D V</td>
<td>Relative</td>
<td>BMI Arg</td>
<td>30</td>
<td>2</td>
</tr>
</tbody>
</table>

### BNE
#### Branch On Anything But Zero

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Z C I D V</td>
<td>Relative</td>
<td>BNE Arg</td>
<td>D0</td>
<td>2</td>
</tr>
<tr>
<td>BPL</td>
<td>Branch On Plus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>----------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status Flags</td>
<td>N Z C I D V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addressing Mode</td>
<td>Mnemonics</td>
<td>Opcode</td>
<td>Size In Bytes</td>
<td></td>
</tr>
<tr>
<td>Relative</td>
<td>BPL Arg</td>
<td>10</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BRK</th>
<th>Break</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Flags</td>
<td>N Z C I D V</td>
</tr>
<tr>
<td>Addressing Mode</td>
<td>Mnemonics</td>
</tr>
<tr>
<td>Implied</td>
<td>BRK</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BVC</th>
<th>Branch On Overflow Clear</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Flags</td>
<td>N Z C I D V</td>
</tr>
<tr>
<td>Addressing Mode</td>
<td>Mnemonics</td>
</tr>
<tr>
<td>Relative</td>
<td>BVC Arg</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BVS</th>
<th>Branch On Overflow Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Flags</td>
<td>N Z C I D V</td>
</tr>
<tr>
<td>Addressing Mode</td>
<td>Mnemonics</td>
</tr>
<tr>
<td>Relative</td>
<td>BVS Arg</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>CLC</td>
<td>Clear Carry Flag</td>
</tr>
<tr>
<td>CLD</td>
<td>Clear Decimal Mode</td>
</tr>
<tr>
<td>CLI</td>
<td>Clear Interrupt Disable Bit</td>
</tr>
<tr>
<td>CLV</td>
<td>Clear Overflow Flag</td>
</tr>
</tbody>
</table>
### CMP

**Compare Memory And Accumulator**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Z C I D V</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>CMP # Arg</td>
<td>C9</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page</td>
<td>CMP Arg</td>
<td>C5</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>CMP Arg, X</td>
<td>D5</td>
<td>2</td>
</tr>
<tr>
<td>Absolute</td>
<td>CMP Arg</td>
<td>CD</td>
<td>3</td>
</tr>
<tr>
<td>Absolute, X</td>
<td>CMP Arg, X</td>
<td>DD</td>
<td>3</td>
</tr>
<tr>
<td>Absolute, Y</td>
<td>CMP Arg, Y</td>
<td>D9</td>
<td>3</td>
</tr>
<tr>
<td>(Indirect, X)</td>
<td>CMP (Arg, X)</td>
<td>C1</td>
<td>2</td>
</tr>
<tr>
<td>(Indirect), Y</td>
<td>CMP (Arg), Y</td>
<td>D1</td>
<td>2</td>
</tr>
</tbody>
</table>

### CPX

**Compare Memory Against X Register**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Z C I D V</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>CPX # Arg</td>
<td>E0</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page</td>
<td>CPX Arg</td>
<td>E4</td>
<td>2</td>
</tr>
<tr>
<td>Absolute</td>
<td>CPX Arg</td>
<td>EC</td>
<td>3</td>
</tr>
</tbody>
</table>

### CPY

**Compare Memory Against Y Register**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Z C I D V</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>CPY # Arg</td>
<td>C0</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page</td>
<td>CPY Arg</td>
<td>C4</td>
<td>2</td>
</tr>
<tr>
<td>Absolute</td>
<td>CPY Arg</td>
<td>CC</td>
<td>3</td>
</tr>
</tbody>
</table>
### DEY

#### DEC
**Decrement Memory By One**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Zero Page</td>
<td>DEC Arg</td>
<td>C6</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Zero Page, X</td>
<td>DEC Arg, X</td>
<td>D6</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Absolute</td>
<td>DEC Arg</td>
<td>CE</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Absolute, X</td>
<td>DEC Arg, X</td>
<td>DE</td>
<td>3</td>
</tr>
</tbody>
</table>

#### DEX
**Decrement X Register By One**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Implied</td>
<td>DEX</td>
<td>CA</td>
<td>1</td>
</tr>
</tbody>
</table>

#### DEY
**Decrement Y Register By One**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Implied</td>
<td>DEY</td>
<td>88</td>
<td>1</td>
</tr>
</tbody>
</table>
### EOR

**Exclusive—Or Memory With Accumulator**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>EOR # Arg</td>
<td>49</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page</td>
<td>EOR Arg</td>
<td>45</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>EOR Arg, X</td>
<td>55</td>
<td>2</td>
</tr>
<tr>
<td>Absolute</td>
<td>EOR Arg</td>
<td>4D</td>
<td>3</td>
</tr>
<tr>
<td>Absolute, X</td>
<td>EOR Arg, X</td>
<td>5D</td>
<td>3</td>
</tr>
<tr>
<td>Absolute, Y</td>
<td>EOR Arg, Y</td>
<td>59</td>
<td>3</td>
</tr>
<tr>
<td>(Indirect, X)</td>
<td>EOR (Arg, X)</td>
<td>41</td>
<td>2</td>
</tr>
<tr>
<td>(Indirect), Y</td>
<td>EOR (Arg), Y</td>
<td>51</td>
<td>2</td>
</tr>
</tbody>
</table>

### INC

**Increment Memory By One**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero Page</td>
<td>INC Arg</td>
<td>E6</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>INC Arg, X</td>
<td>F6</td>
<td>2</td>
</tr>
<tr>
<td>Absolute</td>
<td>INC Arg</td>
<td>EE</td>
<td>3</td>
</tr>
<tr>
<td>Absolute, X</td>
<td>INC Arg, X</td>
<td>FE</td>
<td>3</td>
</tr>
</tbody>
</table>

### INX

**Increment X Register By One**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied</td>
<td>INX</td>
<td>E8</td>
<td>1</td>
</tr>
</tbody>
</table>
### INY
**Increment Y Register By One**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addressing Mode</td>
<td>Mnemonics</td>
<td>Opcode</td>
<td>Size In Bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implied</td>
<td>INY</td>
<td>C8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### JMP
**Jump**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addressing Mode</td>
<td>Mnemonics</td>
<td>Opcode</td>
<td>Size In Bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute</td>
<td>JMP Arg</td>
<td>4C</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indirect</td>
<td>JMP (Arg)</td>
<td>6C</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### JSR
**Jump To New Location, But Save Return Address**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addressing Mode</td>
<td>Mnemonics</td>
<td>Opcode</td>
<td>Size In Bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute</td>
<td>JSR Arg</td>
<td>20</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### LDA

**Load Accumulator With Memory**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>LDA # Arg</td>
<td>A9</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page</td>
<td>LDA Arg</td>
<td>A5</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>LDA Arg, X</td>
<td>B5</td>
<td>2</td>
</tr>
<tr>
<td>Absolute</td>
<td>LDA Arg</td>
<td>AD</td>
<td>3</td>
</tr>
<tr>
<td>Absolute, X</td>
<td>LDA Arg, X</td>
<td>BD</td>
<td>3</td>
</tr>
<tr>
<td>Absolute, Y</td>
<td>LDA Arg, Y</td>
<td>B9</td>
<td>3</td>
</tr>
<tr>
<td>(Indirect, X)</td>
<td>LDA (Arg, X)</td>
<td>A1</td>
<td>2</td>
</tr>
<tr>
<td>(Indirect), Y</td>
<td>LDA (Arg), Y</td>
<td>B1</td>
<td>2</td>
</tr>
</tbody>
</table>

### LDX

**Load X Register**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>LDX # Arg</td>
<td>A2</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page</td>
<td>LDX Arg</td>
<td>A6</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page, Y</td>
<td>LDX Arg, Y</td>
<td>B6</td>
<td>2</td>
</tr>
<tr>
<td>Absolute</td>
<td>LDX Arg</td>
<td>AE</td>
<td>3</td>
</tr>
<tr>
<td>Absolute, Y</td>
<td>LDX Arg, Y</td>
<td>BE</td>
<td>3</td>
</tr>
</tbody>
</table>
## LDY
### Load Y Register

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addressing Mode</strong></td>
<td><strong>Mnemonics</strong></td>
<td><strong>Opcode</strong></td>
<td><strong>Size In Bytes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediate</td>
<td>LDY # Arg</td>
<td>A0</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero Page</td>
<td>LDY Arg</td>
<td>A4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>LDY Arg, X</td>
<td>B4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute</td>
<td>LDY Arg</td>
<td>AC</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute, X</td>
<td>LDY Arg, X</td>
<td>BC</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## LSR
### Shift Right One Bit In Either Memory Or Accumulator

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addressing Mode</strong></td>
<td><strong>Mnemonics</strong></td>
<td><strong>Opcode</strong></td>
<td><strong>Size In Bytes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accumulator</td>
<td>LSR A</td>
<td>4A</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero Page</td>
<td>LSR Arg</td>
<td>46</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>LSR Arg, X</td>
<td>56</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute</td>
<td>LSR Arg</td>
<td>4E</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute, X</td>
<td>LSR Arg, X</td>
<td>5E</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## NOP
### No Operation

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addressing Mode</strong></td>
<td><strong>Mnemonics</strong></td>
<td><strong>Opcode</strong></td>
<td><strong>Size In Bytes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implied</td>
<td>NOP</td>
<td>EA</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**ORA**

OR Memory With Accumulator

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addressing Mode</strong></td>
<td><strong>Mnemonics</strong></td>
<td><strong>Opcode</strong></td>
<td><strong>Size In Bytes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediate</td>
<td>ORA # Arg</td>
<td>09</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero Page</td>
<td>ORA Arg</td>
<td>05</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>ORA Arg, X</td>
<td>15</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute</td>
<td>ORA Arg</td>
<td>0D</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute, X</td>
<td>ORA Arg, X</td>
<td>1D</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute, Y</td>
<td>ORA Arg, Y</td>
<td>19</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Indirect, X)</td>
<td>ORA (Arg, X)</td>
<td>01</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Indirect), Y</td>
<td>ORA (Arg, Y)</td>
<td>11</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PHA**

Push Accumulator Onto The Stack

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addressing Mode</strong></td>
<td><strong>Mnemonics</strong></td>
<td><strong>Opcode</strong></td>
<td><strong>Size In Bytes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implied</td>
<td>PHA</td>
<td>48</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PHP**

Push Processor Status Onto The Stack

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addressing Mode</strong></td>
<td><strong>Mnemonics</strong></td>
<td><strong>Opcode</strong></td>
<td><strong>Size In Bytes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implied</td>
<td>PHP</td>
<td>08</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### PLA

**Pull Accumulator From The Stack**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied</td>
<td>PLA</td>
<td>68</td>
<td>1</td>
</tr>
</tbody>
</table>

### PLP

**Pull Processor Status From The Stack**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>From Stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied</td>
<td>PLP</td>
<td>28</td>
<td>1</td>
</tr>
</tbody>
</table>

### ROL

**Rotate One Bit Left In Memory Or The Accumulator**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulator</td>
<td>ROL A</td>
<td>2A</td>
<td>1</td>
</tr>
<tr>
<td>Zero Page</td>
<td>ROL Arg</td>
<td>26</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>ROL Arg, X</td>
<td>36</td>
<td>2</td>
</tr>
<tr>
<td>Absolute</td>
<td>ROL Arg</td>
<td>2E</td>
<td>3</td>
</tr>
<tr>
<td>Absolute, X</td>
<td>ROL Arg, X</td>
<td>3E</td>
<td>3</td>
</tr>
</tbody>
</table>
### ROR

**Rotate One Bit Right In Memory Or The Accumulator**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulator</td>
<td>ROR A</td>
<td>6A</td>
<td>1</td>
</tr>
<tr>
<td>Zero Page</td>
<td>ROR Arg</td>
<td>66</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>ROR Arg, X</td>
<td>76</td>
<td>2</td>
</tr>
<tr>
<td>Absolute</td>
<td>ROR Arg</td>
<td>6E</td>
<td>3</td>
</tr>
<tr>
<td>Absolute, X</td>
<td>ROR Arg, X</td>
<td>7E</td>
<td>3</td>
</tr>
</tbody>
</table>

### RTI

**Return From Interrupt**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied</td>
<td>RTI</td>
<td>40</td>
<td>1</td>
</tr>
</tbody>
</table>

### RTS

**Return From Subroutine**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied</td>
<td>RTS</td>
<td>60</td>
<td>1</td>
</tr>
</tbody>
</table>
## SBC

**Subtract Memory From Accumulator, With Borrow**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addressing Mode</strong></td>
<td><strong>Mnemonics</strong></td>
<td><strong>Opcode</strong></td>
<td><strong>Size In Bytes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediate</td>
<td>SBC # Arg</td>
<td>E9</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero Page</td>
<td>SBC Arg</td>
<td>E5</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>SBC Arg, X</td>
<td>F5</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute</td>
<td>SBC Arg</td>
<td>ED</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute, X</td>
<td>SBC Arg, X</td>
<td>FD</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute, Y</td>
<td>SBC Arg, Y</td>
<td>F9</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Indirect, X)</td>
<td>SBC (Arg, X)</td>
<td>E1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Indirect), Y</td>
<td>SBC (Arg), Y</td>
<td>F1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## SEC

**Set Carry Flag**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addressing Mode</strong></td>
<td><strong>Mnemonics</strong></td>
<td><strong>Opcode</strong></td>
<td><strong>Size In Bytes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implied</td>
<td>SEC</td>
<td>38</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## SED

**Set Decimal Mode**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addressing Mode</strong></td>
<td><strong>Mnemonics</strong></td>
<td><strong>Opcode</strong></td>
<td><strong>Size In Bytes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implied</td>
<td>SED</td>
<td>F8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### SEI

**Set Interrupt Disable Status**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>N</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Z</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>I</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>D</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>V</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Mode**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied</td>
<td>SEI</td>
<td>78</td>
<td>1</td>
</tr>
</tbody>
</table>

### STA

**Store Accumulator In Memory**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>N</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Z</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>I</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>D</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>V</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Mode**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero Page</td>
<td>STA Arg</td>
<td>85</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>STA Arg, X</td>
<td>95</td>
<td>2</td>
</tr>
<tr>
<td>Absolute</td>
<td>STA Arg</td>
<td>8D</td>
<td>3</td>
</tr>
<tr>
<td>Absolute, X</td>
<td>STA Arg, X</td>
<td>9D</td>
<td>3</td>
</tr>
<tr>
<td>Absolute, Y</td>
<td>STA Arg, Y</td>
<td>99</td>
<td>3</td>
</tr>
<tr>
<td>(Indirect, X)</td>
<td>STA (Arg, X)</td>
<td>81</td>
<td>2</td>
</tr>
<tr>
<td>(Indirect), Y</td>
<td>STA (Arg), Y</td>
<td>91</td>
<td>2</td>
</tr>
</tbody>
</table>

### STX

**Store X Register In Memory**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>N</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Z</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>I</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>D</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>V</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Mode**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero Page</td>
<td>STX Arg</td>
<td>86</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page, Y</td>
<td>STX Arg, Y</td>
<td>96</td>
<td>2</td>
</tr>
<tr>
<td>Absolute</td>
<td>STX Arg</td>
<td>8E</td>
<td>3</td>
</tr>
</tbody>
</table>
### STY
**Status Flags**
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
</table>

#### Addressing Mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero Page</td>
<td>STY Arg</td>
<td>84</td>
<td>2</td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>STY Arg, X</td>
<td>94</td>
<td>2</td>
</tr>
<tr>
<td>Absolute</td>
<td>STY Arg</td>
<td>8C</td>
<td>3</td>
</tr>
</tbody>
</table>

### TAX
**Status Flags**
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
</table>

#### Addressing Mode

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied</td>
<td>TAX</td>
<td>AA</td>
</tr>
</tbody>
</table>

### TAY
**Status Flags**
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
</table>

#### Addressing Mode

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied</td>
<td>TAY</td>
<td>A8</td>
</tr>
</tbody>
</table>

### TSX
**Status Flags**
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
</table>

#### Addressing Mode

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Opcode</th>
<th>Size In Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied</td>
<td>TSX</td>
<td>BA</td>
</tr>
</tbody>
</table>

---

165
### TXA

**Transfer X Register To Accumulator**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addressing Mode</td>
<td>Mnemonics</td>
<td>Opcode</td>
<td>Size In Bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implied</td>
<td>TXA</td>
<td>8A</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### TXS

**Transfer X Register To Stack Pointer**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addressing Mode</td>
<td>Mnemonics</td>
<td>Opcode</td>
<td>Size In Bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implied</td>
<td>TXS</td>
<td>9A</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### TYA

**Transfer Y Register To Accumulator**

<table>
<thead>
<tr>
<th>Status Flags</th>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addressing Mode</td>
<td>Mnemonics</td>
<td>Opcode</td>
<td>Size In Bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implied</td>
<td>TYA</td>
<td>98</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix B

These maps, primarily the work of Jim Butterfield, all originally appeared in COMPUTE! Magazine. (See the copyright page for references.)

Map I. PET Original And Upgrade BASIC.

<table>
<thead>
<tr>
<th>ORIG</th>
<th>UPGR</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>C357</td>
<td>C355</td>
<td>?OUT OF MEMORY</td>
</tr>
<tr>
<td>C359</td>
<td>C357</td>
<td>Send BASIC error message</td>
</tr>
<tr>
<td>C38B</td>
<td>C389</td>
<td>Warm start, BASIC</td>
</tr>
<tr>
<td>C3AC</td>
<td>C3AB</td>
<td>Crunch &amp; insert line</td>
</tr>
<tr>
<td>C430</td>
<td>C439</td>
<td>Fix chaining &amp; READY</td>
</tr>
<tr>
<td>C433</td>
<td>C442</td>
<td>Fix chaining</td>
</tr>
<tr>
<td>C48D</td>
<td>C495</td>
<td>Crunch tokens</td>
</tr>
<tr>
<td>C522</td>
<td>C52C</td>
<td>Find line in BASIC</td>
</tr>
<tr>
<td>C553</td>
<td>C55D</td>
<td>Do NEW</td>
</tr>
<tr>
<td>C56A</td>
<td>C572</td>
<td>Do CLR</td>
</tr>
<tr>
<td>C59A</td>
<td>C5A7</td>
<td>Reset BASIC to start</td>
</tr>
<tr>
<td>C6B5</td>
<td>C6C4</td>
<td>Continue BASIC execution</td>
</tr>
<tr>
<td>C6C4</td>
<td>C6C4</td>
<td>Continue BASIC execution</td>
</tr>
<tr>
<td>C863</td>
<td>C873</td>
<td>Get fixed-point number from BASIC</td>
</tr>
<tr>
<td>C9CE</td>
<td>C9DE</td>
<td>Send Return, LF if in screen mode</td>
</tr>
<tr>
<td>C9D2</td>
<td>C9E2</td>
<td>Send Return, Linefeed</td>
</tr>
<tr>
<td>CA27</td>
<td>CA1C</td>
<td>Print string</td>
</tr>
<tr>
<td>CA2D</td>
<td>CA22</td>
<td>Print precomputed string</td>
</tr>
<tr>
<td>CA49</td>
<td>CA45</td>
<td>Print character</td>
</tr>
<tr>
<td>CE11</td>
<td>CDF8</td>
<td>Check for comma</td>
</tr>
<tr>
<td>CE13</td>
<td>CDF8</td>
<td>Check for specific character</td>
</tr>
<tr>
<td>CE1C</td>
<td>CE03</td>
<td>‘SYNTAX ERROR’</td>
</tr>
<tr>
<td>D079</td>
<td>D069</td>
<td>Bump Variable Address by 2</td>
</tr>
<tr>
<td>D0A7</td>
<td>D09A</td>
<td>Float to Fixed conversion</td>
</tr>
<tr>
<td>D278</td>
<td>D26D</td>
<td>Fixed to Float conversion</td>
</tr>
<tr>
<td>D679</td>
<td>D67B</td>
<td>Get byte to X reg</td>
</tr>
<tr>
<td>D68D</td>
<td>D68F</td>
<td>Evaluate String</td>
</tr>
<tr>
<td>D6C4</td>
<td>D6C6</td>
<td>Get two parameters</td>
</tr>
<tr>
<td>D73C</td>
<td>D773</td>
<td>Add (from memory)</td>
</tr>
<tr>
<td>D8FD</td>
<td>D934</td>
<td>Multiply by memory location</td>
</tr>
<tr>
<td>D9B4</td>
<td>D9EE</td>
<td>Multiply by ten</td>
</tr>
<tr>
<td>DA74</td>
<td>DAAE</td>
<td>Unpack memory variable to Accum #1</td>
</tr>
<tr>
<td>DB1B</td>
<td>DB55</td>
<td>Completion of Fixed to Float conversion</td>
</tr>
<tr>
<td>DC9F</td>
<td>DCD9</td>
<td>Print fixed-point value</td>
</tr>
<tr>
<td>DCA9</td>
<td>DCE3</td>
<td>Print floating-point value</td>
</tr>
<tr>
<td>Code</td>
<td>Meaning</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>DCAF</td>
<td>Convert number to ASCII string</td>
<td></td>
</tr>
<tr>
<td>E3EA</td>
<td>Print a character</td>
<td></td>
</tr>
<tr>
<td>na</td>
<td>Output byte as 2 hex digits</td>
<td></td>
</tr>
<tr>
<td>na</td>
<td>Input 2 hex digits to A</td>
<td></td>
</tr>
<tr>
<td>na</td>
<td>Input 1 hex digit to A</td>
<td></td>
</tr>
<tr>
<td>F0B6</td>
<td>Send ‘talk’ to IEEE</td>
<td></td>
</tr>
<tr>
<td>F0BA</td>
<td>Send ‘listen’ to IEEE</td>
<td></td>
</tr>
<tr>
<td>F12C</td>
<td>Send Secondary Address</td>
<td></td>
</tr>
<tr>
<td>F156</td>
<td>Send canned message</td>
<td></td>
</tr>
<tr>
<td>F167</td>
<td>Send character to IEEE</td>
<td></td>
</tr>
<tr>
<td>F17A</td>
<td>Send ‘untalk’</td>
<td></td>
</tr>
<tr>
<td>F17E</td>
<td>Send ‘unlisten’</td>
<td></td>
</tr>
<tr>
<td>F187</td>
<td>Input from IEEE</td>
<td></td>
</tr>
<tr>
<td>F2C8</td>
<td>Close logical file</td>
<td></td>
</tr>
<tr>
<td>F2CD</td>
<td>Close logical file in A</td>
<td></td>
</tr>
<tr>
<td>F32A</td>
<td>Check for Stop key</td>
<td></td>
</tr>
<tr>
<td>F33F</td>
<td>Send message if Direct mode</td>
<td></td>
</tr>
<tr>
<td>na</td>
<td>LOAD subroutine</td>
<td></td>
</tr>
<tr>
<td>F3DB</td>
<td>?LOAD ERROR</td>
<td></td>
</tr>
<tr>
<td>F3E5</td>
<td>Print READY &amp; reset BASIC to start</td>
<td></td>
</tr>
<tr>
<td>F3FF</td>
<td>Print SEARCHING . .</td>
<td></td>
</tr>
<tr>
<td>F411</td>
<td>Print file name</td>
<td></td>
</tr>
<tr>
<td>F43F</td>
<td>Get LOAD/SAVE type parameters</td>
<td></td>
</tr>
<tr>
<td>F462</td>
<td>Open IEEE channel for output</td>
<td></td>
</tr>
<tr>
<td>F495</td>
<td>Find specific tape header block</td>
<td></td>
</tr>
<tr>
<td>F504</td>
<td>Get string</td>
<td></td>
</tr>
<tr>
<td>F52A</td>
<td>Open logical file from input parameters</td>
<td></td>
</tr>
<tr>
<td>F52D</td>
<td>Open logical file</td>
<td></td>
</tr>
<tr>
<td>F579</td>
<td>?FILE NOT FOUND, clear I/O</td>
<td></td>
</tr>
<tr>
<td>F57B</td>
<td>Send error message</td>
<td></td>
</tr>
<tr>
<td>F5AE</td>
<td>Find any tape header block</td>
<td></td>
</tr>
<tr>
<td>F64D</td>
<td>Get pointers for tape LOAD</td>
<td></td>
</tr>
<tr>
<td>F667</td>
<td>Set tape buffer start address</td>
<td></td>
</tr>
<tr>
<td>F67D</td>
<td>Set cassette buffer pointers</td>
<td></td>
</tr>
<tr>
<td>F6E6</td>
<td>Close IEEE channel</td>
<td></td>
</tr>
<tr>
<td>F78B</td>
<td>Set input device from logical file number</td>
<td></td>
</tr>
<tr>
<td>F7DC</td>
<td>Set output device from LFN</td>
<td></td>
</tr>
<tr>
<td>F83B</td>
<td>PRESS PLAY . . ; wait</td>
<td></td>
</tr>
<tr>
<td>F87F</td>
<td>Read tape to buffer</td>
<td></td>
</tr>
<tr>
<td>F88A</td>
<td>Read tape</td>
<td></td>
</tr>
<tr>
<td>F89B</td>
<td>Write tape from buffer</td>
<td></td>
</tr>
<tr>
<td>F8C1</td>
<td>Write tape, leader length in A</td>
<td></td>
</tr>
<tr>
<td>F913</td>
<td>Wait for I/O complete or Stop key</td>
<td></td>
</tr>
<tr>
<td>FBDC</td>
<td>Reset tape I/O pointer</td>
<td></td>
</tr>
<tr>
<td>FD1B</td>
<td>Set interrupt vector</td>
<td></td>
</tr>
</tbody>
</table>
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
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
<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0042-0043</td>
<td>Current variable name</td>
</tr>
<tr>
<td>0044-0045</td>
<td>Current variable address</td>
</tr>
<tr>
<td>0046-0047</td>
<td>Variable pointer for FOR/NEXT</td>
</tr>
<tr>
<td>0048</td>
<td>Y save register; new-operator save</td>
</tr>
<tr>
<td>004A</td>
<td>Comparison symbol accumulator</td>
</tr>
<tr>
<td>004B-004C</td>
<td>Misc. numeric work area</td>
</tr>
<tr>
<td>004D-0050</td>
<td>Work area; garbage yardstick</td>
</tr>
<tr>
<td>0051-0053</td>
<td>Jump vector for functions</td>
</tr>
<tr>
<td>0054-0058</td>
<td>Misc. numeric storage area</td>
</tr>
<tr>
<td>0059-005D</td>
<td>Misc. numeric storage area</td>
</tr>
<tr>
<td>005E-0063</td>
<td>Accumulator#1:E,M,M,M,M,S</td>
</tr>
<tr>
<td>0064</td>
<td>Series evaluation constant pointer</td>
</tr>
<tr>
<td>0065</td>
<td>Accumulator hi-order propagation word</td>
</tr>
<tr>
<td>0066-006B</td>
<td>Accumulator #2</td>
</tr>
<tr>
<td>006C</td>
<td>Sign comparison, primary vs. secondary</td>
</tr>
<tr>
<td>006D</td>
<td>low-order rounding byte for Acc #1</td>
</tr>
<tr>
<td>006E-006F</td>
<td>Cassette buffer length/Series pointer</td>
</tr>
<tr>
<td>0070-0087</td>
<td>Subrtn: Get BASIC Char; 77,78 = pointer</td>
</tr>
<tr>
<td>0088-008C</td>
<td>RND storage and work area</td>
</tr>
<tr>
<td>008D-008F</td>
<td>Jiffy clock for TI and TI$</td>
</tr>
<tr>
<td>0090-0091</td>
<td>Hardware interrupt vector</td>
</tr>
<tr>
<td>0092-0093</td>
<td>Break interrupt vector</td>
</tr>
<tr>
<td>0094-0095</td>
<td>NMI interrupt vector</td>
</tr>
<tr>
<td>0096</td>
<td>Status word ST</td>
</tr>
<tr>
<td>0097</td>
<td>Which key depressed: 255 = no key</td>
</tr>
<tr>
<td>0098</td>
<td>Shift key: 1 if depressed</td>
</tr>
<tr>
<td>0099-009A</td>
<td>Correction clock</td>
</tr>
<tr>
<td>009B</td>
<td>Keyswitch PIA: STOP and RVS flags</td>
</tr>
<tr>
<td>009C</td>
<td>Timing constant buffer</td>
</tr>
<tr>
<td>009D</td>
<td>Load = 0, Verify = 1</td>
</tr>
<tr>
<td>009E</td>
<td>$characters in keyboard buffer</td>
</tr>
<tr>
<td>009F</td>
<td>Screen reverse flag</td>
</tr>
<tr>
<td>00A0</td>
<td>IEEE-488 mode</td>
</tr>
<tr>
<td>00A1</td>
<td>End-of-line-for-input pointer</td>
</tr>
<tr>
<td>00A3-00A4</td>
<td>Cursor log (row, column)</td>
</tr>
<tr>
<td>00A5</td>
<td>PBD image for tape I/O</td>
</tr>
<tr>
<td>00A6</td>
<td>Key image</td>
</tr>
<tr>
<td>00A7</td>
<td>0 = flashing cursor, else no cursor</td>
</tr>
<tr>
<td>00A8</td>
<td>Countdown for cursor timing</td>
</tr>
<tr>
<td>00A9</td>
<td>Character under cursor</td>
</tr>
<tr>
<td>00AA</td>
<td>Cursor blink flag</td>
</tr>
<tr>
<td>00AB</td>
<td>EOT bit received</td>
</tr>
<tr>
<td>00AC</td>
<td>Input from screen/input from keyboard</td>
</tr>
<tr>
<td>00AD</td>
<td>X save flag</td>
</tr>
<tr>
<td>00AE</td>
<td>How many open files</td>
</tr>
</tbody>
</table>
### Appendix B

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00AF</td>
<td>175</td>
</tr>
<tr>
<td>00B0</td>
<td>176</td>
</tr>
<tr>
<td>00B1</td>
<td>177</td>
</tr>
<tr>
<td>00B2</td>
<td>178</td>
</tr>
<tr>
<td>00B4</td>
<td>180</td>
</tr>
<tr>
<td>00B5</td>
<td>181</td>
</tr>
<tr>
<td>00B7</td>
<td>183</td>
</tr>
<tr>
<td>00B9</td>
<td>185</td>
</tr>
<tr>
<td>00BA</td>
<td>186</td>
</tr>
<tr>
<td>00BB</td>
<td>187</td>
</tr>
<tr>
<td>00BC</td>
<td>188</td>
</tr>
<tr>
<td>00BD</td>
<td>189</td>
</tr>
<tr>
<td>00BE</td>
<td>190</td>
</tr>
<tr>
<td>00BF</td>
<td>191</td>
</tr>
<tr>
<td>00C0</td>
<td>192</td>
</tr>
<tr>
<td>00C1</td>
<td>193</td>
</tr>
<tr>
<td>00C2</td>
<td>194</td>
</tr>
<tr>
<td>00C3</td>
<td>195</td>
</tr>
<tr>
<td>00C4-00C5</td>
<td>196-197</td>
</tr>
<tr>
<td>00C6</td>
<td>198</td>
</tr>
<tr>
<td>00C7-00C8</td>
<td>199-200</td>
</tr>
<tr>
<td>00C9-00CA</td>
<td>201-202</td>
</tr>
<tr>
<td>00CB-00CC</td>
<td>203-204</td>
</tr>
<tr>
<td>00CD</td>
<td>205</td>
</tr>
<tr>
<td>00CE</td>
<td>206</td>
</tr>
<tr>
<td>00CF</td>
<td>207</td>
</tr>
<tr>
<td>00D0</td>
<td>208</td>
</tr>
<tr>
<td>00D1</td>
<td>209</td>
</tr>
<tr>
<td>00D2</td>
<td>210</td>
</tr>
<tr>
<td>00D3</td>
<td>211</td>
</tr>
<tr>
<td>00D4</td>
<td>212</td>
</tr>
<tr>
<td>00D5</td>
<td>213</td>
</tr>
<tr>
<td>00D6-00D7</td>
<td>214-215</td>
</tr>
<tr>
<td>00D8</td>
<td>216</td>
</tr>
<tr>
<td>00D9</td>
<td>217</td>
</tr>
<tr>
<td>00DA-00DB</td>
<td>218-219</td>
</tr>
<tr>
<td>00DC</td>
<td>220</td>
</tr>
<tr>
<td>00DD</td>
<td>221</td>
</tr>
<tr>
<td>00DE</td>
<td>222</td>
</tr>
<tr>
<td>00DF</td>
<td>223</td>
</tr>
<tr>
<td>00E0-00F8</td>
<td>224-248</td>
</tr>
<tr>
<td>00F9</td>
<td>249</td>
</tr>
<tr>
<td>00FA</td>
<td>250</td>
</tr>
<tr>
<td>00FB-00FC</td>
<td>251-252</td>
</tr>
<tr>
<td>0100-010A</td>
<td>256-266</td>
</tr>
</tbody>
</table>

- **Input device**, normally 0
- **Output CMD device**, normally 3
- **Tape character parity**
- **Byte received flag**
- **Tape buffer character**
- **Pointer in file name transfer**
- **Serial bit count**
- **Cycle counter**
- **Countdown for tape write**
- **Tape buffer #1 count**
- **Tape buffer #2 count**
- **Write leader count; Read pass 1/pas 2**
- **Write new byte; Read error flag**
- **Write start bit; Read bit seq error**
- **Pass 1 error log pointer**
- **Pass 2 error correction pointer**
- **0 = Scan; 1-15 = Count; $40 = Load; $80 = End**
- **Checksum**
- **Pointer to screen line**
- **Position of cursor on above line**
- **Utility pointer: tape buffer, scrolling**
- **Tape end address/end of current program**
- **Tape timing constants**
- **$0 = direct cursor, else programmed cursor**
- **Timer 1 enabled for tape read; 00 = disabled**
- **EOT signal received from tape**
- **Read character error**
- **# characters in file name**
- **Current logical file number**
- **Current secondary addr, or R/W command**
- **Current device number**
- **Line length (40 or 80) for screen**
- **Start of tape buffer, address**
- **Line where cursor lives**
- **Last key input; buffer checksum; bit buffer**
- **File name pointer**
- **Number of keyboard INSERTs outstanding**
- **Write shift word/Receive input character**
- **# blocks remaining to write/read**
- **Serial word buffer**
- **Screen line table: hi order address & line wrap**
- **Cassette #1 status switch**
- **Cassette #2 status switch**
- **Tape start address**
- **Binary to ASCII conversion area**

---

171
### Appendix B

- **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.

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000-0002</td>
<td>0-2</td>
<td>USR jump</td>
</tr>
<tr>
<td>0003</td>
<td>3</td>
<td>Search character</td>
</tr>
<tr>
<td>0004</td>
<td>4</td>
<td>Scan-between-quotes flag</td>
</tr>
<tr>
<td>0005</td>
<td>5</td>
<td>Input buffer pointer; # of subscripts</td>
</tr>
<tr>
<td>0006</td>
<td>6</td>
<td>Default DIM flag</td>
</tr>
<tr>
<td>0007</td>
<td>7</td>
<td>Type: FF=string, 00=numeric</td>
</tr>
<tr>
<td>0008</td>
<td>8</td>
<td>Type: 80=integer, 00=floating point</td>
</tr>
<tr>
<td>0009</td>
<td>9</td>
<td>Flag: DATA scan; LIST quote; memory</td>
</tr>
<tr>
<td>000A</td>
<td>10</td>
<td>Subscript flag; FNX flag</td>
</tr>
<tr>
<td>000B</td>
<td>11</td>
<td>0=INPUT; $40=GET; $98=READ</td>
</tr>
<tr>
<td>000C</td>
<td>12</td>
<td>ATN sign/Comparison Evaluation flag</td>
</tr>
<tr>
<td>000D-000F</td>
<td>13-15</td>
<td>Disk status DS$ descriptor</td>
</tr>
<tr>
<td>0010</td>
<td>16</td>
<td>Current I/O device for prompt-suppress</td>
</tr>
<tr>
<td>0011-0012</td>
<td>17-18</td>
<td>Integer value (for SYS, GOTO etc)</td>
</tr>
<tr>
<td>0013-0015</td>
<td>19-21</td>
<td>Pointers for descriptor stack</td>
</tr>
<tr>
<td>0016-001E</td>
<td>22-30</td>
<td>Descriptor stack(temp strings)</td>
</tr>
<tr>
<td>001F-0022</td>
<td>31-34</td>
<td>Utility pointer area</td>
</tr>
<tr>
<td>0023-0027</td>
<td>35-39</td>
<td>Product area for multiplication</td>
</tr>
<tr>
<td>0028-0029</td>
<td>40-41</td>
<td>Pointer: Start-of-Basic</td>
</tr>
<tr>
<td>002A-002B</td>
<td>42-43</td>
<td>Pointer: Start-of-Variables</td>
</tr>
<tr>
<td>002C-002D</td>
<td>44-45</td>
<td>Pointer: Start-of-Arrays</td>
</tr>
<tr>
<td>002E-002F</td>
<td>46-47</td>
<td>Pointer: End-of-Arrays</td>
</tr>
<tr>
<td>0030-0031</td>
<td>48-49</td>
<td>Pointer: String-storage(moving down)</td>
</tr>
<tr>
<td>0032-0033</td>
<td>50-51</td>
<td>Utility string pointer</td>
</tr>
<tr>
<td>0034-0035</td>
<td>52-53</td>
<td>Pointer: Limit-of-memory</td>
</tr>
<tr>
<td>0036-0037</td>
<td>54-55</td>
<td>Current Basic line number</td>
</tr>
<tr>
<td>0038-0039</td>
<td>56-57</td>
<td>Previous Basic line number</td>
</tr>
<tr>
<td>003A-003B</td>
<td>58-59</td>
<td>Pointer: Basic statement for CONT</td>
</tr>
<tr>
<td>Address</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>003C-003D</td>
<td>Current DATA line number</td>
<td></td>
</tr>
<tr>
<td>003E-003F</td>
<td>Current DATA address</td>
<td></td>
</tr>
<tr>
<td>0040-0041</td>
<td>Input vector</td>
<td></td>
</tr>
<tr>
<td>0042-0043</td>
<td>Current variable name</td>
<td></td>
</tr>
<tr>
<td>0044-0045</td>
<td>Current variable address</td>
<td></td>
</tr>
<tr>
<td>0046-0047</td>
<td>Variable pointer for FOR/NEXT</td>
<td></td>
</tr>
<tr>
<td>0048-0049</td>
<td>Y-save; op-save; Basic pointer save</td>
<td></td>
</tr>
<tr>
<td>004A</td>
<td>Comparison symbol accumulator</td>
<td></td>
</tr>
<tr>
<td>004B-0050</td>
<td>Misc work area, pointers, etc.</td>
<td></td>
</tr>
<tr>
<td>0051-0053</td>
<td>Jump vector for functions</td>
<td></td>
</tr>
<tr>
<td>0054-005D</td>
<td>Misc numeric work area</td>
<td></td>
</tr>
<tr>
<td>005E</td>
<td>Accum#1: Exponent</td>
<td></td>
</tr>
<tr>
<td>005F-0062</td>
<td>Accum#1: Mantissa</td>
<td></td>
</tr>
<tr>
<td>0063</td>
<td>Accum#1: Sign</td>
<td></td>
</tr>
<tr>
<td>0064</td>
<td>Series evaluation constant pointer</td>
<td></td>
</tr>
<tr>
<td>0065</td>
<td>Accum#1 hi-order (overflow)</td>
<td></td>
</tr>
<tr>
<td>0066-006B</td>
<td>Accum#2: Exponent, etc.</td>
<td></td>
</tr>
<tr>
<td>006C</td>
<td>Sign comparison, Acc#1 vs #2</td>
<td></td>
</tr>
<tr>
<td>006D</td>
<td>Accum#1 lo-order (rounding)</td>
<td></td>
</tr>
<tr>
<td>006E-006F</td>
<td>Cassette buff len/Series pointer</td>
<td></td>
</tr>
<tr>
<td>0070-0087</td>
<td>CHRGET subroutine; get Basic char</td>
<td></td>
</tr>
<tr>
<td>0077-0078</td>
<td>Basic pointer (within subrtn)</td>
<td></td>
</tr>
<tr>
<td>0088-008C</td>
<td>Random number seed.</td>
<td></td>
</tr>
<tr>
<td>008D-008F</td>
<td>Jiffy clock for TI and TI§</td>
<td></td>
</tr>
<tr>
<td>0090-0091</td>
<td>Hardware interrupt vector</td>
<td></td>
</tr>
<tr>
<td>0092-0093</td>
<td>BRK interrupt vector</td>
<td></td>
</tr>
<tr>
<td>0094-0095</td>
<td>NMI interrupt vector</td>
<td></td>
</tr>
<tr>
<td>0096</td>
<td>Status word ST</td>
<td></td>
</tr>
<tr>
<td>0097</td>
<td>Which key down; 255=no key</td>
<td></td>
</tr>
<tr>
<td>0098</td>
<td>Shift key: 1 if depressed</td>
<td></td>
</tr>
<tr>
<td>0099-009A</td>
<td>Correction clock</td>
<td></td>
</tr>
<tr>
<td>009B</td>
<td>Keyswitch PIA: STOP and RVS flags</td>
<td></td>
</tr>
<tr>
<td>009C</td>
<td>Timing constant for tape</td>
<td></td>
</tr>
<tr>
<td>009D</td>
<td>Load=0, Verify=1</td>
<td></td>
</tr>
<tr>
<td>009E</td>
<td>Number of characters in keybd buffer</td>
<td></td>
</tr>
<tr>
<td>009F</td>
<td>Screen reverse flag</td>
<td></td>
</tr>
<tr>
<td>00A0</td>
<td>IEEE output; 255=character pending</td>
<td></td>
</tr>
<tr>
<td>00A1</td>
<td>End-of-line-for-input pointer</td>
<td></td>
</tr>
<tr>
<td>00A3-00A4</td>
<td>Cursor log (row, column)</td>
<td></td>
</tr>
<tr>
<td>00A5</td>
<td>IEEE output buffer</td>
<td></td>
</tr>
<tr>
<td>00A6</td>
<td>Key image</td>
<td></td>
</tr>
<tr>
<td>00A7</td>
<td>0=flash cursor</td>
<td></td>
</tr>
<tr>
<td>00A8</td>
<td>Cursor timing countdown</td>
<td></td>
</tr>
<tr>
<td>00A9</td>
<td>Character under cursor</td>
<td></td>
</tr>
<tr>
<td>00AA</td>
<td>Cursor in blink phase</td>
<td></td>
</tr>
<tr>
<td>00AB</td>
<td>EOT received from tape</td>
<td></td>
</tr>
<tr>
<td>00AC</td>
<td>Input from screen/from keyboard</td>
<td></td>
</tr>
<tr>
<td>00AD</td>
<td>X save</td>
<td></td>
</tr>
<tr>
<td>00AE</td>
<td>How many open files</td>
<td></td>
</tr>
<tr>
<td>00AF</td>
<td>Input device, normally 0</td>
<td></td>
</tr>
<tr>
<td>00B0</td>
<td>Output CMD device, normally 3</td>
<td></td>
</tr>
<tr>
<td>00B1</td>
<td>Tape character parity</td>
<td></td>
</tr>
<tr>
<td>00B2</td>
<td>Byte received flag</td>
<td></td>
</tr>
<tr>
<td>00B3</td>
<td>Logical Address temporary save</td>
<td></td>
</tr>
<tr>
<td>00B4</td>
<td>Tape buffer character; MLM command</td>
<td></td>
</tr>
<tr>
<td>00B5</td>
<td>File name pointer; MLM flag, counter</td>
<td></td>
</tr>
<tr>
<td>00B7</td>
<td>Serial bit count</td>
<td></td>
</tr>
<tr>
<td>00B9</td>
<td>Cycle counter</td>
<td></td>
</tr>
</tbody>
</table>
### Appendix B

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00BA</td>
<td>186</td>
<td>Tape writer countdown</td>
</tr>
<tr>
<td>00BB-00BC</td>
<td>187-188</td>
<td>Tape buffer pointers, #1 and #2</td>
</tr>
<tr>
<td>00BD</td>
<td>189</td>
<td>Write leader count; read pass 1/2</td>
</tr>
<tr>
<td>00BE</td>
<td>190</td>
<td>Write new byte; read error flag</td>
</tr>
<tr>
<td>00BF</td>
<td>191</td>
<td>Write start bit; read bit seq error</td>
</tr>
<tr>
<td>00C0-00C1</td>
<td>192-193</td>
<td>Error log pointers, pass 1/2</td>
</tr>
<tr>
<td>00C2</td>
<td>194</td>
<td>0=Scan/1-15=Count/$40=Load/$80=End</td>
</tr>
<tr>
<td>00C3</td>
<td>195</td>
<td>Write leader length; read checksum</td>
</tr>
<tr>
<td>00C4-00C5</td>
<td>196-197</td>
<td>Pointer to screen line</td>
</tr>
<tr>
<td>00C6</td>
<td>198</td>
<td>Position of cursor on above line</td>
</tr>
<tr>
<td>00C7-00C8</td>
<td>199-200</td>
<td>Utility pointer: tape, scroll</td>
</tr>
<tr>
<td>00C9-00CA</td>
<td>201-202</td>
<td>Tape end addr/End of current program</td>
</tr>
<tr>
<td>00CB-00CC</td>
<td>203-204</td>
<td>Tape timing constants</td>
</tr>
<tr>
<td>00CD</td>
<td>205</td>
<td>0=direct cursor, else programmed</td>
</tr>
<tr>
<td>00CE</td>
<td>206</td>
<td>Tape read timer 1 enabled</td>
</tr>
<tr>
<td>00CF</td>
<td>207</td>
<td>EOT received from tape</td>
</tr>
<tr>
<td>00DD</td>
<td>208</td>
<td>Read character error</td>
</tr>
<tr>
<td>00D1</td>
<td>209</td>
<td># characters in file name</td>
</tr>
<tr>
<td>00D2</td>
<td>210</td>
<td>Current file logical address</td>
</tr>
<tr>
<td>00D3</td>
<td>211</td>
<td>Current file secondary addr</td>
</tr>
<tr>
<td>00D4</td>
<td>212</td>
<td>Current file device number</td>
</tr>
<tr>
<td>00D5</td>
<td>213</td>
<td>Right-hand window or line margin</td>
</tr>
<tr>
<td>00D6-00D7</td>
<td>214-215</td>
<td>Pointer: Start of tape buffer</td>
</tr>
<tr>
<td>00DB</td>
<td>216</td>
<td>Line where cursor lives</td>
</tr>
<tr>
<td>00D9</td>
<td>217</td>
<td>Last key/checksum/misc.</td>
</tr>
<tr>
<td>00DA-00DB</td>
<td>218-219</td>
<td>File name pointer</td>
</tr>
<tr>
<td>00DC</td>
<td>220</td>
<td>Number of INSERTs outstanding</td>
</tr>
<tr>
<td>00DD</td>
<td>221</td>
<td>Write shift word/read character in</td>
</tr>
<tr>
<td>00DE</td>
<td>222</td>
<td>Tape blocks remaining to write/read</td>
</tr>
<tr>
<td>00DF</td>
<td>223</td>
<td>Serial word buffer</td>
</tr>
<tr>
<td>00E0-00F8</td>
<td>224-248</td>
<td>(40-column) Screen line wrap table</td>
</tr>
<tr>
<td>00E0-00E1</td>
<td>224-225</td>
<td>(80-column) Top, bottom of window</td>
</tr>
<tr>
<td>00E2</td>
<td>226</td>
<td>(80-column) Left window margin</td>
</tr>
<tr>
<td>00E3</td>
<td>227</td>
<td>(80-column) Limit of keybd buffer</td>
</tr>
<tr>
<td>00E4</td>
<td>228</td>
<td>(80-column) Key repeat flag</td>
</tr>
<tr>
<td>00E5</td>
<td>229</td>
<td>(80-column) Repeat countdown</td>
</tr>
<tr>
<td>00E6</td>
<td>230</td>
<td>(80-column) New key marker</td>
</tr>
<tr>
<td>00E7</td>
<td>231</td>
<td>(80-column) Chime time</td>
</tr>
<tr>
<td>00E8</td>
<td>232</td>
<td>(80-column) HOME count</td>
</tr>
<tr>
<td>00E9-00EA</td>
<td>233-234</td>
<td>(80-column) Input vector</td>
</tr>
<tr>
<td>00EB-00EC</td>
<td>235-236</td>
<td>(80-column) Output vector</td>
</tr>
<tr>
<td>00F9-00FA</td>
<td>249-250</td>
<td>Cassette status, #1 and #2</td>
</tr>
<tr>
<td>00FB-00FC</td>
<td>251-252</td>
<td>MLM pointer/Tape start address</td>
</tr>
<tr>
<td>00FD-00FE</td>
<td>253-254</td>
<td>MLM, DOS pointer, misc.</td>
</tr>
<tr>
<td>0100-010A</td>
<td>256-266</td>
<td>STR$ work area, MLM work</td>
</tr>
<tr>
<td>0100-013E</td>
<td>256-318</td>
<td>Tape read error log</td>
</tr>
<tr>
<td>0100-01FF</td>
<td>256-511</td>
<td>Processor stack</td>
</tr>
<tr>
<td>0200-0250</td>
<td>512-592</td>
<td>MLM work area; Input buffer</td>
</tr>
<tr>
<td>0251-025A</td>
<td>593-602</td>
<td>File logical address table</td>
</tr>
<tr>
<td>025B-0264</td>
<td>603-612</td>
<td>File device number table</td>
</tr>
<tr>
<td>0265-026E</td>
<td>613-622</td>
<td>File secondary adds table</td>
</tr>
<tr>
<td>026F-0278</td>
<td>623-632</td>
<td>Keyboard input buffer</td>
</tr>
<tr>
<td>027A-0339</td>
<td>634-825</td>
<td>Tape#1 input buffer</td>
</tr>
<tr>
<td>033A-03F9</td>
<td>826-1017</td>
<td>Tape#2 input buffer</td>
</tr>
<tr>
<td>033A</td>
<td>826</td>
<td>DOS character pointer</td>
</tr>
<tr>
<td>033B</td>
<td>827</td>
<td>DOS drive 1 flag</td>
</tr>
<tr>
<td>033C</td>
<td>828</td>
<td>DOS drive 2 flag</td>
</tr>
<tr>
<td>033D</td>
<td>829</td>
<td>DOS length/write flag</td>
</tr>
</tbody>
</table>

174
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-B44B5 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:
Appendix B

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-BA1A Perform CMD
BAA2-BBA1C Perform PRINT
BB1D-BBB39 Print string from memory
BB3A-BB4B Print single format character
BB4C-BBB79 Handle bad input data
BB7A-BBA3 Perform GET
BBA4-BBBBD 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
C1CO-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-C7BB Concatenate
C78C-C7BB Store string
C7B5-C810 Discard unwanted string
Appendix B

Clean descriptor stack
Perform CHR$
Perform LEFT$
Perform RIGHT$
Perform lHD$
Pull string data
Perform LEN
Switch string to numeric
Perform ASC
Get byte parameter
Perform VAL
Get two parameters for POKE or WAIT
Convert floating-to-fixed
Perform PEEK
Perf or m POKE
Perform WAIT
Add 0.5
Perform subtraction
Perform addition
Complement accum#1
Overflow exit
I<lul tiply-a-byte
Constants
Perform LOG
Perform multiplication
Unpack memory into accum#2
Test & adjust accumula tors
Handle overflow and underflow
Mul tiply by 10
10 in floating binary
Divide by 10
Perform divid e -by
Perform divide -i nto
Unpack memory into accum#1
Pack accum#1 into memory
Move accum#2 to #1
Move accum#1 to #2
Round accumll1
Get 2ccum!t1 sign
Perform SGN
Perform ABS
Compare accum#1 to memory
Floating-to-fixed
Perform INT
Convert string to floating-point
Get new ASCII digit
Con stan ts
Print IN, then:
cn8
CF7F-CF92 Print Basic line #
CF93-DOC6 Convert floating-point to ASC I I
DOC7 -D1 07 Constants
Perform SQR
D10 8

CB11-CB21
CB22-CB35
CB36-CB61
CB62-CB6C
CB6D-CB96
CB97-CBBl
CBB2-CBB7
CB BB-CBCO
CBC1-C8DO
CBD1-CBE2
CBE3-C920
C921-C92C
C92D -C942
C943-C959
C95A-C962
C963-C97E
C97F-C9B5
C9B6
C99B-CA7C
CA7D-CAB3
CAB4-CABB
CAB9-CAFl
CAF2-CB 1F
CB20
CB5E-CBC l
CBC2-CBEC
CBED-CC09
cCOA-CC17
CC1B-CC2E
CC2F-CC33
CC34
CC3D
CC45-CCD7
CCDB-CCFC
CCFD-CD31
CD32-CD41
CD42-CD50
CD51-CD60
CD6 1- CD6E
CD6F-CDBD
CD8E -CD 90
CD91-CDDO
CDD1-CE01
CE02-CE2B
CE29-CEB3
CEB 4-CEE B
CEE9-CEF8

177


Appendix B

D112  Perform power function
D14B-D155 Perform negation
D156-D183 Constants
D184-D196 Perform EXP
D1D7-D220 Series evaluation
D221-D228 RND constants
D229-D231 Perform RND
D232 Perform COS
D239-D241 Perform SIN
D242-D244 Perform TAN
D245-D247 Constants
D248-D250 Perform ATN
D251-D253 CHRGET sub for zero page
D254-D256 Basic cold start
D257-D259 Machine Language Monitor
D25A-D25C MLM subroutines
D263-D265 Perform RECORD
D266-D268 Disk parameter checks
D269-D271 Dummy disk control messages
D272-D274 Perform CATALOG or DIRECTORY
D275-D277 Output
D278-D280 Find spare secondary address
D281-D283 Perform DOPEN
D284-D286 Get disk status
D287-D289 Perform HEADER
D28A-D28C Perform DCLOSE
D28D-D28F Set up disk record
D290-D292 Perform COLLECT
D293-D295 Perform BACKUP
D296-D298 Perform COPY
D299-D29B Insert command string values
D29C-D29E Perform DSAVE
D29F-DB01 Perform DLOAD
DB02-DB08 Perform SCRATCH
DB09-DB0F Check Direct command
DB10-DB16 Query ARE YOU SURE?
DB17-DB1F Print BAD DISK
DB20-DB2F Clear DS$ and ST
DB30-DB3F Assemble disk command string
DB40-DB4F Parse Basic DOS command
DB50-DB5F Get Device number
DB60-DB6F Get file name
DB70-DB7F Get small variable parameter
** Entry points only for EO00-E7FF **
EO00 Register/screen initialization
EO07 Input from keyboard
E116 Input from screen
E202 Output character
E442 Main Interrupt entry
Appendix B

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
F55D-F5E4 Perform OPEN
F5E5-F618 Find any tape header
F619-F67A Write tape header
F67B-F694 Get start/end addr s from header
F695-F6AA Set buffer address
F6AB-F6C2 Set buffer start & end addr s
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
Appendix B

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-FBB3A Read tape characters
FBBB-FBC3 Reset tape read address
FBCC-FBC8 Flag error into ST
FBCD-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
FCD8-FCEA Set interrupt vector
FCEB-FCF8 Turn off tape motor
FCF9-FD01 Checksum calculation
FD0C-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
FFE8 Update clock
FFFA-FFFF Hard vectors: NMI, Reset, INT
# Map 5. VIC Zero Page And BASIC ROMs.

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000-0002</td>
<td>0-2</td>
<td>USR jump</td>
</tr>
<tr>
<td>0003-0004</td>
<td>3-4</td>
<td>Float-Fixed vector</td>
</tr>
<tr>
<td>0005-0006</td>
<td>5-6</td>
<td>Fixed-Float vector</td>
</tr>
<tr>
<td>0007</td>
<td>7</td>
<td>Search character</td>
</tr>
<tr>
<td>0008</td>
<td>8</td>
<td>Scan-quotes flag</td>
</tr>
<tr>
<td>0009</td>
<td>9</td>
<td>TAB column save</td>
</tr>
<tr>
<td>000A</td>
<td>10</td>
<td>0=LOAD, 1=VERIFY</td>
</tr>
<tr>
<td>000B</td>
<td>11</td>
<td>Input buffer pointer/# subscript</td>
</tr>
<tr>
<td>000C</td>
<td>12</td>
<td>Default DIM flag</td>
</tr>
<tr>
<td>000D</td>
<td>13</td>
<td>Type: FF=string, 00=numeric</td>
</tr>
<tr>
<td>000E</td>
<td>14</td>
<td>Type: 80=integer, 00=floating point</td>
</tr>
<tr>
<td>000F</td>
<td>15</td>
<td>DATA scan/LIST quote/memry flag</td>
</tr>
<tr>
<td>0010</td>
<td>16</td>
<td>Subscript/FNx flag</td>
</tr>
<tr>
<td>0011</td>
<td>17</td>
<td>0=INPUT;$40=GET;$98=READ</td>
</tr>
<tr>
<td>0012</td>
<td>18</td>
<td>ATN sign/Comparison eval flag</td>
</tr>
<tr>
<td>0013</td>
<td>19</td>
<td>Current I/O prompt flag</td>
</tr>
<tr>
<td>0014-0015</td>
<td>20-21</td>
<td>Integer value</td>
</tr>
<tr>
<td>0016</td>
<td>22</td>
<td>Pointer: temporary strg stack</td>
</tr>
<tr>
<td>0017-0018</td>
<td>23-24</td>
<td>Last temp string vector</td>
</tr>
<tr>
<td>0019-0021</td>
<td>25-33</td>
<td>Stack for temporary strings</td>
</tr>
<tr>
<td>0022-0025</td>
<td>34-37</td>
<td>Utility pointer area</td>
</tr>
<tr>
<td>0026-002A</td>
<td>38-42</td>
<td>Product area for multiplication</td>
</tr>
<tr>
<td>002B-002C</td>
<td>43-44</td>
<td>Pointer: Start-of-Basic</td>
</tr>
<tr>
<td>Address</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>002D-002E</td>
<td>45-46</td>
<td>Pointer: Start-of-Variables</td>
</tr>
<tr>
<td>002F-0030</td>
<td>47-48</td>
<td>Pointer: Start-of-Arrays</td>
</tr>
<tr>
<td>0031-0032</td>
<td>49-50</td>
<td>Pointer: End-of-Arrays</td>
</tr>
<tr>
<td>0033-0034</td>
<td>51-52</td>
<td>Pointer: String-storage(moving down)</td>
</tr>
<tr>
<td>0035-0036</td>
<td>53-54</td>
<td>Utility string pointer</td>
</tr>
<tr>
<td>0037-0038</td>
<td>55-56</td>
<td>Pointer: Limit-of-memory</td>
</tr>
<tr>
<td>0039-003A</td>
<td>57-58</td>
<td>Current Basic line number</td>
</tr>
<tr>
<td>003B-003C</td>
<td>59-60</td>
<td>Previous Basic line number</td>
</tr>
<tr>
<td>003D-003E</td>
<td>61-62</td>
<td>Pointer: Basic statement for CONT</td>
</tr>
<tr>
<td>003F-0040</td>
<td>63-64</td>
<td>Current DATA line number</td>
</tr>
<tr>
<td>0041-0042</td>
<td>65-66</td>
<td>Current DATA address</td>
</tr>
<tr>
<td>0043-0044</td>
<td>67-68</td>
<td>Input vector</td>
</tr>
<tr>
<td>0045-0046</td>
<td>69-70</td>
<td>Current variable name</td>
</tr>
<tr>
<td>0047-0048</td>
<td>71-72</td>
<td>Current variable address</td>
</tr>
<tr>
<td>0049-004A</td>
<td>73-74</td>
<td>Variable pointer for FOR/NEXT</td>
</tr>
<tr>
<td>004B-004C</td>
<td>75-76</td>
<td>Y-save; op-save; Basic pointer save</td>
</tr>
<tr>
<td>004D</td>
<td>77</td>
<td>Comparison symbol accumulator</td>
</tr>
<tr>
<td>004E-0053</td>
<td>78-83</td>
<td>Misc work area, pointers, etc</td>
</tr>
<tr>
<td>0054-0056</td>
<td>84-86</td>
<td>Jump vector for functions</td>
</tr>
<tr>
<td>0057-0060</td>
<td>87-96</td>
<td>Misc numeric work area</td>
</tr>
<tr>
<td>0061</td>
<td>97</td>
<td>Accum#1: Exponent</td>
</tr>
<tr>
<td>0062-0065</td>
<td>98-101</td>
<td>Accum#1: Mantissa</td>
</tr>
<tr>
<td>0066</td>
<td>102</td>
<td>Accum#1: Sign</td>
</tr>
<tr>
<td>0067</td>
<td>103</td>
<td>Series evaluation constant pointer</td>
</tr>
<tr>
<td>0068</td>
<td>104</td>
<td>Accum#1 hi-order (overflow)</td>
</tr>
<tr>
<td>0069-006E</td>
<td>105-110</td>
<td>Accum#2: Exponent, etc.</td>
</tr>
<tr>
<td>Address</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>006F</td>
<td>111</td>
<td>Sign comparison, Acc#1 vs #2</td>
</tr>
<tr>
<td>0070</td>
<td>112</td>
<td>Accum#1 lo-order (rounding)</td>
</tr>
<tr>
<td>0071-0072</td>
<td>113-114</td>
<td>Cassette buff len/Series pointer</td>
</tr>
<tr>
<td>0073-008A</td>
<td>115-138</td>
<td>CHRGET subroutine; get Basic char</td>
</tr>
<tr>
<td>007A-007B</td>
<td>122-123</td>
<td>Basic pointer (within subrtn)</td>
</tr>
<tr>
<td>008B-008F</td>
<td>139-143</td>
<td>RND seed value</td>
</tr>
<tr>
<td>0090</td>
<td>144</td>
<td>Status word ST</td>
</tr>
<tr>
<td>0091</td>
<td>145</td>
<td>Keyswitch PIA: STOP and RVS flags</td>
</tr>
<tr>
<td>0092</td>
<td>146</td>
<td>Timing constant for tape</td>
</tr>
<tr>
<td>0093</td>
<td>147</td>
<td>Load=0, Verify=1</td>
</tr>
<tr>
<td>0094</td>
<td>148</td>
<td>Serial output: deferred char flag</td>
</tr>
<tr>
<td>0095</td>
<td>149</td>
<td>Serial deferred character</td>
</tr>
<tr>
<td>0096</td>
<td>150</td>
<td>Tape EOT received</td>
</tr>
<tr>
<td>0097</td>
<td>151</td>
<td>Register save</td>
</tr>
<tr>
<td>0098</td>
<td>152</td>
<td>How many open files</td>
</tr>
<tr>
<td>0099</td>
<td>153</td>
<td>Input device, normally 0</td>
</tr>
<tr>
<td>009A</td>
<td>154</td>
<td>Output CMD device, normally 3</td>
</tr>
<tr>
<td>009B</td>
<td>155</td>
<td>Tape character parity</td>
</tr>
<tr>
<td>009C</td>
<td>156</td>
<td>Byte-received flag</td>
</tr>
<tr>
<td>009D</td>
<td>157</td>
<td>Direct=80/RUN=0 output control</td>
</tr>
<tr>
<td>009E</td>
<td>158</td>
<td>Tp Pass 1 error log/char buffer</td>
</tr>
<tr>
<td>009F</td>
<td>159</td>
<td>Tp Pass 2 err log corrected</td>
</tr>
<tr>
<td>00A0-00A2</td>
<td>160-162</td>
<td>Jiffy Clock HML</td>
</tr>
<tr>
<td>00A3</td>
<td>163</td>
<td>Serial bit count/EOI flag</td>
</tr>
<tr>
<td>00A4</td>
<td>164</td>
<td>Cycle count</td>
</tr>
<tr>
<td>00A5</td>
<td>165</td>
<td>Countdown, tape write/bit count</td>
</tr>
</tbody>
</table>
Tape buffer pointer
Tp Wrt ldr count/Rd pass/inbit
Tp Wrt new byte/Rd error/inbit cnt
Wrt start bit/Rd bit err/stbit
Tp Scan;Cnt;Ld;End/byte assy
Wrt lead length/Rd checksum/parity

Pointer: tape bufr, scrolling
Tape end adds/End of program
Tape timing constants
Pntr: start of tape buffer
1=Tp timer enabled; bit cnt
Tp EOT/RS232 next bit to send
Read character error/outbyte buf
# characters in file name
Current logical file
Current secndy address
Current device
Pointer to file name
Wr shift word/Rd input char
# blocks remaining to Wr/Rd
Serial word buffer
Tape motor interlock
I/O start adds
Kernel setup pointer
Last key pressed
# chars in keybd buffer
<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00C7</td>
<td>199</td>
<td>Screen reverse flag</td>
</tr>
<tr>
<td>00C8</td>
<td>200</td>
<td>End-of-line for input pointer</td>
</tr>
<tr>
<td>00C9-C0CA</td>
<td>201-202</td>
<td>Input cursor log (row, column)</td>
</tr>
<tr>
<td>00CB</td>
<td>203</td>
<td>Which key: 64 if no key</td>
</tr>
<tr>
<td>00CC</td>
<td>204</td>
<td>0=flash cursor</td>
</tr>
<tr>
<td>00CD</td>
<td>205</td>
<td>Cursor timing countdown</td>
</tr>
<tr>
<td>00CE</td>
<td>206</td>
<td>Character under cursor</td>
</tr>
<tr>
<td>00CF</td>
<td>207</td>
<td>Cursor in blink phase</td>
</tr>
<tr>
<td>00D0</td>
<td>208</td>
<td>Input from screen/from keyboard</td>
</tr>
<tr>
<td>00D1-D0D2</td>
<td>209-210</td>
<td>Pointer to screen line</td>
</tr>
<tr>
<td>00D3</td>
<td>211</td>
<td>Position of cursor on above line</td>
</tr>
<tr>
<td>00D4</td>
<td>212</td>
<td>0=direct cursor, else programmed</td>
</tr>
<tr>
<td>00D5</td>
<td>213</td>
<td>Current screen line length</td>
</tr>
<tr>
<td>00D6</td>
<td>214</td>
<td>Row where cursor lives</td>
</tr>
<tr>
<td>00D7</td>
<td>215</td>
<td>Last inkey/checksum/buffer</td>
</tr>
<tr>
<td>00D8</td>
<td>216</td>
<td># of INSERTs outstanding</td>
</tr>
<tr>
<td>00D9-D0F0</td>
<td>217-240</td>
<td>Screen line link table</td>
</tr>
<tr>
<td>00F1</td>
<td>241</td>
<td>Dummy screen link</td>
</tr>
<tr>
<td>00F2</td>
<td>242</td>
<td>Screen row marker</td>
</tr>
<tr>
<td>00F3-F0F4</td>
<td>243-244</td>
<td>Screen color pointer</td>
</tr>
<tr>
<td>00F5-F0F6</td>
<td>245-246</td>
<td>Keyboard pointer</td>
</tr>
<tr>
<td>00F7-F0F8</td>
<td>247-248</td>
<td>RS-232 Rcv pntr</td>
</tr>
<tr>
<td>00F9-F0FA</td>
<td>249-250</td>
<td>RS-232 Tx pntr</td>
</tr>
<tr>
<td>00FF-010A</td>
<td>255-266</td>
<td>Floating to ASCII work area</td>
</tr>
</tbody>
</table>
### 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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c000</td>
<td>ROM control vectors</td>
</tr>
<tr>
<td>c00c</td>
<td>Keyword action vectors</td>
</tr>
<tr>
<td>c052</td>
<td>Function vectors</td>
</tr>
<tr>
<td>c080</td>
<td>Operator vectors</td>
</tr>
<tr>
<td>c09e</td>
<td>Keywords</td>
</tr>
<tr>
<td>c19e</td>
<td>Error messages</td>
</tr>
<tr>
<td>c328</td>
<td>Error message vectors</td>
</tr>
<tr>
<td>c365</td>
<td>Miscellaneous messages</td>
</tr>
<tr>
<td>c38a</td>
<td>Scan stack for FOR/GOSUB</td>
</tr>
<tr>
<td>c3b8</td>
<td>Move memory</td>
</tr>
<tr>
<td>c3fb</td>
<td>Check stack depth</td>
</tr>
<tr>
<td>c408</td>
<td>Check memory space</td>
</tr>
<tr>
<td>c435</td>
<td>'OUT OF MEMORY'</td>
</tr>
<tr>
<td>c437</td>
<td>Error routine</td>
</tr>
<tr>
<td>c469</td>
<td>Break entry</td>
</tr>
<tr>
<td>c474</td>
<td>'READY.'</td>
</tr>
<tr>
<td>c480</td>
<td>Ready for Basic</td>
</tr>
<tr>
<td>c49c</td>
<td>Handle new line</td>
</tr>
<tr>
<td>c533</td>
<td>Re-chain lines</td>
</tr>
<tr>
<td>c560</td>
<td>Receive input line</td>
</tr>
<tr>
<td>c579</td>
<td>Crunch tokens</td>
</tr>
<tr>
<td>c613</td>
<td>Find Basic line</td>
</tr>
<tr>
<td>c642</td>
<td>Perform [NEW]</td>
</tr>
<tr>
<td>c65e</td>
<td>Perform [CLR]</td>
</tr>
<tr>
<td>c68e</td>
<td>Back up text pointer</td>
</tr>
<tr>
<td>c69c</td>
<td>Perform [LIST]</td>
</tr>
<tr>
<td>c742</td>
<td>Perform [FOR]</td>
</tr>
<tr>
<td>c7ed</td>
<td>Execute statement</td>
</tr>
<tr>
<td>c81d</td>
<td>Perform [RESTORE]</td>
</tr>
<tr>
<td>c82c</td>
<td>Break</td>
</tr>
<tr>
<td>c82f</td>
<td>Perform [STOP]</td>
</tr>
<tr>
<td>c831</td>
<td>Perform [END]</td>
</tr>
<tr>
<td>c857</td>
<td>Perform [CONT]</td>
</tr>
<tr>
<td>c871</td>
<td>Perform [RUN]</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>c883</td>
<td>Perform [GOSUB]</td>
</tr>
<tr>
<td>c8a0</td>
<td>Perform [GOTO]</td>
</tr>
<tr>
<td>c8d2</td>
<td>Perform [RETURN]</td>
</tr>
<tr>
<td>c8f8</td>
<td>Perform [DATA]</td>
</tr>
<tr>
<td>c906</td>
<td>Scan for next statement</td>
</tr>
<tr>
<td>c928</td>
<td>Perform [IF]</td>
</tr>
<tr>
<td>c93b</td>
<td>Perform [REM]</td>
</tr>
<tr>
<td>c94b</td>
<td>Perform [ON]</td>
</tr>
<tr>
<td>c96b</td>
<td>Get fixed point number</td>
</tr>
<tr>
<td>c9a5</td>
<td>Perform [LET]</td>
</tr>
<tr>
<td>ca80</td>
<td>Perform [PRINT#]</td>
</tr>
<tr>
<td>ca86</td>
<td>Perform [CMD]</td>
</tr>
<tr>
<td>caa0</td>
<td>Perform [PRINT]</td>
</tr>
<tr>
<td>cb1e</td>
<td>Print message from (y,a)</td>
</tr>
<tr>
<td>cb3b</td>
<td>Print format character</td>
</tr>
<tr>
<td>cb4d</td>
<td>Bad-input routines</td>
</tr>
<tr>
<td>cb7b</td>
<td>Perform [GET]</td>
</tr>
<tr>
<td>cba5</td>
<td>Perform [INPUT#]</td>
</tr>
<tr>
<td>cbbf</td>
<td>Perform [INPUT]</td>
</tr>
<tr>
<td>cbf9</td>
<td>Prompt &amp; input</td>
</tr>
<tr>
<td>cc06</td>
<td>Perform [READ]</td>
</tr>
<tr>
<td>ccfc</td>
<td>Input error messages</td>
</tr>
<tr>
<td>cd1e</td>
<td>Perform [NEXT]</td>
</tr>
<tr>
<td>cd78</td>
<td>Type-match check</td>
</tr>
<tr>
<td>cd9e</td>
<td>Evaluate expression</td>
</tr>
<tr>
<td>cea8</td>
<td>Constant - PI</td>
</tr>
</tbody>
</table>
d3e1  Check FN syntax

d3f4  Perform [FN]

d465  Perform [STR$]

d475  Calculate string vector

d487  Set up string

d4f4  Make room for string

d526  Garbage collection

d5bd  Check salvageability

d606  Collect string

d63d  Concatenate

Build string to memory

Discard unwanted string

Clean descriptor stack

Perform [CHR$]

Perform [LEFT$]

Perform [RIGHT$]

Perform [MID$]

Pull string parameters

Perform [LEN]

Exit string-mode

Perform [ASC]

Input byte parameter

Perform [VAL]

Get params for poke/wait

Float-fixed

Perform [POKE]

Perform [WAIT]

Add 0.5

Subtract-from

Perform [SUBTRACT]

Perform [ADD]

Complement fac#1

'OVERFLOW'

Multiply by zero byte

Perform [LOG]

Perform [MULTIPLY]

Multiply-a-bit

Memory to FAC#2

Adjust FAC#1/#2

Underflow/overflow

Multiply by 10

+10 in floating pt

Divide by 10

Perform [DIVIDE]

Memory to fac#1

FAC#1 to memory

FAC#2 to fac#1

FAC#1 to FAC#2

Round FAC#1

Get sign

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]
df84  Perform [NEGATIVE]
dfed  Perform [EXP]
e040  Series evaluate 1
e056  Series evaluate 2
e094  Perform [RND]
e0f6  ?? Breakpoints ??
e127  Perform [SYS]
e143  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]
e2bd  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 addr
e505  Set screen limits
e50a  Track cursor location
e518  Initialize I/O
e54c  Normalize screen
e55f  Clear screen
e581  Home cursor
e587  Set screen pointers
e5bb  Set I/o defaults
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>e5c3</td>
<td>Set vic chip defaults</td>
</tr>
<tr>
<td>e5cf</td>
<td>Input from keyboard</td>
</tr>
<tr>
<td>e64f</td>
<td>Input from screen</td>
</tr>
<tr>
<td>e6b8</td>
<td>Quote mark test</td>
</tr>
<tr>
<td>e6c5</td>
<td>Set up screen print</td>
</tr>
<tr>
<td>e6ea</td>
<td>Advance cursor</td>
</tr>
<tr>
<td>e715</td>
<td>Retreat cursor</td>
</tr>
<tr>
<td>e72d</td>
<td>Back into previous line</td>
</tr>
<tr>
<td>e742</td>
<td>Output to screen</td>
</tr>
<tr>
<td>e8c3</td>
<td>Go to next line</td>
</tr>
<tr>
<td>e8d8</td>
<td>Do 'RETURN'</td>
</tr>
<tr>
<td>e8e8</td>
<td>Check line decrement</td>
</tr>
<tr>
<td>e8fa</td>
<td>Check line increment</td>
</tr>
<tr>
<td>e912</td>
<td>Set colour code</td>
</tr>
<tr>
<td>e921</td>
<td>Colour code table</td>
</tr>
<tr>
<td>e929</td>
<td>Code conversion</td>
</tr>
<tr>
<td>e975</td>
<td>Scroll screen</td>
</tr>
<tr>
<td>e9ee</td>
<td>Open space on screen</td>
</tr>
<tr>
<td>ea56</td>
<td>Move screen line</td>
</tr>
<tr>
<td>ea6e</td>
<td>Synch colour transfer</td>
</tr>
<tr>
<td>ea7e</td>
<td>Set start-of-line</td>
</tr>
<tr>
<td>ea8d</td>
<td>Clear screen line</td>
</tr>
<tr>
<td>eaal</td>
<td>Print to screen</td>
</tr>
<tr>
<td>eaaa</td>
<td>Store on screen</td>
</tr>
<tr>
<td>eab2</td>
<td>Synch colour to char</td>
</tr>
<tr>
<td>eabf</td>
<td>Interrupt (IRQ)</td>
</tr>
<tr>
<td>eble</td>
<td>Check keyboard</td>
</tr>
<tr>
<td>ec00</td>
<td>Set text mode</td>
</tr>
<tr>
<td>ec46</td>
<td>Keyboard vectors</td>
</tr>
<tr>
<td>ec5e</td>
<td>Keyboard maps</td>
</tr>
<tr>
<td>ed21</td>
<td>Graphics/text control</td>
</tr>
<tr>
<td>ed30</td>
<td>Set graphics mode</td>
</tr>
<tr>
<td>ed5b</td>
<td>Wrap up screen line</td>
</tr>
<tr>
<td>ed6a</td>
<td>Shifted key matrix</td>
</tr>
<tr>
<td>eda3</td>
<td>Control key matrix</td>
</tr>
<tr>
<td>ede4</td>
<td>Vic chip defaults</td>
</tr>
<tr>
<td>edfd</td>
<td>Screen line adds low</td>
</tr>
<tr>
<td>ee14</td>
<td>Send 'talk'</td>
</tr>
<tr>
<td>ee17</td>
<td>Send 'listen'</td>
</tr>
<tr>
<td>ee1c</td>
<td>Send control char</td>
</tr>
<tr>
<td>ee49</td>
<td>Send to serial bus</td>
</tr>
<tr>
<td>eeb7</td>
<td>Timeout on serial</td>
</tr>
<tr>
<td>eec0</td>
<td>Send listen SA</td>
</tr>
<tr>
<td>eec5</td>
<td>Clear ATN</td>
</tr>
<tr>
<td>eece</td>
<td>Send talk SA</td>
</tr>
<tr>
<td>eee4</td>
<td>Send serial deferred</td>
</tr>
<tr>
<td>eef6</td>
<td>Send 'untalk'</td>
</tr>
<tr>
<td>ef04</td>
<td>Send 'unlisten'</td>
</tr>
<tr>
<td>efl9</td>
<td>Receive from serial bus</td>
</tr>
<tr>
<td>ef84</td>
<td>Clock line on</td>
</tr>
<tr>
<td>ef8d</td>
<td>Clock line off</td>
</tr>
<tr>
<td>ef96</td>
<td>Delay 1 ms</td>
</tr>
</tbody>
</table>
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...
       ....from RS232
f205    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
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f894</td>
<td>_PRESS PLAY .. '</td>
</tr>
<tr>
<td>f8ab</td>
<td>Check cassette status</td>
</tr>
<tr>
<td>f8b7</td>
<td>_PRESS RECORD .. '</td>
</tr>
<tr>
<td>f8c0</td>
<td>Initiate tape read</td>
</tr>
<tr>
<td>f8e3</td>
<td>Initiate tape write</td>
</tr>
<tr>
<td>f8f4</td>
<td>Common tape read/write</td>
</tr>
<tr>
<td>f94b</td>
<td>Check tape stop</td>
</tr>
<tr>
<td>f95d</td>
<td>Set timing</td>
</tr>
<tr>
<td>f98e</td>
<td>Read bits (IRQ)</td>
</tr>
<tr>
<td>faad</td>
<td>Store characters</td>
</tr>
<tr>
<td>fbd2</td>
<td>Reset pointer</td>
</tr>
<tr>
<td>fbd8b</td>
<td>New tape character setup</td>
</tr>
<tr>
<td>fbea</td>
<td>Toggle tape</td>
</tr>
<tr>
<td>fc06</td>
<td>Data write</td>
</tr>
<tr>
<td>fc0b</td>
<td>Tape write (IRQ)</td>
</tr>
<tr>
<td>fc95</td>
<td>Leader write (IRQ)</td>
</tr>
<tr>
<td>fccf</td>
<td>Restore vectors</td>
</tr>
<tr>
<td>fc6f</td>
<td>Set vector</td>
</tr>
<tr>
<td>fd08</td>
<td>Kill motor</td>
</tr>
<tr>
<td>fd11</td>
<td>Check read/write pointer</td>
</tr>
<tr>
<td>fd1b</td>
<td>Bump read/write pointer</td>
</tr>
<tr>
<td>fd22</td>
<td>Powerup entry</td>
</tr>
<tr>
<td>fd3f</td>
<td>Check A-rom</td>
</tr>
<tr>
<td>fd52</td>
<td>Set kernal12</td>
</tr>
<tr>
<td>fd8d</td>
<td>Initialize system constants</td>
</tr>
<tr>
<td>fdf1</td>
<td>IRQ vectors</td>
</tr>
<tr>
<td>fdf9</td>
<td>Initialize I/O regs</td>
</tr>
<tr>
<td>fe49</td>
<td>Save data name</td>
</tr>
<tr>
<td>fe50</td>
<td>Save file details</td>
</tr>
<tr>
<td>fe57</td>
<td>Get status</td>
</tr>
<tr>
<td>fe66</td>
<td>Flag ST</td>
</tr>
<tr>
<td>fe6f</td>
<td>Set timeout</td>
</tr>
<tr>
<td>fe73</td>
<td>Read/set top of memory</td>
</tr>
<tr>
<td>fe82</td>
<td>Read/set bottom of memory</td>
</tr>
<tr>
<td>fe91</td>
<td>Test memory location</td>
</tr>
<tr>
<td>fea9</td>
<td>NMI interrupt entry</td>
</tr>
<tr>
<td>fed2</td>
<td>RESET/STOP warm start</td>
</tr>
<tr>
<td>fede</td>
<td>NMI RS232 sequences</td>
</tr>
<tr>
<td>ff56</td>
<td>Restore &amp; exit</td>
</tr>
<tr>
<td>ff5c</td>
<td>RS232 timing table</td>
</tr>
<tr>
<td>ff72</td>
<td>Main IRQ entry</td>
</tr>
<tr>
<td>ff8a</td>
<td>Jumbo jump table</td>
</tr>
<tr>
<td>fffa</td>
<td>Hardware vectors</td>
</tr>
</tbody>
</table>
Map 6. Commodore 64 Memory Map.

<table>
<thead>
<tr>
<th>SID</th>
<th>Frequency</th>
<th>Pulse Width</th>
<th>Voice Type</th>
<th>Attack Time</th>
<th>Decay Time</th>
<th>Sustain Level</th>
<th>Release Time</th>
<th>Filter Frequency</th>
<th>Filter Voices</th>
<th>Filter &amp; Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>D400 D407 D40E</td>
<td>0400 0407 040E</td>
<td>0401 0408 040F</td>
<td>L</td>
<td>2 ms - 8 sec</td>
<td>6 ms - 24 sec</td>
<td>6 ms - 24 sec</td>
<td>54272 54279 54286</td>
<td>54273 54280 54287</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D401 D408 D40F</td>
<td></td>
<td></td>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54274 54281 54288</td>
<td>54275 54282 54289</td>
<td></td>
</tr>
<tr>
<td>D402 D409 D410</td>
<td></td>
<td></td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54276 54283 54290</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D403 D40A D411</td>
<td></td>
<td></td>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54277 54284 54291</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D404 D40B D412</td>
<td></td>
<td></td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54278 54285 54292</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D405 D40C D413</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54293 54294</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D406 D40D D414</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54295</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

-- Frequency

-- Pulse Width

-- Voice Type

-- Attack Time

-- Decay Time

-- Sustain Level

-- Release Time

-- Filter Frequency

-- Filter Voices

-- Resonance

-- Master Volume

-- Passband

-- Off

-- Hi

-- Bd

-- Lo

(Voice Type): NSE, PUL, SAW, TRI

(Resonance): EXT, V3, V2, V1

(Filter & Volume): Write Only
Special voice features (TEST, RING MOD, SYNC) are omitted from the above diagram.

*Connected but not used by system.*
### Processor I/O Port (6510) Commodore 64

<table>
<thead>
<tr>
<th>Address</th>
<th>IN</th>
<th>IN</th>
<th>Out</th>
<th>IN</th>
<th>Out</th>
<th>Out</th>
<th>Out</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0001</td>
<td></td>
<td></td>
<td>Tape Motor</td>
<td>Tape Sense</td>
<td>Tape Write</td>
<td>D-Rom Switch</td>
<td>EF.RAM Switch</td>
<td>AB.RAM Switch</td>
</tr>
</tbody>
</table>

### CIA 1 (IRQ) (6526) Commodore 64

#### Paddle SEL
- **A**
- **B**

#### Joystick 0
- **R**
- **L**
- **D**
- **U**

Keyboard Row Select (Inverted)

#### Keyboard Column Read

- **$FF** = All Output
- **$00** = All Input

#### Timer A

#### Timer B

#### Tape Input

One Shot Out Mode

#### Timer Interr. A

#### Timer B

#### Timer Interr. B

#### Time PB6 Out

#### Time PB7 Out

#### Shot Mode

#### Start

#### Input

#### Output

---

### Additional Addresses

- **PRA 56320**
- **PRB 56321**
- **DDRA 56322**
- **DDR B 56323**
- **TAL 56324**
- **TAH 56325**
- **TBL 56326**
- **TBH 56327**
- **ICR 56333**
- **CRA 56334**
- **CRB 56335**
### 64 Memory Map

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>Chip directional register</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>Chip I/O; memory &amp; tape control</td>
</tr>
<tr>
<td>0003-0004</td>
<td>3-4</td>
<td>Float-Fixed vector</td>
</tr>
<tr>
<td>0005-0006</td>
<td>5-6</td>
<td>Fixed-Float vector</td>
</tr>
<tr>
<td>0007</td>
<td>7</td>
<td>Search character</td>
</tr>
<tr>
<td>0008</td>
<td>8</td>
<td>Scan-quotes flag</td>
</tr>
<tr>
<td>0009</td>
<td>9</td>
<td>TAB column save</td>
</tr>
<tr>
<td>000A</td>
<td>10</td>
<td>0=LOAD, 1=VERIFY</td>
</tr>
<tr>
<td>000B</td>
<td>11</td>
<td>Input buffer pointer/# subscript</td>
</tr>
<tr>
<td>000C</td>
<td>12</td>
<td>Default DIM flag</td>
</tr>
<tr>
<td>000D</td>
<td>13</td>
<td>Type: FF=string, 00=numeric</td>
</tr>
<tr>
<td>000E</td>
<td>14</td>
<td>Type: 80=integer, 00=floating point</td>
</tr>
<tr>
<td>000F</td>
<td>15</td>
<td>DATA scan/LIST quote/memry flag</td>
</tr>
<tr>
<td>0010</td>
<td>16</td>
<td>Subscript/FNx flag</td>
</tr>
<tr>
<td>0011</td>
<td>17</td>
<td>0=INPUT;$40=GET;$98=READ</td>
</tr>
<tr>
<td>0012</td>
<td>18</td>
<td>ATN sign/Comparison eval flag</td>
</tr>
<tr>
<td>0013</td>
<td>19</td>
<td>Current I/O prompt flag</td>
</tr>
<tr>
<td>0014-0015</td>
<td>20-21</td>
<td>Integer value</td>
</tr>
<tr>
<td>0016</td>
<td>22</td>
<td>Pointer: temporary strg stack</td>
</tr>
<tr>
<td>0017-0018</td>
<td>23-24</td>
<td>Last temp string vector</td>
</tr>
<tr>
<td>0019-0021</td>
<td>25-33</td>
<td>Stack for temporary strings</td>
</tr>
<tr>
<td>0022-0025</td>
<td>34-37</td>
<td>Utility pointer area</td>
</tr>
<tr>
<td>0026-002A</td>
<td>38-42</td>
<td>Product area for multiplication</td>
</tr>
<tr>
<td>002B-002C</td>
<td>43-44</td>
<td>Pointer: Start-of-Basic</td>
</tr>
<tr>
<td>Address</td>
<td>Bytes</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>002D-002E</td>
<td>45-46</td>
<td>Pointer: Start-of-Variables</td>
</tr>
<tr>
<td>002F-0030</td>
<td>47-48</td>
<td>Pointer: Start-of-Arrays</td>
</tr>
<tr>
<td>0031-0032</td>
<td>49-50</td>
<td>Pointer: End-of-Arrays</td>
</tr>
<tr>
<td>0033-0034</td>
<td>51-52</td>
<td>Pointer: String-storage(moving down)</td>
</tr>
<tr>
<td>0035-0036</td>
<td>53-54</td>
<td>Utility string pointer</td>
</tr>
<tr>
<td>0037-0038</td>
<td>55-56</td>
<td>Pointer: Limit-of-memory</td>
</tr>
<tr>
<td>0039-003A</td>
<td>57-58</td>
<td>Current Basic line number</td>
</tr>
<tr>
<td>003B-003C</td>
<td>59-60</td>
<td>Previous Basic line number</td>
</tr>
<tr>
<td>003D-003E</td>
<td>61-62</td>
<td>Pointer: Basic statement for CONT</td>
</tr>
<tr>
<td>003F-0040</td>
<td>63-64</td>
<td>Current DATA line number</td>
</tr>
<tr>
<td>0041-0042</td>
<td>65-66</td>
<td>Current DATA address</td>
</tr>
<tr>
<td>0043-0044</td>
<td>67-68</td>
<td>Input vector</td>
</tr>
<tr>
<td>0045-0046</td>
<td>69-70</td>
<td>Current variable name</td>
</tr>
<tr>
<td>0047-0048</td>
<td>71-72</td>
<td>Current variable address</td>
</tr>
<tr>
<td>0049-004A</td>
<td>73-74</td>
<td>Variable pointer for FOR/NEXT</td>
</tr>
<tr>
<td>004B-004C</td>
<td>75-76</td>
<td>Y-save; op-save; Basic pointer save</td>
</tr>
<tr>
<td>004D</td>
<td>77</td>
<td>Comparison symbol accumulator</td>
</tr>
<tr>
<td>004E-0053</td>
<td>78-83</td>
<td>Misc work area, pointers, etc</td>
</tr>
<tr>
<td>0054-0056</td>
<td>84-86</td>
<td>Jump vector for functions</td>
</tr>
<tr>
<td>0057-0060</td>
<td>87-96</td>
<td>Misc numeric work area</td>
</tr>
<tr>
<td>0061</td>
<td>97</td>
<td>Accum#1: Exponent</td>
</tr>
<tr>
<td>0062-0065</td>
<td>98-101</td>
<td>Accum#1: Mantissa</td>
</tr>
<tr>
<td>0066</td>
<td>102</td>
<td>Accum#1: Sign</td>
</tr>
<tr>
<td>0067</td>
<td>103</td>
<td>Series evaluation constant pointer</td>
</tr>
<tr>
<td>0068</td>
<td>104</td>
<td>Accum#1 hi-order (overflow)</td>
</tr>
<tr>
<td>Address</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>0069-006E</td>
<td>105-110</td>
<td>Accum#2: Exponent, etc.</td>
</tr>
<tr>
<td>006F</td>
<td>111</td>
<td>Sign comparison, Acc#1 vs #2</td>
</tr>
<tr>
<td>0070</td>
<td>112</td>
<td>Accum#1 lo-order (rounding)</td>
</tr>
<tr>
<td>0071-0072</td>
<td>113-114</td>
<td>Cassette buff len/Series pointer</td>
</tr>
<tr>
<td>0073-008A</td>
<td>115-138</td>
<td>CHRGET subroutine; get Basic char</td>
</tr>
<tr>
<td>007A-007B</td>
<td>122-123</td>
<td>Basic pointer (within subrtn)</td>
</tr>
<tr>
<td>008B-008F</td>
<td>139-143</td>
<td>RND seed value</td>
</tr>
<tr>
<td>0090</td>
<td>144</td>
<td>Status word ST</td>
</tr>
<tr>
<td>0091</td>
<td>145</td>
<td>Keyswitch PIA: STOP and RVS flags</td>
</tr>
<tr>
<td>0092</td>
<td>146</td>
<td>Timing constant for tape</td>
</tr>
<tr>
<td>0093</td>
<td>147</td>
<td>Load=0, Verify=1</td>
</tr>
<tr>
<td>0094</td>
<td>148</td>
<td>Serial output: deferred char flag</td>
</tr>
<tr>
<td>0095</td>
<td>149</td>
<td>Serial deferred character</td>
</tr>
<tr>
<td>0096</td>
<td>150</td>
<td>Tape EOT received</td>
</tr>
<tr>
<td>0097</td>
<td>151</td>
<td>Register save</td>
</tr>
<tr>
<td>0098</td>
<td>152</td>
<td>How many open files</td>
</tr>
<tr>
<td>0099</td>
<td>153</td>
<td>Input device, normally 0</td>
</tr>
<tr>
<td>009A</td>
<td>154</td>
<td>Output CMD device, normally 3</td>
</tr>
<tr>
<td>009B</td>
<td>155</td>
<td>Tape character parity</td>
</tr>
<tr>
<td>009C</td>
<td>156</td>
<td>Byte-received flag</td>
</tr>
<tr>
<td>009D</td>
<td>157</td>
<td>Direct=$80/RUN=0 output control</td>
</tr>
<tr>
<td>009E</td>
<td>158</td>
<td>Tp Pass 1 error log/char buffer</td>
</tr>
<tr>
<td>009F</td>
<td>159</td>
<td>Tp Pass 2 err log corrected</td>
</tr>
<tr>
<td>00A0-00A2</td>
<td>160-162</td>
<td>Jiffy Clock HML</td>
</tr>
<tr>
<td>00A3</td>
<td>163</td>
<td>Serial bit count/EOI flag</td>
</tr>
</tbody>
</table>
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 L=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 cursor 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
<table>
<thead>
<tr>
<th>Address Range</th>
<th>Memory Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100-013E</td>
<td>256-318</td>
<td>Tape error log</td>
</tr>
<tr>
<td>0100-01FF</td>
<td>256-511</td>
<td>Processor stack area</td>
</tr>
<tr>
<td>0200-0258</td>
<td>512-600</td>
<td>Basic input buffer</td>
</tr>
<tr>
<td>0259-0262</td>
<td>601-610</td>
<td>Logical file table</td>
</tr>
<tr>
<td>0263-026C</td>
<td>611-620</td>
<td>Device # table</td>
</tr>
<tr>
<td>026D-0276</td>
<td>621-630</td>
<td>Sec Adds table</td>
</tr>
<tr>
<td>0277-0280</td>
<td>631-640</td>
<td>Keybd buffer</td>
</tr>
<tr>
<td>0281-0282</td>
<td>641-642</td>
<td>Start of Basic Memory</td>
</tr>
<tr>
<td>0283-0284</td>
<td>643-644</td>
<td>Top of Basic Memory</td>
</tr>
<tr>
<td>0285</td>
<td>645</td>
<td>Serial bus timeout flag</td>
</tr>
<tr>
<td>0286</td>
<td>646</td>
<td>Current color code</td>
</tr>
<tr>
<td>0287</td>
<td>647</td>
<td>Color under cursor</td>
</tr>
<tr>
<td>0288</td>
<td>648</td>
<td>Screen memory page</td>
</tr>
<tr>
<td>0289</td>
<td>649</td>
<td>Max size of keybd buffer</td>
</tr>
<tr>
<td>028A</td>
<td>650</td>
<td>Repeat all keys</td>
</tr>
<tr>
<td>028B</td>
<td>651</td>
<td>Repeat speed counter</td>
</tr>
<tr>
<td>028C</td>
<td>652</td>
<td>Repeat delay counter</td>
</tr>
<tr>
<td>028D</td>
<td>653</td>
<td>Keyboard Shift/Control flag</td>
</tr>
<tr>
<td>028E</td>
<td>654</td>
<td>Last shift pattern</td>
</tr>
<tr>
<td>028F-0290</td>
<td>655-656</td>
<td>Keyboard table setup pointer</td>
</tr>
<tr>
<td>0291</td>
<td>657</td>
<td>Keyboard shift mode</td>
</tr>
<tr>
<td>0292</td>
<td>658</td>
<td>0=scroll enable</td>
</tr>
<tr>
<td>0293</td>
<td>659</td>
<td>RS-232 control reg</td>
</tr>
<tr>
<td>0294</td>
<td>660</td>
<td>RS-232 command reg</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>0295-0296</td>
<td>661-662  Bit timing</td>
<td></td>
</tr>
<tr>
<td>0297</td>
<td>663  RS-232 status</td>
<td></td>
</tr>
<tr>
<td>0298</td>
<td>664  # bits to send</td>
<td></td>
</tr>
<tr>
<td>0299-029A</td>
<td>665  RS-232 speed/code</td>
<td></td>
</tr>
<tr>
<td>029B</td>
<td>667  RS232 receive pointer</td>
<td></td>
</tr>
<tr>
<td>029C</td>
<td>668  RS232 input pointer</td>
<td></td>
</tr>
<tr>
<td>029D</td>
<td>669  RS232 transmit pointer</td>
<td></td>
</tr>
<tr>
<td>029E</td>
<td>670  RS232 output pointer</td>
<td></td>
</tr>
<tr>
<td>029F-02A0</td>
<td>671-672  IRQ save during tape I/O</td>
<td></td>
</tr>
<tr>
<td>02A1</td>
<td>673  CIA 2 (NMI) Interrupt Control</td>
<td></td>
</tr>
<tr>
<td>02A2</td>
<td>674  CIA 1 Timer A control log</td>
<td></td>
</tr>
<tr>
<td>02A3</td>
<td>675  CIA 1 Interrupt Log</td>
<td></td>
</tr>
<tr>
<td>02A4</td>
<td>676  CIA 1 Timer A enabled flag</td>
<td></td>
</tr>
<tr>
<td>02A5</td>
<td>677  Screen row marker</td>
<td></td>
</tr>
<tr>
<td>02C0-02FE</td>
<td>704-766  (Sprite 11)</td>
<td></td>
</tr>
<tr>
<td>0300-0301</td>
<td>768-769  Error message link</td>
<td></td>
</tr>
<tr>
<td>0302-0303</td>
<td>770-771  Basic warm start link</td>
<td></td>
</tr>
<tr>
<td>0304-0305</td>
<td>772-773  Crunch Basic tokens link</td>
<td></td>
</tr>
<tr>
<td>0306-0307</td>
<td>774-775  Print tokens link</td>
<td></td>
</tr>
<tr>
<td>0308-0309</td>
<td>776-777  Start new Basic code link</td>
<td></td>
</tr>
<tr>
<td>030A-030B</td>
<td>778-779  Get arithmetic element link</td>
<td></td>
</tr>
<tr>
<td>030C</td>
<td>780  SYS A-reg save</td>
<td></td>
</tr>
<tr>
<td>030D</td>
<td>781  SYS X-reg save</td>
<td></td>
</tr>
<tr>
<td>030E</td>
<td>782  SYS Y-reg save</td>
<td></td>
</tr>
<tr>
<td>030F</td>
<td>783  SYS status reg save</td>
<td></td>
</tr>
<tr>
<td>Address Range</td>
<td>Value Range</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>0310-0312</td>
<td>784-785</td>
<td>USR function jump</td>
</tr>
<tr>
<td>0314-0315</td>
<td>788-789</td>
<td>Hardware interrupt vector</td>
</tr>
<tr>
<td>0316-0317</td>
<td>790-791</td>
<td>Break interrupt vector</td>
</tr>
<tr>
<td>0318-0319</td>
<td>792-793</td>
<td>NMI interrupt vector</td>
</tr>
<tr>
<td>031A-031B</td>
<td>794-795</td>
<td>OPEN vector</td>
</tr>
<tr>
<td>031C-031D</td>
<td>796-797</td>
<td>CLOSE vector</td>
</tr>
<tr>
<td>031E-031F</td>
<td>798-799</td>
<td>Set-input vector</td>
</tr>
<tr>
<td>0320-0321</td>
<td>800-801</td>
<td>Set-output vector</td>
</tr>
<tr>
<td>0322-0323</td>
<td>802-803</td>
<td>Restore I/O vector</td>
</tr>
<tr>
<td>0324-0325</td>
<td>804-805</td>
<td>INPUT vector</td>
</tr>
<tr>
<td>0326-0327</td>
<td>806-807</td>
<td>Output vector</td>
</tr>
<tr>
<td>0328-0329</td>
<td>808-809</td>
<td>Test-STOP vector</td>
</tr>
<tr>
<td>032A-032B</td>
<td>810-811</td>
<td>GET vector</td>
</tr>
<tr>
<td>032C-032D</td>
<td>812-813</td>
<td>Abort I/O vector</td>
</tr>
<tr>
<td>032E-032F</td>
<td>814-815</td>
<td>Warm start vector</td>
</tr>
<tr>
<td>0330-0331</td>
<td>816-817</td>
<td>LOAD link</td>
</tr>
<tr>
<td>0332-0333</td>
<td>818-819</td>
<td>SAVE link</td>
</tr>
<tr>
<td>033C-03FB</td>
<td>828-1019</td>
<td>Cassette buffer</td>
</tr>
<tr>
<td>0340-037E</td>
<td>832-894</td>
<td>(Sprite 13)</td>
</tr>
<tr>
<td>0380-03BE</td>
<td>896-958</td>
<td>(Sprite 14)</td>
</tr>
<tr>
<td>03C0-03FE</td>
<td>960-1022</td>
<td>(Sprite 15)</td>
</tr>
<tr>
<td>0400-07FF</td>
<td>1024-2047</td>
<td>Screen memory</td>
</tr>
<tr>
<td>0800-9FFF</td>
<td>2048-40959</td>
<td>Basic RAM memory</td>
</tr>
<tr>
<td>8000-9FFF</td>
<td>32768-40959</td>
<td>Alternate: ROM plug-in area</td>
</tr>
<tr>
<td>Address Range</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>A000-BFFF</td>
<td>40960-49151 ROM: Basic A000-BFFF 49060-49151 Alternate: RAM</td>
<td></td>
</tr>
<tr>
<td>C000-CFFF</td>
<td>49152-53247 RAM memory, including alternate</td>
<td></td>
</tr>
<tr>
<td>D000-D02E</td>
<td>53248-53294 Video Chip (6566)</td>
<td></td>
</tr>
<tr>
<td>D400-D41C</td>
<td>54272-54300 Sound Chip (6581 SID)</td>
<td></td>
</tr>
<tr>
<td>D800-DFF</td>
<td>55296-56319 Color nybble memory</td>
<td></td>
</tr>
<tr>
<td>D800-DF0F</td>
<td>56320-56335 Interface chip 1, IRQ (6526 CIA)</td>
<td></td>
</tr>
<tr>
<td>DD00-DD1F</td>
<td>56576-56591 Interface chip 2, NMI (6526 CIA)</td>
<td></td>
</tr>
<tr>
<td>D000-DFFF</td>
<td>53248-57343 Alternate: Character set</td>
<td></td>
</tr>
<tr>
<td>E000-FFFF</td>
<td>57344-65535 ROM: Operating System E000-FFFF 57344-65535 Alternate: RAM</td>
<td></td>
</tr>
<tr>
<td>FF81-FFF5</td>
<td>65409-65525 Jump Table, Including:</td>
<td></td>
</tr>
<tr>
<td>FFCC = SET INPUT CHANNEL</td>
<td>FFCC = SET OUTPUT CHANNEL</td>
<td></td>
</tr>
<tr>
<td>FFCC = SET INPUT CHANNEL</td>
<td>FFCC = RESTORE DEFAULT I/O CHANNELS</td>
<td></td>
</tr>
<tr>
<td>FFCC = TEST STOP KEY</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Map 7. Atari Memory.

PAGE ZERO RAM ASSIGNMENTS

0000  LINZBS = $0000 ; LINBUG RAM (WILL BE REPLACED BY MONITOR RAM)

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

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

0010  INTZBS = $0010 ; INTERRUPT HANDLER
0010  POKMSK = $0010 ; SYSTEM MASK FOR POKEY IRQ HANDLER
0011  BRKKEY = $0011 ; BREAK KEY FLAG
0012  RTCLK = $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
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>001A</td>
<td>DSKUTL</td>
<td>$001A ;DISK UTILITIES POINTER</td>
</tr>
<tr>
<td>001C</td>
<td>PTIMOT</td>
<td>$001C ;PRINTER TIME OUT REGISTER</td>
</tr>
<tr>
<td>001D</td>
<td>PBPN'T</td>
<td>$001D ;PRINTER BUFFER POINTER</td>
</tr>
<tr>
<td>001E</td>
<td>PBUFNS</td>
<td>$001E ;PRINT BUFFER SIZE</td>
</tr>
<tr>
<td>001F</td>
<td>PTEMP</td>
<td>$001F ;TEMPORARY REGISTER</td>
</tr>
<tr>
<td>0020</td>
<td>ZIOC</td>
<td>$0020 ;ZERO PAGE I/O CONTROL BLOCK</td>
</tr>
<tr>
<td>0020</td>
<td>IOCBSZ</td>
<td>16 ;NUMBER OF BYTES PER IOCB</td>
</tr>
<tr>
<td>0020</td>
<td>MAXIOC</td>
<td>8*IOCBSZ ;LENGTH OF THE IOCB AREA</td>
</tr>
<tr>
<td>0020</td>
<td>IOCBAS</td>
<td>$0020</td>
</tr>
<tr>
<td>0020</td>
<td>ICHIDZ</td>
<td>$0020 ;HANDLER INDEX NUMBER (FF == IOCB FREE)</td>
</tr>
<tr>
<td>0021</td>
<td>ICDNOZ</td>
<td>$0021 ;DEVICE NUMBER (DRIVE NUMBER)</td>
</tr>
<tr>
<td>0022</td>
<td>ICCOMZ</td>
<td>$0022 ;COMMAND CODE</td>
</tr>
<tr>
<td>0023</td>
<td>ICSTAZ</td>
<td>$0023 ;STATUS OF LAST IOCB ACTION</td>
</tr>
<tr>
<td>0024</td>
<td>ICBALZ</td>
<td>$0024 ;BUFFER ADDRESS LOW BYTE</td>
</tr>
<tr>
<td>0025</td>
<td>ICBAHZ</td>
<td>$0026 ;BUFFER ADDRESS HIGH BYTE</td>
</tr>
<tr>
<td>0026</td>
<td>ICPTLZ</td>
<td>$0026 ;PUT BYTE ROUTINE ADDRESS - 1</td>
</tr>
<tr>
<td>0027</td>
<td>ICPTHZ</td>
<td>$0027</td>
</tr>
<tr>
<td>0028</td>
<td>ICBLZ</td>
<td>$0028 ;BUFFER LENGTH LOW BYTE</td>
</tr>
<tr>
<td>0029</td>
<td>ICBLHZ</td>
<td>$0029</td>
</tr>
<tr>
<td>002A</td>
<td>ICAZ1Z</td>
<td>$002A ;AUXILIARY INFORMATION FIRST BYTE</td>
</tr>
<tr>
<td>002B</td>
<td>ICAZ2Z</td>
<td>$002B</td>
</tr>
<tr>
<td>002C</td>
<td>ICSPRZ</td>
<td>$002C ;TWO SPARE BYTES (CIO LOCAL USE)</td>
</tr>
<tr>
<td>002E</td>
<td>ICIDNO</td>
<td>ICSPRZ+2 ;IOCB NUMBER X 16</td>
</tr>
<tr>
<td>002F</td>
<td>CIOCHR</td>
<td>ICSPRZ+3 ;CHARACTER BYTE FOR CURRENT OPERATION</td>
</tr>
<tr>
<td>0030</td>
<td>STATUS</td>
<td>$0030 ;INTERNAL STATUS STORAGE</td>
</tr>
<tr>
<td>0031</td>
<td>CHKSUM</td>
<td>$0031 ;CHECKSUM (SINGLE BYTE SUM WITH CARRY)</td>
</tr>
<tr>
<td>0032</td>
<td>BUFRLO</td>
<td>$0032 ;POINTER TO DATA BUFFER (LO BYTE)</td>
</tr>
<tr>
<td>0033</td>
<td>BUFRHI</td>
<td>$0033 ;POINTER TO DATA BUFFER (HI BYTE)</td>
</tr>
<tr>
<td>0034</td>
<td>BFENLO</td>
<td>$0034 ;NEXT BYTE PAST END OF DATA BUFFER (LO BYTE)</td>
</tr>
</tbody>
</table>
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  RECVDON  =  $0039  ;RECEIVE 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 VALLE AT COLD START
;REDGE  =  39  

0050  TMPCHR  =  $0050
0051  HOLD1  =  $0051

Appendix B
LMARGN = $0052 ;LEFT MARGIN (SET TO ONE AT POWER ON)
RMARGN = $0053 ;RIGHT MARGIN (SET TO ONE AT POWER ON)
ROWCRS = $0054 ;CURSOR COUNTERS
COLCRS = $0055
DINDEX = $0057
SAVMSC = $0058
OLDROW = $005A
OLDCOL = $005B
OLDCHR = $005D ;DATA UNDER CURSOR
OLDADR = $005E
NEWROW = $0060 ;POINT DRAW GOES TO
NEWCOL = $0061
LOGCOL = $0063 ;POINTS AT COLUMN IN LOGICAL LINE
ADRESS = $0064
MLITMP = $0066
SAVADR = $0068
RAMTOP = $006A ;RAM SIZE DEFINED BY POWER ON LOGIC
BUFCNT = $006B ;BUFFER COUNT
BUFSTR = $006C ;EDITOR GETCH POINTER
BITMSK = $006E ;BIT MASK
; LOTS OF RANDOM TEMPS
SWPFLG = $007B ;NON-O IF TXT AND REGULAR RAM IS SWAPPED
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 ; DEFERRD 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
## Appendix B

<table>
<thead>
<tr>
<th>Light Pen Horizontal Value</th>
<th>Light Pen Vertical Value</th>
<th>Global Priority Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>0234</td>
<td>0235</td>
<td>$026F</td>
</tr>
</tbody>
</table>

### Potentiometers
- PADDL0
- PADDL1
- PADDL2
- PADDL3
- PADDL4
- PADDL5
- PADDL6
- PADDL7

### Joysticks
- STICK0
- STICK1
- STICK2
- STICK3

---

<table>
<thead>
<tr>
<th>0224</th>
<th>0225</th>
<th>026F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0270</td>
<td>0271</td>
<td>0272</td>
</tr>
<tr>
<td>0273</td>
<td>0274</td>
<td>0275</td>
</tr>
<tr>
<td>0276</td>
<td>0277</td>
<td></td>
</tr>
</tbody>
</table>

---

210
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 occurs
02BE SHFLOK $02BE ;Shift lock
02BF BOTSCR $02BF ;Bottom of screen: 24 Norm, 4 Split.

COLORS

PCOLR0 $02C0 ;P0 COLOR
PCOLR1 $02C1 ;P1 COLOR
PCOLR2 $02C2 ;P2 COLOR
PCOLR3 $02C3 ;P3 COLOR
COLOR0 $02C4 ;COLOR 0
COLOR1 $02C5 ;COLOR 1
COLOR2 $02C6 ;COLOR 2
COLOR3 $02C7 ;COLOR 3
COLOR4 $02C8 ;COLOR 4

GLOBAL VARIABLES

RAMSIZ $02E4 ;RAM SIZE (HI BYTE ONLY)
MEMTOP $02E5 ;TOP OF AVAILABLE USER MEMORY
MEMLO $02E7 ;BOTTOM OF AVAILABLE USER MEMORY
DVSTAT $02EA ;STATUS BUFFER

CRSINH $02F0 ;CURSOR INHIBIT (00 = CURSOR ON)
KEYDEL $02F1 ;Key delay
CHACT $02F3 ;CHACTL REGISTER RAM
CHBAS $02F4 ;CHBAS REGISTER RAM
FILDAT $02FD ;RIGHT FILL DATA (DRAW)
02FB ATACHR = $02FB ; ASCII 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 ODEVIC = $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 buffer pointe low
0305 DBUFI = $0305
0306 DTIMLO = $0306 ; Device time out in 1 second units
0308 DBYTLLO = $0308 ; Number of bytes to be transferred low byte
0309 DBYTHI = $0309
030A DAUX1 = $030A ; Command Aux byte 1
030B DAUX2 = $030B
;
0340 IOC = $0340 ; Handler index number (FF = IOC free)
0340 ICID = $0340 ; Device number (drive number)
0341 ICICD = $0341 ; Command code
0343 ICSTA = $0343 ; Status of last IOC action
0344 ICBAI = $0344 ; Buffer address low byte
0345 ICBAH = $0345
0346 ICPTL = $0346 ; Put byte routine address - 1
ICPTH = $0347
ICBLL = $0348 ; Buffer length low byte
ICBLH = $0349
ICAX1 = $034A ; Auxiliary information first byte
ICAX2 = $034B
ICSPR = $034C ; four spare bytes
;
PRNBUF = $03C0 ; Printer buffer (40 bytes)
(21 spare bytes)
;
Page Four Ram Assignments
;
CASBUF = $03FD ; Cassette Buffer (131 bytes)
;
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

AFP = $D800 ; ASCII -> FLOATING POINT (FP)
INBUFF + CIX -> FRO, CIX, CARRY
D8E6 FASC = $D8E6 ; FP -> ASCII FRO -> FRO,FRO+1, CARRY
D9AA IFP = $D9AA ; INTEGER -> FP
;
D92D FPI = $D92D ; FP -> INTEGER FRO -> FRO,FRO+1, CARRY
DA60 FSUB = $DA60 ; FRO <- FRO - FR1, CARRY
DA66 FADD = $DA66 ; FRO <- FRO + FR1 , CARRY
DADB FMUL = $DADB ; FRO <- FRO * FR1 , CARRY
DB28 FDIV = $DB28 ; FRO <- FRO / FR1 , CARRY
DD89 FLOOR = $DD89 ; FLOATING LOAD REGO FRO <- (X,Y)
DD80 FLDOP = $DD80 ; FRO <- (FLPTR)
DD98 FLDIR = $DD98 ; FR1 <- (X,Y)
DD9C FLDIP = $DD9C ; FR1 <- (FLPTR)
DDA7 FSTOR = $DDA7 ; FLOATING STORE REG0 (X,Y) <- FRO
DDA8 FSTOP = $DDA8 ; (FLPTR)<- FRO
DB66 FMOVE = $DB66 ; FR1 <- FRO
DD40 PLYEVL = $DD40 ; FRO <- 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
; FRO = Z

DDC0 EXP = $DDC0 ; FRO <- E**FRO = EXP10(FRO * LOG10(E)) CARRY
DDCC EXP10 = $DDCC ; FRO <- 10**FRO CARRY
DECD LOG = $DECD ; FRO <- LN(FRO) = LOG10(FRO) / LOG10(E) CARRY
DED1 LOG10 = $DED1 ; FRO <- LOG10(FRO) CARRY

; ; THE FOLLOWING ARE IN THE BASIC CARTRIDGE:

BD81 SIN = $BD81 ; FRO <- SIN(FRO) DEGFLG=0 => RADS, 6=>DEG. CARRY
BD73 COS = $BD73 ; FRO <- COS(FRO) CARRY
BD43 ATAN = $BD43 ; FRO <- ATN(FRO) CARRY
BE81 SQR = $BE81 ; FRO <- SQUAREROOT(FRO) CARRY

; ; FLOATING POINT ROUTINES ZERO PAGE (NEEDED ONLY IF F.P. ROUTINES ARE CALLED)

00D4 FRO = $00D4 ; FP REG0
00E0 FR1 = $00E0 ; FP REG1
00F2 CIX = $00F2 ; CURRENT INPUT INDEX
INBUFF = $00F3 ;POINTS TO USER'S LINE INPUT BUFFER
RADFLG = $00FB ;0 = RADIANS, 6 = DEGREES
FLTPTR = $00FC ;POINTS TO USERS FLOATING POINT NUMBER

FLOATING POINT ROUTINES' NON-ZP RAM

(L05E to 05FF)

LBUFF = $0580 ;LINE BUFFER
PLYARG = LBUFF+$60 ;POLYNOMIAL ARGUMENTS

COLLEEN MNEMONICS

POKEY = $D200 ;VBLANK ACTION:
POKEY0 = D200 ;POT0 -> PADDL0
POKEY1 = D201 ;POT1 -> PADDL1
POKEY2 = D202 ;POT2 -> PADDL2
POKEY3 = D203 ;POT3 -> PADDL3
POKEY4 = D204 ;POT4 -> PADDL4
POKEY5 = D205 ;POT5 -> PADDL5
POKEY6 = D206 ;POT6 -> PADDL6
POKEY7 = D207 ;POT7 -> PADDL7

0-227 IN RAM CELL
D208  ALLPOT  =  POKEY+8 ;???
D209  KRCODE  =  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
          ;NONE
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:)
          SKCTL --> SKCTL
D20F  SKCTL  =  POKEY+15 ;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 |

;PCOLRO-->COLPM0 WITH ATTRACT MODE

| D013 | COLPM1  =  CTIA+19 |
| 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 |

;S08-->CONSOL TURN OFF SPEAKER

; D000 M0PF  =  CTIA+0
| D002 M2PF  =  CTIA+2 |
| D003 M3PF  =  CTIA+3 |
D004   POPF   =   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   TRIGO  =   CTIA+16 ;TRIGO-->STRIG1
D011   TRIG1  =   CTIA+17 ;ETC.
D012   TRIG2  =   CTIA+18
D013   TRIG3  =   CTIA+19
D014   PAL    =   CTIA+20

D400   ANTI C  =   $D400
D400   DMAC TL  =   ANTI C+0 ;DMAC TL<--SDMCTL
                          ON OPEN S: OR E:
D401   CHACTL  =   ANTI C+1 ;CHACTL<--CHACT
                          ON OPEN S: OR E:
D402   DLISTL  =   ANTI C+2 ;DLISTL<--SDLSTL
                          ON OPEN S: OR E:
D403   DLISTH  =   ANTI C+3 ;DLISTH<--SDLSTH
                          ON OPEN S: OR E:
D404   HSCROL  =   ANTI C+4
D405   VSCROL  =   ANTI C+5
D407   PMBASE  =   ANTI C+7
D409   CHBASE  =   ANTI C+9 ;CHBASE<--CHBAS
                          ON OPEN S: OR E:
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 ;  6
0006 SETIMM = 6 ; Immediate VBLANK
0007 SETDEF = 7 ; Deferred VBLANK

E45F SYSVBV = $E46F ; SYSTEM VERTICAL BLANK CALCULATIONS
E462 XITVBL = $E462 ; EXIT VERTICAL BLANK CALCULATIONS
E465 SIOINV = $E465 ; SERIAL INPUT OUTPUT INITIALIZATION
For Further Reference


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.

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Simple Assembler</th>
<th>Traditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>LDA #15</td>
<td>LDA #$15</td>
</tr>
<tr>
<td>Absolute</td>
<td>LDA 1500</td>
<td>LDA $1500</td>
</tr>
<tr>
<td>Zero Page</td>
<td>LDA 15</td>
<td>LDA $15 (sometimes LDA *$15)</td>
</tr>
<tr>
<td>Accumulator</td>
<td>ASL</td>
<td>ASL A</td>
</tr>
<tr>
<td>Zero Page, X</td>
<td>LDA 15X</td>
<td>LDA $15,X</td>
</tr>
<tr>
<td>Zero Page, Y</td>
<td>LDX 15Y</td>
<td>LDX $15,Y</td>
</tr>
<tr>
<td>Absolute, X</td>
<td>LDA 1500X</td>
<td>LDA $1500,X</td>
</tr>
<tr>
<td>Absolute, Y</td>
<td>LDA 1500Y</td>
<td>LDA $1500,Y</td>
</tr>
<tr>
<td>Indexed Indirect</td>
<td>LDA (15X)</td>
<td>LDA ($15,X)</td>
</tr>
<tr>
<td>Indirect Indexed</td>
<td>LDA (15)Y</td>
<td>LDA ($15),Y</td>
</tr>
</tbody>
</table>

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,"

224
Appendix C

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 POKE'd 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 POKE'd 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.
Appendix C

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-I. Adding The Conversion Pseudo-op.

245  IFMNS$="HEX"THEN GOTO 7000
246  IFMNS$="DECIMAL"THEN GOTO 7200
7000  PRINT "ENTER DECIMAL NUMBER"; : INPUT DE: IF D
E > 255 THEN SZ = 3: GOTO 7020
7010  SZ = 1
7020  GOSUB 4000: PRINT " $" : 
H$: GOTO 230
7200  PRINT "ENTER HEX NUMBER"; : INPUT H$:
7210  SX = LEN (H$): BK$ = "000": H$ = LEFT$(BK$, 4 - SX) + H$
7220  GOSUB 5000: PRINT "" : DE: GOTO 230

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 DECIMAL
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$(I),3)
140 NEXTI: PRINT
150 PRINT"IMMEDIATE LDA #15
155 PRINT"ABSOLUTE LDA 1500
160 PRINT"ZERO PAGE LDA 15
165 PRINT"ACCUMULATOR ASL
170 PRINT"INDIRECT X LDA (15X)
175 PRINT"INDIRECT Y LDA (15)Y
177 PRINT"ZERO PAGE X LDA 15X
179 PRINT"ZERO PAGE Y LDX 15Y
180 PRINT"ABSOLUTE X LDA 1500X
185 PRINT"ABSOLUTE Y LDA 1500Y
189 PRINT:PRINT" ENTER ALL NUMBERS IN ";
190 IFH=1 THENPRINT"HEX":GOTO200
195 PRINT"DECIMAL"
200 PRINT:PRINT"PLEASE INPUT STARTING ADDRESSES FOR ML PROGRAM":INPUT SA$
210 IFH=1THENH$=SA$:GOSUB5000:SA=DE:GOTO220
215 SA=VAL(SA$)
220 TA=SA:PRINT"{CLEAR}":REM CLEAR THE SCREEN
230 IFH=1THENDE=SA:SZ=3:GOSUB4000:PRINTH$;:GOTO240
235 PRINTSA" ";
240 INPUTMN$:PRINT"{UP}"SPC(20);:REM GO UP ONE LINE AND OVER 20 SPACES
241 REM ADD NEW PSEUDO-OPS HERE
242 IFRIGHT$(MN$,7)="FORWARD"THENFB=SA
243 IFRIGHT$(MN$,7)="RESOLVE"THENFR=SA-FB:POKEFB+1,FR-2:PRINT" OK":GOTO230
244 IFRIGHT$(MN$,4)="POKE"THENPRINT"ADDR,NUMBER(DEC)";:INPUTADR,NUM:POKEADR,NUM
:GOTO230
250 IF MN$="END" THEN PRINT: PRINT "PROGRAM IS FROM " TA TO " SA: END"
260 L = LEN (MN$) : L$ = LEFT$(MN$, 3)
270 FOR I = 1 TO 56 : IFL$ = M$(I) THEN 300
280 NEXT I
290 GOTO 850
300 REM PRIMARY OPCODE CATEGORIES
301 TY = TY(I) : OP = OP(I)
305 IFF B = S A THEN TN$ = 0 : GOTO 2010
310 IFF TY = 0 THEN GOTO 1000
320 IFF TY = 3 THEN TY = 1 : IFL = 3 THEN OP = OP + 8 : GOTO 1000
330 R$ = RIGHT$(MN$, L - 4) : IFH = 1 THEN GOSUB 6000
340 LR$ = LEFT$(R$, 1) : LL = LEN (R$) : IFLR$ = "#" THEN 480
350 IFLR$ = "(" THEN 520
360 IFFY = 8 THEN 600
370 IFF TY = 3 THEN OP = OP + 8 : GOTO 1000
380 IFF RIGHT$(R$, 1) = "X" THEN RIGHT$(R$, 1) = "Y" THEN 630
390 IFF LEFT$(L$, 1) = "J" THEN 820
400 TN = VAL (R$) : IFTN > 255 THEN 430
410 IFFY = 1 OR TY = 3 OR TY = 4 OR TY = 5 THEN OP = OP + 4
420 GOTO 2000
430 H% = TN / 256 : L% = TN - 256 * H% : IFFY = 2 OR TY = 7 THEN OP = OP + 8 : GOTO 470
440 IFFY = 1 OR TY = 3 OR TY = 4 OR TY = 5 THEN OP = OP + 12 : GOTO 4713
450 IFFY = 6 OR TY = 9 THEN 470
460 GOTO 850
470 GOTO 3000
480 TN = VAL (RIGHT$(R$, LL - 1))
490 IFFY = 1 THEN OP = OP + 8 : GOTO 2000
500 IFFY = 4 OR TY = 5 THEN GOTO 2000
510 GOTO 850
520 IFFRIGHT$(R$, 2) = "") THEN 540
530 IFFRIGHT$(R$, 2) = "X" THEN 570
540 TN = VAL (MID$(R$, 2, LL - 3))
550 IFFY = 1 THEN OP = OP + 16 : GOTO 2000
560 GOTO 850
570 TN = VAL(MID$(R$, 2, LL - 3))
580 IFFY = 1 THEN GOTO 2000
590 GOTO 850
600 TN = VAL(R$) : TN = TN - SA - 2 : IFTN < - 128 OR TN > 127 THEN PRINT " TOO FAR " ; GOTO 850
610  IFTN<0THEN TN=TN+256
620  GOTO20000
630  IFRIGHT$(R$,2)="y"THEN 540
640  IFRIGHT$(R$,1)="x"THEN 720
650  REM *ZERO Y
660  TN=VAL(LEFT$(R$, LL-1)): IFTN>255 THEN 680
670  IFTY=2: ORTY=5 THEN 730
675  IFTY=1 THEN 760
680  GOSUB 770: IFTY=1 THEN 770: OP=OP+24: GOTO710
690  IFTY=5 THEN 770: OP=OP+28: GOTO710
700  GOTO850
710  GOTO30000
720  TN=VAL(LEFT$(R$, LL-1)): IFTN>255 THEN GOSUB 770: GOTO780
730  IFTY=2 THEN 770: OP=OP+16: GOTO760
740  IFTY=10: ORTY=3 THEN 770: OP=OP+20: GOTO760
750  GOTO850
760  GOTO20000
770  H% = TN/256: L%= TN-256*H%: RETURN
780  IFTY=2 THEN 770: OP=OP+24: GOTO810
790  IFTY=10: ORTY=3 THEN 770: OP=OP+28: GOTO810
800  GOTO850
810  GOTO30000
820  TN=VAL(R$)
830  GOSUB 770
840  GOTO710
850  PRINT"{REV}\ ERROR ": GOTO230
1000  REM 1 BYTE INSTRUCTIONS
1010  POKESA, OP: SA=SA+1: IFH=1 THEN 1030
1020  PRINTOP: GOTO230
1030  DE = OP: GOSUB40000: PRINTH$: GOTO230
2000  REM 2 BYTE INSTRUCTIONS
2005  IFTN>256 THEN PRINT" INCORRECT ARGUMENT. ( #5 IN HEX IS #05) ": GOTO230
2010  POKESA, OP: POKESA+1, TN: SA=SA+2: IFH=1 THEN 2020
2020  PRINTOP; TN: GOTO230
2030  DE = OP: GOSUB 40000: PRINTH$" ";
2040  DE = TN: GOSUB 40000: PRINTH$: GOTO230
3000  REM 3 BYTE INSTRUCTIONS
3010  POKESA, OP: POKESA+1, L%; POKESA+2, H%: SA=SA+3: IFH=1 THEN 3030
3020  PRINTOP; L%; H%: GOTO230
3030  DE = OP: GOSUB 40000: PRINTH$" ";
Appendix C

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 HS = "": FORM = $T00$STEP - 1: N% = DE / (16 ^ M): DE = D E - N% * 16 ^ M: H$ = H$ + MID$(H$, N% + 1, 1)
4020 NEXT: SZ = 1: RETURN
5000 REM HEX TO DECIMAL (H$ TO DE)
5010 D = 0: Q = 3: FORM = 1 T04: FOR W = 0 TO 15: IF MID$(H$, M + 1, 1) = MID$(HE$, W + 1, 1) THEN 5030
5020 NEXT W
5030 D1 = W * (16 ^ (Q)): D = D + D1: Q = Q - 1: NEXT M: DE = INT(D): RETURN
6000 REM ACCEPT HEX OPCODE INPUT AND TRANSLATE IT TO DECIMAL
6010 IF LEFT$(R$, 1) = "" THEN HS = ":0": PRINT HS + RIGHT$(R$, 1): GOSUB 5000: R$ = ": + STR$(DE): RETURN
6030 IF ZA$ = "Y": THEN HS = ":0": PRINT HS + MID$(R$, 2, 2): GOSUB 5000: R$ = ": + STR$(DE): "Y": RETURN
6040 IF ZA$ = "Y": THEN HS = ":0": PRINT HS + MID$(R$, 2, 2): GOSUB 5000: R$ = ": + STR$(DE): ":X": RETURN
6050 IF ZA$ = "Y": OR ZA$ = "Y": THEN 6070
6060 H$ = LEFT$(Z$O$, 4 - LS) + R$: GOSUB 5000: R$ = STR$(DE): RETURN
6070 FLIS = 5 THEN HS = LEFT$(R$, 4): GOTO 6090
6080 H$ = "00": PRINT H$ + LEFT$(R$, 2)
6090 GOSUB 5000: R$ = STR$(DE) + ZA$: RETURN
20000 DATA ADC1097, AND1033, ASL3002, BCC8144, BCS8176, BEQ8240, BIT7036, BMI8048
20010 DATA BNE8208, BPL8016, BRK0000, BVC8080, BVS8112, CLC0024, CLD0216, CLI0088
20020 DATA CLCV0184, CMP1193, CPX4224, CPY4192, DEC2198, DEO202, DEY136, EOR1065
20030 DATA INCV2230, INX0232, INY0200, JMP6076, JSR9032, LDAl161, LDX5162, LDY5160
20040 DATA LSR3066, NOP0234, ORA1001, PHA0072, PHP008, PLA0104, PLP0040, ROL3034
20050 DATA ORS098, RTI0064, RTS0096, SBC1225, SEC056, SED0248, SEI0120, STA1129
20060 DATA TX2134, STY2132, TAX0170, TAY0168, TSX0186, TXA0138, TXS0154, TYA0152

230

10 HX=1:REM IF HX= 0 THEN ASSEMBLY IS 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$="0123456789ABCDEF":SZ=1:Z0$="000"
100 PRINT "{(3 SPACES) SIMPLE
    (3 SPACES) ASSEMBLER CONVENTIONS
";
110 DIM M$(56*3), TY(56), OP(56)
120 FOR I=1 TO 56:READ MN$:M$(I*3-2,
    I*3)=MN$(I, 3)
122 TY(I)=VAL(MN$(4, 4)):OP(I)=VAL(MN
    $(5))
130 NEXT I
140 PRINT ?
150 PRINT "Immediate{(5 SPACES) LDA #15"
155 PRINT "Absolute{(6 SPACES) LDA 150"
160 PRINT "Zero page{(5 SPACES) LDA 15"
165 PRINT "Accumulator{(3 SPACES) ASL"
170 PRINT "Indirect X{(4 SPACES) LDA (15X)"
175 PRINT "Indirect Y{(4 SPACES) LDA (15Y)"
177 PRINT "Zero page X{(3 SPACES) LDA 15X"
179 PRINT "Zero page Y{(3 SPACES) LDX 15Y"
180 PRINT "Absolute X{(4 SPACES) LDA 1
    500X"
185 PRINT "Absolute Y{(4 SPACES) LDA 1
    500Y"
189 PRINT :PRINT "{(4 SPACES) Enter all
    numbers in ":
190 IF HX=1 THEN PRINT "HEXA";
195 PRINT "decimal"
197 ? :: "Addresses: Use 1536-1791 ($
\text{ \textcolor{red}{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$)
217 IF SA<256 OR SA>40960 THEN ?
\{4 UP\}: Not ZPAGE or ROM! ? : GOTO
200
220 TA=SA : PRINT "\{CLEAR\}" : GOTO 230
225 ? :: "\{BELL\}: INPUT_ERROR" :: ? : IF
HX=1 THEN ? "(e.g. #5 should be
\#05)" :: ?
230 IF HX=1 THEN DE=SA : SZ=3 : GOSUB 40
\text{ \textcolor{red}{00}} : PRINT H$ : :: 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 \text{ \textcolor{red}{8000}}
260 L=LEN(MN$) : L$=MN$ (1,3)
270 FOR I=1 TO 56 : IF L$=M$ (I*3-2,I*3
) THEN \text{ \textcolor{red}{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
320 IF TY=3 THEN TY=1: IF L=3 THEN OP = OP + 8: GOTO 1000
330 R$=MN$(5): IF HX=1 THEN GOSUB 600
340 LR$=R$(1,1): LL=LEN(R$): IF LR$="#" THEN 480
350 IF LR$="(" THEN 520
360 IF TY=9 THEN 600
370 IF TY=3 THEN OP=OP+8: GOTO 1000
380 IF R$(LL)="X" OR R$(LL)="Y" THEN 630
390 IF L$(1,1)="J" THEN 820
400 TN=VAL(R$): IF TN>255 THEN 430
410 IF TY=1 OR TY=3 OR TY=4 OR TY=5 THEN OP=OP+4
420 GOTO 2000
430 H=INT(TN/256): L=(TN-256*H): IF TY =2 OR TY=7 THEN OP=OP+8: GOTO 470
440 IF TY=1 OR TY=3 OR TY=4 OR TY=5 THEN OP=OP+12: GOTO 470
450 IF TY=6 OR TY=9 THEN 470
460 GOTO 850
470 GOTO 3000
480 TN=VAL(R$(2))
490 IF TY=1 THEN OP=OP+8: GOTO 2000
500 IF TY=4 OR TY=5 THEN GOTO 2000
510 GOTO 850
520 IF R$(LL-1)="Y" THEN 540
530 IF R$(LL-1)="X" THEN 570
540 TN=VAL(R$(2,LL-1))
550 IF TY=1 THEN OP=OP+16: GOTO 2000
560 GOTO 850
570 TN=VAL(R$(2,LL-1))
580 IF TY=1 THEN GOTO 2000
590 GOTO 850
600 TN=VAL(R$): TN=TN-5A-2: IF TN<-128 OR TN>127 THEN PRINT "TOO FAR";
610 GOTO 850
620 IF TN<0 THEN TN=TN+256
630 GOTO 2000
640 IF R$(LL-1)="Y" THEN 540
640 IF R$(LL-1)="X" THEN 720
650 REM *ZERO Y
660 TN=VAL(R$(1,LL-1)):IF TN>255 THEN 690
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$(1,LL-1)):IF TN>255 THEN GOSUB 770:GOTO 780
730 IF TY=2 THEN OP=OP+16:GOTO 810
740 IF TY=1 OR TY=3 OR TY=5 THEN OP=OP+28:GOTO 810
750 GOTO 850
760 GOTO 2000
770 H=INT(TN/256):L=TN-256*H:RETURN
780 IF TY=2 THEN OP=OP+24:GOTO 810
790 IF TY=1 OR TY=3 OR TY=5 THEN OP=OP+28:GOTO 810
800 GOTO 850
810 GOTO 3000
820 TN=VAL(R$)
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--";TN;">256 ($100)";GOTO 230
2010 POKE SA,OP:POKE SA+1,TN:SA=SA+2
2015 IF HX=1 THEN 2030
2020 PRINT OP:";TN;GOTO 230
2030 DE=OP:GOSUB 4000:PRINT H$;",";
2040 DE=TN:GOSUB 4000:PRINT H$;GOTO 230
Appendix C

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>9 THEN
   AH=INT(A/16):AL=A-AH*16:H$=HE$(AH+1,AL+1)
4020 A=DE-A*256:AH=INT(A/16):AL=A-AH*16:
   H$(LEN(H$)+1)=HE$(AH+1,AL+1):
   H$(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: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$=Z0$(1,4-LS)
6065 H$(LEN(H$)+1)=R$=GOSUB 5000:R$=
   STR$(DE):RETURN
Appendix C

6070 IF LS=5 THEN H$=R$(1,4):GOTO 60

6080 H$="00":H$(3)=R$(1,2)

6090 GOSUB 5000:R$=STR$(DE):R$(LEN(R$)+1)=ZA$:RETURN


8010 PRINT " ENDS(3 SPACES)":SA:=DE=

20000 DATA ADC1097,AND1033,ASL3002,BCS8144,BCS8176,BEO8240,BIT7036,BMI8048

20010 DATA BNE8208,BPL8016,BRK0000,BVC8080,BVS8112,CLC0024,CLD0216

20020 DATA CLV0184,CMP1193,CPX4224,CY4192,DEC2198,DEX0202,DEY0136

20030 DATA INC2230,INX0232,INY0200,IMP5076,JSR9032,LDA1161,LDX5162

20040 DATA LSR3066,NOP0234,ORA1001,PHA0072,PHP0008,PLA0104,PLP0040

20050 DATA ROR3098,RTI0084,RTS0096,SBC1225,SEC0056,SED0248,SEI0120

20060 DATA STX2134,STY2132,TAX0170,TAY0168,TSX0186,TXA0138,TXS0154

236
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$="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2 LS="---------------------------
   - - - - - - - - - - - - - -
4 JS=" --->"  
13 PRINT" DISASSEMBLER
14 PRINT
16 DIMS$(15,15)
17 FORI=0 TO 15:FORB=0 TO 14:READM$(I,B):NEXTB: NEXI
25 REM START MAIN LOOP
30 PRINT"STARTING ADDRESS (DECIMAL)"; : INPUT 
   SA:TA=SA
31 PRINT"START ADDRESS HEX  ";:DE=SA:ZX=3:G OSUBL200:PRINTH$" "
35 IFSA<0 THEN END
41 I=SA
45 REM PRINT ADDRESS
46 PRINT" ";
50 X=PEEK(I)
55 GOSUB5000
56 IFL%<15ORM$(H%,L%)="0"THENPRINT" ?
      X:CK=0:LN=LN+1:GOTO70
58 PRINM$(H%,L%);
60 GOSUB6000:IFEQTHENEQ=0
70 I=I+1
72 IFLN=20THENLN=0:GOTO2000
80 GOTO45
600 IFCK=12 THEN 603
601 B=PEEK(I+1):IFB>12THENB=((NOTB)AND255)+
     1:B=-B
602 BAD=I+2+B:PRINT" "BAD:I=I+1:RETURN
603 IFH%>8 THEN 800
604 IFH%=2 THEN J=1:GOTO850
605 IFH%=6 THEN PRINT:PRINTL$:EQ=1:RETURN
606 IFH% = 6 THEN RETURN
607 PRINT
608 RETURN
610 IF CK = 12 THEN 615
611 PRINT" ("PEEK(I+1)")", Y"
612 I = I + 1: RETURN
615 PRINT" ("PEEK(I+1)", X")"
616 I = I + 1: RETURN
630 IF CK = 12 THEN 635
631 PRINT" "PEEK(I+1), X"
632 I = I + 1: RETURN
635 PRINT" "PEEK(I+1)
636 I = I + 1: RETURN
640 IF CK = 12 THEN 645
641 PRINT" "PEEK(I+1), X"
642 I = I + 1: RETURN
645 PRINT" "PEEK(I+1)
646 I = I + 1: RETURN
660 IF CK = 12 THEN 665
661 IFH% = 90 RH% = 11 THEN PRINT" "PEEK(I+1)", Y"
662 IFH% = 70 RH% = 150 RH% = 50 RH% = 3 THEN 640
663 IFH% = 13 THEN 631
664 PRINT: GOTO 642
680 PRINT: RETURN
690 IF CK = 12 THEN 680
691 I$ = "Y": GOTO 850
720 IF CK = 12 THEN 725
722 I$ = "X": GOTO 850
725 IFH% = 6 THEN PRINT" (IND. ", ; I = I + 1
726 IFH% = 2 THEN 850
727 IFH% = 4 THEN PRINT J$; GOTO 850
728 IFH% = 80 RH% = 100 RH% = 120 RH% = 14 THEN 850
729 GOTO 610
730 IF CK = 12 THEN 850
731 I$ = "X": GOTO 850
740 IF CK = 12 THEN 850
741 IFH% = 11 THEN I$ = "Y": GOTO 850
742 I$ = "X": GOTO 850
800 PRINT" #"PEEK(I+1)
801 I = I + 1: RETURN
850 N = PEEK(I+1) + PEEK(I+2) * 256
860 IF I$ = "THEN 900
870 IF I$ = "X" THEN PRINT" "N", X"
880 IF I$ = "Y" THEN PRINT" "N", Y"
890 I$="":I=I+2:RETURN
900 PRINT" "N:I=I+2
906 RETURN
1000 DATABASE,ORA,Ø,Ø,Ø,ORA,ASL,Ø,PHP,ORA,ASL,
Ø,Ø,ORA,ASL,BPL,ORA,Ø,Ø,Ø,ORA,ASL
1010 DATAØ,CLC,ORA,Ø,Ø,Ø,ORA,ASL,JSR,AND,Ø,Ø,
BIT,AND,ROL,Ø,PLP,AND,ROL,Ø,BIT
1020 DATAAND,ROL,BMI,AND,Ø,Ø,AND,ROL,Ø,SEC,
AND,Ø,Ø,AND,ROL,RTI,EOR,Ø,Ø,Ø
1030 DATAEOR,LSR,Ø,PHA,EOR,LSR,Ø,JMP,EOR,LSR,
BVC,EOR,Ø,Ø,Ø,EOR,LSR,Ø,CLI,EOR,Ø
1040 DATAØ,Ø,EOR,LSR,RTS,ADC,Ø,Ø,Ø,ADC,ROR,Ø,
PLA,ADC
1045 DATAROR,Ø,JMP,ADC,ROR,BVS,ADC,Ø,Ø,Ø
1050 DATAADC,ROR,Ø,SEI,ADC,Ø,Ø,Ø,ADC,ROR,Ø,STA
1055 DATAØ,Ø,STY,STA,STX,Ø,DEY,Ø,TXA,Ø,STY,STA
1060 DATASTX,BCC,STA,Ø,Ø,STY,STA,STX,Ø,TYA,STA,
TXS,Ø,STA,Ø,LYD,LDA,LDX,Ø
1070 DATALDY,LDA,LDX,Ø,TAY,LDA,TAX,Ø,LYD,LDA,
LDX,BCS,LDA,Ø,Ø,LYD,LDA,LDX,Ø
1080 DATACLV,LDA,TSX,Ø
1090 DATALDY,LDA,LDX,CPR,CMP,Ø,Ø,CPR,CMP,DEC,
Ø,INY,CMP,DEX,Ø,CPR,CMP,DEC
1095 DATABNE,CMP,Ø,Ø,Ø,CMP,DEC,Ø,CLD,CMP,Ø,Ø,
Ø,CMP,DEC,CPX,SBC,Ø,Ø,CPX,SBC,INC
1098 DATAØ,INX,SBC,Ø,CPX,SBC,INC,BEQ,SBC,
Ø,Ø,Ø,SBC,INC,Ø,SED,SBC,Ø,Ø,Ø,SBC
1099 DATAINC
1200 REM MAKE DECIMAL INTO HEX
1201 H$="":FORM=ZXTO0STEP-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
Appendix D


100 REM DISASSEMBLER
105 GRAPHICS 0: POSITION 11,0: ; "FAST DISASSEMBLER": ; "Loading opcodes..."
110 DIM OPCODE$(256*10), LN(255), NB(255), T$(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}Starting 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: GOTO 0 300

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
270 PRINT PEEK(ADDR+1);:POKE 85,15:PRINT PEEK(ADDR+2);:POKE 85,20
280 D$=STR$(PEEK(ADDR+1)+256*PEEK(ADDR+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*(LN(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,CLC,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,BIT ,3,AND ,3,ROL ,3,?,0,BMI,1,AND ()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,RTI,1,EOR (X),2,?,0,?,0,?,0,EOR ,2
570 DATA LSR ,2,?,0,PHA,1,EOR # ,2,LSR ,3,?,0,BPL,1,JSR ,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, ?, 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,
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 $OC hex) and put the $FD MSB together with the $OC LSB for your answer: $FDOC.

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.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex (LSB)</th>
<th>Hex (MSB)</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>256</td>
<td>000000001</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>512</td>
<td>000000100</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>768</td>
<td>000000111</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>1024</td>
<td>000001000</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>1280</td>
<td>000001011</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>1536</td>
<td>000001110</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>1792</td>
<td>000001111</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>2048</td>
<td>000010000</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>2304</td>
<td>000010011</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>2560</td>
<td>000010100</td>
</tr>
</tbody>
</table>


### Appendix E

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex (LSB)</th>
<th>Hex (MSB)</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0B</td>
<td>2816</td>
<td>00001011</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>3072</td>
<td>00001100</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>3328</td>
<td>00001101</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>3584</td>
<td>00001110</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>3840</td>
<td>00001111</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>4096</td>
<td>00010000</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>4352</td>
<td>00010001</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>4608</td>
<td>00010010</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>4864</td>
<td>00010011</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>5120</td>
<td>00010100</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>5376</td>
<td>00010101</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>5632</td>
<td>00010110</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>5888</td>
<td>00010111</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>6144</td>
<td>00011000</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>6400</td>
<td>00011001</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>6656</td>
<td>00011010</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>6912</td>
<td>00011011</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>7168</td>
<td>00011100</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>7424</td>
<td>00011101</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>7680</td>
<td>00011110</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>7936</td>
<td>00011111</td>
</tr>
<tr>
<td>32</td>
<td>20</td>
<td>8192</td>
<td>00100000</td>
</tr>
<tr>
<td>33</td>
<td>21</td>
<td>8448</td>
<td>00100001</td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>8704</td>
<td>00100010</td>
</tr>
<tr>
<td>35</td>
<td>23</td>
<td>8960</td>
<td>00100011</td>
</tr>
<tr>
<td>36</td>
<td>24</td>
<td>9216</td>
<td>00100100</td>
</tr>
<tr>
<td>37</td>
<td>25</td>
<td>9472</td>
<td>00100101</td>
</tr>
<tr>
<td>38</td>
<td>26</td>
<td>9728</td>
<td>00100110</td>
</tr>
<tr>
<td>39</td>
<td>27</td>
<td>9984</td>
<td>00100111</td>
</tr>
<tr>
<td>40</td>
<td>28</td>
<td>10240</td>
<td>00101000</td>
</tr>
<tr>
<td>41</td>
<td>29</td>
<td>10496</td>
<td>00101001</td>
</tr>
<tr>
<td>42</td>
<td>2A</td>
<td>10752</td>
<td>00101010</td>
</tr>
<tr>
<td>43</td>
<td>2B</td>
<td>11008</td>
<td>00101011</td>
</tr>
<tr>
<td>44</td>
<td>2C</td>
<td>11264</td>
<td>00101100</td>
</tr>
<tr>
<td>45</td>
<td>2D</td>
<td>11520</td>
<td>00101101</td>
</tr>
<tr>
<td>46</td>
<td>2E</td>
<td>11776</td>
<td>00101110</td>
</tr>
<tr>
<td>47</td>
<td>2F</td>
<td>12032</td>
<td>00101111</td>
</tr>
<tr>
<td>48</td>
<td>30</td>
<td>12288</td>
<td>00110000</td>
</tr>
<tr>
<td>49</td>
<td>31</td>
<td>12544</td>
<td>00110001</td>
</tr>
<tr>
<td>Decimal</td>
<td>Hex (LSB)</td>
<td>Hex (MSB)</td>
<td>Binary</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>-----------</td>
<td>--------------</td>
</tr>
<tr>
<td>50</td>
<td>32</td>
<td>12800</td>
<td>00110010</td>
</tr>
<tr>
<td>51</td>
<td>33</td>
<td>13056</td>
<td>00110011</td>
</tr>
<tr>
<td>52</td>
<td>34</td>
<td>13312</td>
<td>00110100</td>
</tr>
<tr>
<td>53</td>
<td>35</td>
<td>13568</td>
<td>00110101</td>
</tr>
<tr>
<td>54</td>
<td>36</td>
<td>13824</td>
<td>00110110</td>
</tr>
<tr>
<td>55</td>
<td>37</td>
<td>14080</td>
<td>00110111</td>
</tr>
<tr>
<td>56</td>
<td>38</td>
<td>14336</td>
<td>00111000</td>
</tr>
<tr>
<td>57</td>
<td>39</td>
<td>14592</td>
<td>00111001</td>
</tr>
<tr>
<td>58</td>
<td>3A</td>
<td>14848</td>
<td>00111010</td>
</tr>
<tr>
<td>59</td>
<td>3B</td>
<td>15104</td>
<td>00111011</td>
</tr>
<tr>
<td>60</td>
<td>3C</td>
<td>15360</td>
<td>00111100</td>
</tr>
<tr>
<td>61</td>
<td>3D</td>
<td>15616</td>
<td>00111101</td>
</tr>
<tr>
<td>62</td>
<td>3E</td>
<td>15872</td>
<td>00111110</td>
</tr>
<tr>
<td>63</td>
<td>3F</td>
<td>16128</td>
<td>00111111</td>
</tr>
<tr>
<td>64</td>
<td>40</td>
<td>16384</td>
<td>01000000</td>
</tr>
<tr>
<td>65</td>
<td>41</td>
<td>16640</td>
<td>01000001</td>
</tr>
<tr>
<td>66</td>
<td>42</td>
<td>16896</td>
<td>01000010</td>
</tr>
<tr>
<td>67</td>
<td>43</td>
<td>17152</td>
<td>01000011</td>
</tr>
<tr>
<td>68</td>
<td>44</td>
<td>17408</td>
<td>01000100</td>
</tr>
<tr>
<td>69</td>
<td>45</td>
<td>17664</td>
<td>01000101</td>
</tr>
<tr>
<td>70</td>
<td>46</td>
<td>17920</td>
<td>01000110</td>
</tr>
<tr>
<td>71</td>
<td>47</td>
<td>18176</td>
<td>01000111</td>
</tr>
<tr>
<td>72</td>
<td>48</td>
<td>18432</td>
<td>01001000</td>
</tr>
<tr>
<td>73</td>
<td>49</td>
<td>18688</td>
<td>01001001</td>
</tr>
<tr>
<td>74</td>
<td>4A</td>
<td>18944</td>
<td>01001010</td>
</tr>
<tr>
<td>75</td>
<td>4B</td>
<td>19200</td>
<td>01001011</td>
</tr>
<tr>
<td>76</td>
<td>4C</td>
<td>19456</td>
<td>01001100</td>
</tr>
<tr>
<td>77</td>
<td>4D</td>
<td>19712</td>
<td>01001101</td>
</tr>
<tr>
<td>78</td>
<td>4E</td>
<td>19968</td>
<td>01001110</td>
</tr>
<tr>
<td>79</td>
<td>4F</td>
<td>20224</td>
<td>01001111</td>
</tr>
<tr>
<td>80</td>
<td>50</td>
<td>20480</td>
<td>01010000</td>
</tr>
<tr>
<td>81</td>
<td>51</td>
<td>20736</td>
<td>01010001</td>
</tr>
<tr>
<td>82</td>
<td>52</td>
<td>20992</td>
<td>01010010</td>
</tr>
<tr>
<td>83</td>
<td>53</td>
<td>21248</td>
<td>01010011</td>
</tr>
<tr>
<td>84</td>
<td>54</td>
<td>21504</td>
<td>01010100</td>
</tr>
<tr>
<td>85</td>
<td>55</td>
<td>21760</td>
<td>01010101</td>
</tr>
<tr>
<td>86</td>
<td>56</td>
<td>22016</td>
<td>01010110</td>
</tr>
<tr>
<td>87</td>
<td>57</td>
<td>22272</td>
<td>01010111</td>
</tr>
<tr>
<td>88</td>
<td>58</td>
<td>22528</td>
<td>01011000</td>
</tr>
<tr>
<td>Decimal</td>
<td>Hex (LSB)</td>
<td>Hex (MSB)</td>
<td>Binary</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>-----------</td>
<td>--------------</td>
</tr>
<tr>
<td>89</td>
<td>59</td>
<td>22784</td>
<td>010111001</td>
</tr>
<tr>
<td>90</td>
<td>5A</td>
<td>23040</td>
<td>010111010</td>
</tr>
<tr>
<td>91</td>
<td>5B</td>
<td>23296</td>
<td>010111011</td>
</tr>
<tr>
<td>92</td>
<td>5C</td>
<td>23552</td>
<td>010111000</td>
</tr>
<tr>
<td>93</td>
<td>5D</td>
<td>23808</td>
<td>010111101</td>
</tr>
<tr>
<td>94</td>
<td>5E</td>
<td>24064</td>
<td>01011110</td>
</tr>
<tr>
<td>95</td>
<td>5F</td>
<td>24320</td>
<td>010111111</td>
</tr>
<tr>
<td>96</td>
<td>60</td>
<td>24576</td>
<td>011000000</td>
</tr>
<tr>
<td>97</td>
<td>61</td>
<td>24832</td>
<td>011000001</td>
</tr>
<tr>
<td>98</td>
<td>62</td>
<td>25088</td>
<td>011000101</td>
</tr>
<tr>
<td>99</td>
<td>63</td>
<td>25344</td>
<td>011000111</td>
</tr>
<tr>
<td>100</td>
<td>64</td>
<td>25600</td>
<td>011001000</td>
</tr>
<tr>
<td>101</td>
<td>65</td>
<td>25856</td>
<td>011001010</td>
</tr>
<tr>
<td>102</td>
<td>66</td>
<td>26112</td>
<td>011001011</td>
</tr>
<tr>
<td>103</td>
<td>67</td>
<td>26368</td>
<td>0110010000</td>
</tr>
<tr>
<td>104</td>
<td>68</td>
<td>26624</td>
<td>011001101</td>
</tr>
<tr>
<td>105</td>
<td>69</td>
<td>26880</td>
<td>011001011</td>
</tr>
<tr>
<td>106</td>
<td>6A</td>
<td>27136</td>
<td>011010101</td>
</tr>
<tr>
<td>107</td>
<td>6B</td>
<td>27392</td>
<td>011010101</td>
</tr>
<tr>
<td>108</td>
<td>6C</td>
<td>27648</td>
<td>011011100</td>
</tr>
<tr>
<td>109</td>
<td>6D</td>
<td>27904</td>
<td>011011101</td>
</tr>
<tr>
<td>110</td>
<td>6E</td>
<td>28160</td>
<td>011011110</td>
</tr>
<tr>
<td>111</td>
<td>6F</td>
<td>28416</td>
<td>011011111</td>
</tr>
<tr>
<td>112</td>
<td>70</td>
<td>28672</td>
<td>011100000</td>
</tr>
<tr>
<td>113</td>
<td>71</td>
<td>28928</td>
<td>011100001</td>
</tr>
<tr>
<td>114</td>
<td>72</td>
<td>29184</td>
<td>011101000</td>
</tr>
<tr>
<td>115</td>
<td>73</td>
<td>29440</td>
<td>011110011</td>
</tr>
<tr>
<td>116</td>
<td>74</td>
<td>29696</td>
<td>011110100</td>
</tr>
<tr>
<td>117</td>
<td>75</td>
<td>29952</td>
<td>011110101</td>
</tr>
<tr>
<td>118</td>
<td>76</td>
<td>30208</td>
<td>011111010</td>
</tr>
<tr>
<td>119</td>
<td>77</td>
<td>30464</td>
<td>011111111</td>
</tr>
<tr>
<td>120</td>
<td>78</td>
<td>30720</td>
<td>011111100</td>
</tr>
<tr>
<td>121</td>
<td>79</td>
<td>30976</td>
<td>011110001</td>
</tr>
<tr>
<td>122</td>
<td>7A</td>
<td>31232</td>
<td>011110101</td>
</tr>
<tr>
<td>123</td>
<td>7B</td>
<td>31488</td>
<td>011110111</td>
</tr>
<tr>
<td>124</td>
<td>7C</td>
<td>31744</td>
<td>011111100</td>
</tr>
<tr>
<td>125</td>
<td>7D</td>
<td>32000</td>
<td>011111101</td>
</tr>
<tr>
<td>126</td>
<td>7E</td>
<td>32256</td>
<td>011111110</td>
</tr>
<tr>
<td>127</td>
<td>7F</td>
<td>32512</td>
<td>011111111</td>
</tr>
</tbody>
</table>
### Appendix E

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex (LSB)</th>
<th>Hex (MSB)</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>80</td>
<td>32768</td>
<td>100000000</td>
</tr>
<tr>
<td>129</td>
<td>81</td>
<td>33024</td>
<td>100000001</td>
</tr>
<tr>
<td>130</td>
<td>82</td>
<td>33280</td>
<td>100000100</td>
</tr>
<tr>
<td>131</td>
<td>83</td>
<td>33536</td>
<td>100000111</td>
</tr>
<tr>
<td>132</td>
<td>84</td>
<td>33792</td>
<td>100001100</td>
</tr>
<tr>
<td>133</td>
<td>85</td>
<td>34048</td>
<td>100010111</td>
</tr>
<tr>
<td>134</td>
<td>86</td>
<td>34304</td>
<td>100010110</td>
</tr>
<tr>
<td>135</td>
<td>87</td>
<td>34560</td>
<td>100010101</td>
</tr>
<tr>
<td>136</td>
<td>88</td>
<td>34816</td>
<td>100010100</td>
</tr>
<tr>
<td>137</td>
<td>89</td>
<td>35072</td>
<td>100010101</td>
</tr>
<tr>
<td>138</td>
<td>8A</td>
<td>35328</td>
<td>100010110</td>
</tr>
<tr>
<td>139</td>
<td>8B</td>
<td>35584</td>
<td>100010111</td>
</tr>
<tr>
<td>140</td>
<td>8C</td>
<td>35840</td>
<td>100010111</td>
</tr>
<tr>
<td>141</td>
<td>8D</td>
<td>36096</td>
<td>100011011</td>
</tr>
<tr>
<td>142</td>
<td>8E</td>
<td>36352</td>
<td>100011100</td>
</tr>
<tr>
<td>143</td>
<td>8F</td>
<td>36608</td>
<td>100011111</td>
</tr>
<tr>
<td>144</td>
<td>90</td>
<td>36864</td>
<td>100100000</td>
</tr>
<tr>
<td>145</td>
<td>91</td>
<td>37120</td>
<td>100100011</td>
</tr>
<tr>
<td>146</td>
<td>92</td>
<td>37376</td>
<td>100101010</td>
</tr>
<tr>
<td>147</td>
<td>93</td>
<td>37632</td>
<td>100101011</td>
</tr>
<tr>
<td>148</td>
<td>94</td>
<td>37888</td>
<td>100101100</td>
</tr>
<tr>
<td>149</td>
<td>95</td>
<td>38144</td>
<td>100101101</td>
</tr>
<tr>
<td>150</td>
<td>96</td>
<td>38400</td>
<td>100101110</td>
</tr>
<tr>
<td>151</td>
<td>97</td>
<td>38656</td>
<td>100101111</td>
</tr>
<tr>
<td>152</td>
<td>98</td>
<td>38912</td>
<td>100110000</td>
</tr>
<tr>
<td>153</td>
<td>99</td>
<td>39168</td>
<td>100110001</td>
</tr>
<tr>
<td>154</td>
<td>9A</td>
<td>39424</td>
<td>100110100</td>
</tr>
<tr>
<td>155</td>
<td>9B</td>
<td>39680</td>
<td>100110111</td>
</tr>
<tr>
<td>156</td>
<td>9C</td>
<td>39936</td>
<td>100111100</td>
</tr>
<tr>
<td>157</td>
<td>9D</td>
<td>40192</td>
<td>100111101</td>
</tr>
<tr>
<td>158</td>
<td>9E</td>
<td>40448</td>
<td>100111110</td>
</tr>
<tr>
<td>159</td>
<td>9F</td>
<td>40704</td>
<td>100111111</td>
</tr>
<tr>
<td>160</td>
<td>A0</td>
<td>40960</td>
<td>101000000</td>
</tr>
<tr>
<td>161</td>
<td>A1</td>
<td>41216</td>
<td>101000001</td>
</tr>
<tr>
<td>162</td>
<td>A2</td>
<td>41472</td>
<td>101000010</td>
</tr>
<tr>
<td>163</td>
<td>A3</td>
<td>41728</td>
<td>101000111</td>
</tr>
<tr>
<td>164</td>
<td>A4</td>
<td>41984</td>
<td>101001000</td>
</tr>
<tr>
<td>165</td>
<td>A5</td>
<td>42240</td>
<td>101001011</td>
</tr>
<tr>
<td>166</td>
<td>A6</td>
<td>42496</td>
<td>101001100</td>
</tr>
<tr>
<td>Decimal</td>
<td>Hex (LSB)</td>
<td>Hex (MSB)</td>
<td>Binary</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>-----------</td>
<td>------------</td>
</tr>
<tr>
<td>167</td>
<td>A7</td>
<td>42752</td>
<td>10100111</td>
</tr>
<tr>
<td>168</td>
<td>A8</td>
<td>43008</td>
<td>10101000</td>
</tr>
<tr>
<td>169</td>
<td>A9</td>
<td>43264</td>
<td>10101001</td>
</tr>
<tr>
<td>170</td>
<td>AA</td>
<td>43520</td>
<td>10101010</td>
</tr>
<tr>
<td>171</td>
<td>AB</td>
<td>43776</td>
<td>10101011</td>
</tr>
<tr>
<td>172</td>
<td>AC</td>
<td>44032</td>
<td>10101100</td>
</tr>
<tr>
<td>173</td>
<td>AD</td>
<td>44288</td>
<td>10101101</td>
</tr>
<tr>
<td>174</td>
<td>AE</td>
<td>44544</td>
<td>10101110</td>
</tr>
<tr>
<td>175</td>
<td>AF</td>
<td>44800</td>
<td>10101111</td>
</tr>
<tr>
<td>176</td>
<td>B0</td>
<td>45056</td>
<td>10110000</td>
</tr>
<tr>
<td>177</td>
<td>B1</td>
<td>45312</td>
<td>10110001</td>
</tr>
<tr>
<td>178</td>
<td>B2</td>
<td>45568</td>
<td>10110010</td>
</tr>
<tr>
<td>179</td>
<td>B3</td>
<td>45824</td>
<td>10110011</td>
</tr>
<tr>
<td>180</td>
<td>B4</td>
<td>46080</td>
<td>10110100</td>
</tr>
<tr>
<td>181</td>
<td>B5</td>
<td>46336</td>
<td>10110101</td>
</tr>
<tr>
<td>182</td>
<td>B6</td>
<td>46592</td>
<td>10110110</td>
</tr>
<tr>
<td>183</td>
<td>B7</td>
<td>46848</td>
<td>10110111</td>
</tr>
<tr>
<td>184</td>
<td>B8</td>
<td>47104</td>
<td>10111000</td>
</tr>
<tr>
<td>185</td>
<td>B9</td>
<td>47360</td>
<td>10111001</td>
</tr>
<tr>
<td>186</td>
<td>BA</td>
<td>47616</td>
<td>10111010</td>
</tr>
<tr>
<td>187</td>
<td>BB</td>
<td>47872</td>
<td>10111011</td>
</tr>
<tr>
<td>188</td>
<td>BC</td>
<td>48128</td>
<td>10111100</td>
</tr>
<tr>
<td>189</td>
<td>BD</td>
<td>48384</td>
<td>10111101</td>
</tr>
<tr>
<td>190</td>
<td>BE</td>
<td>48640</td>
<td>10111110</td>
</tr>
<tr>
<td>191</td>
<td>BF</td>
<td>48896</td>
<td>10111111</td>
</tr>
<tr>
<td>192</td>
<td>C0</td>
<td>49152</td>
<td>11000000</td>
</tr>
<tr>
<td>193</td>
<td>C1</td>
<td>49408</td>
<td>11000001</td>
</tr>
<tr>
<td>194</td>
<td>C2</td>
<td>49664</td>
<td>11000010</td>
</tr>
<tr>
<td>195</td>
<td>C3</td>
<td>49920</td>
<td>11000011</td>
</tr>
<tr>
<td>196</td>
<td>C4</td>
<td>50176</td>
<td>11000100</td>
</tr>
<tr>
<td>197</td>
<td>C5</td>
<td>50432</td>
<td>11000101</td>
</tr>
<tr>
<td>198</td>
<td>C6</td>
<td>50688</td>
<td>11000110</td>
</tr>
<tr>
<td>199</td>
<td>C7</td>
<td>50944</td>
<td>11000111</td>
</tr>
<tr>
<td>200</td>
<td>C8</td>
<td>51200</td>
<td>11001000</td>
</tr>
<tr>
<td>201</td>
<td>C9</td>
<td>51456</td>
<td>11001001</td>
</tr>
<tr>
<td>202</td>
<td>CA</td>
<td>51712</td>
<td>11001010</td>
</tr>
<tr>
<td>203</td>
<td>CB</td>
<td>51968</td>
<td>11001011</td>
</tr>
<tr>
<td>204</td>
<td>CC</td>
<td>52224</td>
<td>11001100</td>
</tr>
</tbody>
</table>
## Appendix E

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex (LSB)</th>
<th>Hex (MSB)</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>205</td>
<td>CD</td>
<td>52480</td>
<td>11001101</td>
</tr>
<tr>
<td>206</td>
<td>CE</td>
<td>52736</td>
<td>11001110</td>
</tr>
<tr>
<td>207</td>
<td>CF</td>
<td>52992</td>
<td>11001111</td>
</tr>
<tr>
<td>208</td>
<td>D0</td>
<td>53248</td>
<td>11010000</td>
</tr>
<tr>
<td>209</td>
<td>D1</td>
<td>53504</td>
<td>11010001</td>
</tr>
<tr>
<td>210</td>
<td>D2</td>
<td>53760</td>
<td>11010010</td>
</tr>
<tr>
<td>211</td>
<td>D3</td>
<td>54016</td>
<td>11010011</td>
</tr>
<tr>
<td>212</td>
<td>D4</td>
<td>54272</td>
<td>11010100</td>
</tr>
<tr>
<td>213</td>
<td>D5</td>
<td>54528</td>
<td>11010101</td>
</tr>
<tr>
<td>214</td>
<td>D6</td>
<td>54784</td>
<td>11010110</td>
</tr>
<tr>
<td>215</td>
<td>D7</td>
<td>55040</td>
<td>11010111</td>
</tr>
<tr>
<td>216</td>
<td>D8</td>
<td>55296</td>
<td>11011000</td>
</tr>
<tr>
<td>217</td>
<td>D9</td>
<td>55552</td>
<td>11011001</td>
</tr>
<tr>
<td>218</td>
<td>DA</td>
<td>55808</td>
<td>11011010</td>
</tr>
<tr>
<td>219</td>
<td>DB</td>
<td>56064</td>
<td>11011011</td>
</tr>
<tr>
<td>220</td>
<td>DC</td>
<td>56320</td>
<td>11011100</td>
</tr>
<tr>
<td>221</td>
<td>DD</td>
<td>56576</td>
<td>11011101</td>
</tr>
<tr>
<td>222</td>
<td>DE</td>
<td>56832</td>
<td>11011110</td>
</tr>
<tr>
<td>223</td>
<td>DF</td>
<td>57088</td>
<td>11011111</td>
</tr>
<tr>
<td>224</td>
<td>E0</td>
<td>57344</td>
<td>11100000</td>
</tr>
<tr>
<td>225</td>
<td>E1</td>
<td>57600</td>
<td>11100001</td>
</tr>
<tr>
<td>226</td>
<td>E2</td>
<td>57856</td>
<td>11100010</td>
</tr>
<tr>
<td>227</td>
<td>E3</td>
<td>58112</td>
<td>11100011</td>
</tr>
<tr>
<td>228</td>
<td>E4</td>
<td>58368</td>
<td>11100100</td>
</tr>
<tr>
<td>229</td>
<td>E5</td>
<td>58624</td>
<td>11100101</td>
</tr>
<tr>
<td>230</td>
<td>E6</td>
<td>58880</td>
<td>11100110</td>
</tr>
<tr>
<td>231</td>
<td>E7</td>
<td>59136</td>
<td>11100111</td>
</tr>
<tr>
<td>232</td>
<td>E8</td>
<td>59392</td>
<td>11101000</td>
</tr>
<tr>
<td>233</td>
<td>E9</td>
<td>59648</td>
<td>11101001</td>
</tr>
<tr>
<td>234</td>
<td>EA</td>
<td>59904</td>
<td>11101010</td>
</tr>
<tr>
<td>235</td>
<td>EB</td>
<td>60160</td>
<td>11101011</td>
</tr>
<tr>
<td>236</td>
<td>EC</td>
<td>60416</td>
<td>11101100</td>
</tr>
<tr>
<td>237</td>
<td>ED</td>
<td>60672</td>
<td>11101101</td>
</tr>
<tr>
<td>238</td>
<td>EE</td>
<td>60928</td>
<td>11101110</td>
</tr>
<tr>
<td>239</td>
<td>EF</td>
<td>61184</td>
<td>11101111</td>
</tr>
<tr>
<td>240</td>
<td>F0</td>
<td>61440</td>
<td>11110000</td>
</tr>
<tr>
<td>241</td>
<td>F1</td>
<td>61696</td>
<td>11110001</td>
</tr>
<tr>
<td>242</td>
<td>F2</td>
<td>61952</td>
<td>11110010</td>
</tr>
</tbody>
</table>
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-I. Microsoft Table Printer.**

```basic
10 OPEN4,4:REM OPEN CHANNEL TO PRINTER
100 HE$="0123456789ABCDEF"
110 FOR X=1 TO 255
120 B=2:C=1
122 IF X<10 THEN PRINT#4," ";:GOTO130
124 IF X<100 THEN PRINT#4," ";
130 PRINT#4,X;" ";:DE=X:GOSUB240
135 REM CREATE BINARY
140 IFXAND1THENK$(C)="1":GOTO160
150 K$(C)="0"
160 C=C+1:IF BANDXTHENK$(C)="1":GOTO180
170 K$(C)="0"
180 B=B*2:IF C>8 THEN 200
190 GOTO160
200 FOR I=8 TO 1 STEP-1:PRINT#4,K$(I);:NEXT I
```
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 0600 0648
.: 0600 28 58 FF FF 00 0B 06 AD
.: 0608 FF FC 00 21 06 03 AD A9
.: 0610 CB 85 1F A9 0C 85 20 A5
.: 0618 34 85 21 A5 35 85 22 A0
.: 0620 00 93 06 06 00 16 20 38
.: 0628 06 F0 11 85 23 20 38 06
.: 0630 18 65 34 AA A5 23 65 35
.: 0638 20 43 06 8A 20 43 06 20
.: 0640 50 06 90 DB 60 EA EA A5
.: 0648 1F D0 02 C6 20 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 finished 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 0600, 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",01,7A33,8000
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 $A000,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
```
Appendix F

Go to address 1000 hex and begin running code.

Hunt memory
\[ .H \text{C000 D000 'READ} \]
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.
\[ .H \text{C000 D000 20 D2 FF} \]
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.
\[ .L "\text{RAM TEST}" \]
Load from cassette #1 the program named RAM Test.
\[ .L "\text{RAM TEST},08 \]
Load from disk (device 8) the program named RAM Test.
This command leaves basic pointers unchanged.

Memory display
\[ .M \text{0000 0080} \]
\[ : \text{0000 00 01 02 03 04 05 06 07} \]
\[ : \text{0008 08 09 0A 0B 0C 0D 0E 0F} \]
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
\[ : \text{0000 E62E 01 02 03 04 05} \]
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",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",08,1200,1F50  
SAVE TO DISK DRIVE #0 MEMORY FROM 1200 HEX UP TO BUT NOT INCLUDING 1F50 HEX AND NAME IT PROGRAM NAME.

TRANSFER MEMORY
.T 1000 1100 5000  
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 I. CONTROL.

100 PRINT"{CLEAR}{Ø2 DOWN}{REV} SUP ERMON!!"
110 PRINT"{DOWN} DISSASSEMBLER ~ {REV}{OFF} BY WOZNIAK/BAU M
120 PRINT" SINGLE STEP {REV}{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  \( L = PEEK(52) + PEEK(53) \times 256 : SYS1536 \)
     \( : M = PEEK(33) : N = PEEK(34) \)
180  \( POKE52,M:POKE53,N:POKE48,M:POKE \)
     \( 49,N:N=M+N\times256 \)
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  
     \( \%\)\times16:IFX%>9THENX%=X%+7
260  PRINTCHR$(X%+48);:NEXTJ:RETURN

Program 2. SUPERMON 4.0

```assembly
. 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
```
Appendix F

```plaintext
07E0 98 D7 C9 27 D0 14 20 98
07E8 D7 9D 10 02 E8 20 CF FF
07F0 C9 0D F0 22 E0 20 D0 F1
07F8 F0 1C 8E 00 00 01 20 6B
0800 D7 90 C6 9D 10 02 E8 20
0808 CF FF C9 0D F0 09 20 63
0810 D7 90 B6 E0 20 D0 EC 86
0818 B4 20 34 D5 A2 00 00 A0
0820 00 00 B1 FB DD 10 02 D0
0828 0C C8 8E E0 20 00 F1
0830 17 D7 20 31 D5 20 39 D5
0838 A6 DE D0 92 20 CA FA 00
0840 B0 DD 4C BA D4 20 81 FA
0848 00 8D 0D 02 A5 FC 8D 0E
0850 02 A9 04 A2 00 00 8D 09
0858 02 8E 0A 02 A9 93 20 D2
0860 FF A9 16 85 B5 20 06 FC
0868 00 20 64 FC 00 85 FB 84
0870 FC C6 B5 D0 F2 A9 91 20
0878 D2 FF 4C BA D4 A0 2C 20
0880 79 D5 20 17 D7 20 31 D5
0888 A2 00 00 A1 FB 20 74 FC
0890 00 48 20 BB FC 00 68 20
0898 D3 FC 00 A2 06 E0 03 D0
08A0 13 AC 1C 02 F0 0E A5 FF
08A8 C9 E8 B1 FB B0 1C 20 5C
08B0 FC 00 88 D0 F2 06 FF 90
08B8 0E BD 51 FF 00 20 45 FD
08C0 00 BD 57 FF 00 F0 03 20
08C8 45 FD 00 CA D0 D4 60 20
08D0 68 FC 00 AA E8 D0 01 C8
08D8 98 20 5C FC 00 8A 86 B4
08E0 20 22 D7 A6 B4 60 AD 1C
08E8 02 38 A4 FC AA 10 01 88
08F0 65 FB 90 01 C8 60 A8 4A
08F8 90 0B 4A B0 17 C9 22 F0
0900 13 29 07 09 80 4A AA BD
0908 00 FF 00 B0 04 4A 4A 4A
```
Appendix F

```
. . . : 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 8E 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
. . . : 0958 0B 02 B9 9E FF 00 8D 0C
. . . : 0960 02 A9 00 00 A0 05 0E 0C
. . . : 0968 02 2E 0B 02 2A 88 D0 F6
. . . : 0968 69 3F 20 D2 FF CA D0 EA
. . . : 0980 4C 31 D5 20 81 FA 00 20
. . . : 0980 44 D7 20 92 FA 00 20 44
. . . : 0990 0D 85 0A 20 02 8D 09
. . . : 0998 02 8E 0A 02 20 34 D5 20
. . . : 09A0 0B FC 00 20 64 FC 00 85
. . . : 09A0 68 80 02 08 02 68 80 04 02
. . . : 09B0 20 CA FA 00 B0 9E 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
```
Program 3. Changes For SUPERMON 3.0.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0C90</td>
<td>26</td>
<td>72</td>
<td>72</td>
<td>88</td>
</tr>
<tr>
<td>0C98</td>
<td>48</td>
<td>44</td>
<td>44</td>
<td>A2</td>
</tr>
<tr>
<td>0CA0</td>
<td>44</td>
<td>50</td>
<td>2C</td>
<td>41</td>
</tr>
<tr>
<td>0CA8</td>
<td>DB</td>
<td>FA</td>
<td>00</td>
<td>30</td>
</tr>
<tr>
<td>0CB0</td>
<td>00</td>
<td>D1</td>
<td>FB</td>
<td>00</td>
</tr>
<tr>
<td>0CB8</td>
<td>FD</td>
<td>00</td>
<td>D4</td>
<td>FD</td>
</tr>
<tr>
<td>0CC0</td>
<td>B9</td>
<td>D4</td>
<td>7F</td>
<td>FD</td>
</tr>
<tr>
<td>0CC8</td>
<td>33</td>
<td>FA</td>
<td>00</td>
<td>AA</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>06D0</td>
<td>20</td>
<td>EB</td>
</tr>
<tr>
<td>06E0</td>
<td>79</td>
<td>FA</td>
</tr>
<tr>
<td>06E8</td>
<td>E7</td>
<td>90</td>
</tr>
<tr>
<td>06F0</td>
<td>A7</td>
<td>E7</td>
</tr>
<tr>
<td>06F8</td>
<td>4C</td>
<td>F7</td>
</tr>
<tr>
<td>0738</td>
<td>81</td>
<td>FA</td>
</tr>
<tr>
<td>0748</td>
<td>FA</td>
<td>00</td>
</tr>
<tr>
<td>0750</td>
<td>E7</td>
<td>90</td>
</tr>
<tr>
<td>0760</td>
<td>FD</td>
<td>20</td>
</tr>
<tr>
<td>0798</td>
<td>00</td>
<td>20</td>
</tr>
<tr>
<td>07A0</td>
<td>20</td>
<td>92</td>
</tr>
<tr>
<td>07A8</td>
<td>EB</td>
<td>E7</td>
</tr>
<tr>
<td>07C0</td>
<td>D5</td>
<td>FD</td>
</tr>
<tr>
<td>07C8</td>
<td>4C</td>
<td>56</td>
</tr>
<tr>
<td>07D0</td>
<td>97</td>
<td>E7</td>
</tr>
<tr>
<td>07D8</td>
<td>E7</td>
<td>20</td>
</tr>
<tr>
<td>07E0</td>
<td>EB</td>
<td>E7</td>
</tr>
<tr>
<td>07E8</td>
<td>E7</td>
<td>9D</td>
</tr>
<tr>
<td>07F0</td>
<td>F0</td>
<td>1C</td>
</tr>
<tr>
<td>0800</td>
<td>E7</td>
<td>90</td>
</tr>
<tr>
<td>0808</td>
<td>CF</td>
<td>FF</td>
</tr>
<tr>
<td>0810</td>
<td>E7</td>
<td>90</td>
</tr>
<tr>
<td>0818</td>
<td>B4</td>
<td>20</td>
</tr>
<tr>
<td>0830</td>
<td>6A</td>
<td>E7</td>
</tr>
<tr>
<td>0840</td>
<td>B0</td>
<td>DD</td>
</tr>
</tbody>
</table>

266
Program 4. SUPERMON 4.0 Checksum.

100 REM SUPERMON 4 CHECKSUM
110 DATA7331,12186,10071,10387,1082
9,9175,10314,9823,9715,871
4,8852
120 DATA8850,9748,7754,10247,10423,
10948,10075,6093,5492,7805
:S=1536
Appendix F

130 FORB=1TO21:READX:FORI=STOS+79:N =PEEK(I):Y=Y+N
140 NEXTI:IFY<>X THENPRINT"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 DATA7331,12186,10467,10880,11124,10005,10906,10196,9951,8813
120 DATA8852,9329,10239,8457,10334,10423,11047,10311,6093,5492,7805: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 #S12
.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
Appendix F

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 CO00 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 CO00 D000 20 D2 FF
Hunt memory from C000 hex to D000 hex for the sequence of bytes 20 D2 FF and print the address. 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 1FBO 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 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
Appendix F

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

.54142 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 AFF 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
Appendix F

% 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.

<table>
<thead>
<tr>
<th>Location</th>
<th>Old Value</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>X3E7</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>X3EC</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>X3F6</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>X427</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>XD18</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>XDA3</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>XCFC</td>
<td>28</td>
<td>50</td>
</tr>
<tr>
<td>XD7B</td>
<td>28</td>
<td>50</td>
</tr>
<tr>
<td>XE16</td>
<td>83</td>
<td>87</td>
</tr>
<tr>
<td>XE20</td>
<td>28</td>
<td>50</td>
</tr>
<tr>
<td>XE24</td>
<td>C0</td>
<td>80</td>
</tr>
<tr>
<td>XE26</td>
<td>04</td>
<td>08</td>
</tr>
<tr>
<td>XE37</td>
<td>27</td>
<td>4F</td>
</tr>
<tr>
<td>XE46</td>
<td>28</td>
<td>50</td>
</tr>
<tr>
<td>X681</td>
<td>24</td>
<td>00</td>
</tr>
</tbody>
</table>

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
.

Sets a form feed for printout. Gives 57 printed lines per page. Works with the Shift D and Shift M commands.
.
.1 "Heading"

Sets form feed with a message to be printed at the top of each page.
.
.1 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.
Appendix F

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.
Appendix F

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 15463,14894,14290,11897,12453,13919,14116,11715,1257
20 DATA 13693,11853,12903,14513,12137,15006,12654,13291,1243
30 DATA 15366,9999,11834,13512,12892,14475,15149,14896,15782
40 DATA 12171,8985
100 Q=4096
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 #"BLOCK:GOTO170
165 PRINT" BLOCK" IS CORRECT
170 CK=0:Q=Q+128
180 NEXT BLOCK
```

Program 2. Micromon.

```
1000 4C 0C 10 4C 6F 10 4C CF
1008 FF 4C D2 FF 78 A5 92 A6
1010 93 8D E5 02 8E E6 02 AD
1018 F6 1F AE F7 1F 8D E3 02
1020 8E E4 02 AD F0 1F AE F1
1028 1F 85 92 8E 93 A5 90 A6
1030 91 CD EE 1F D0 05 EC EF
1038 1F F0 10 8D 9E 02 8E 9F
1040 02 AD EE 1F AE EF 1F 85
1048 90 86 91 AD EC 1F AE ED
1050 1F EE 80 B0 08 85 34 86
1058 35 85 30 86 31 A9 10 8D
1060 84 02 8D 85 02 A9 00 8D
```
Appendix F

1068 86 02 8D A2 02 58 00 38
1070 AD 7B 02 E9 01 8D 7B 02
1078 AD 7A 02 E9 00 8D 7A 02

1080 20 55 19 A2 42 A9 2A 20
1088 29 18 A9 52 D0 23 A9 3F
1090 20 09 10 20 55 19 A9 2E
1098 20 09 10 A9 00 8D 94 02
10A0 8D A2 02 A2 FF 9A 20 A4
10A8 18 C9 2E F0 F9 C9 20 F0
10B0 F5 A2 1D DD 92 1F D0 13
10B8 8D 87 02 8A 0A AA BD B0
10C0 1F 85 FB BD B1 1F 85 FC
10C8 6C FB 00 CA 10 E5 6C E3
10D0 02 A2 02 D0 02 A2 00 B4
10D8 FB D0 09 B4 FC D0 03 EE
10E0 94 02 D6 FC D6 FB 60 A9
10E8 00 8D 8C 02 20 4F 12 A2
10F0 09 20 52 19 CA D0 FA 60
10F8 A2 02 B5 FA 48 BD 91 02

1100 95 FA 68 9D 91 02 CA D0
1108 F1 60 AD 92 02 AC 93 02
1110 4C 17 11 A5 FD A4 FE 38
1118 E5 FB 8D 91 02 98 E5 FC
1120 A8 0D 91 02 60 A9 00 F0
1128 02 A9 01 8D 95 02 20 E6
1130 17 20 55 19 20 13 11 20
1138 3C 18 90 1B 20 0A 11 B0
1140 03 4C C5 11 20 7F 11 E6
1148 FD D0 02 E6 FE 20 3B 19
1150 AC 94 02 D0 45 F0 E5 20
1158 0A 11 18 AD 91 02 65 FD
1160 85 FD 98 65 FE 85 FE 20
1168 F8 10 20 7F 11 20 0A 11
1170 B0 53 20 D1 10 20 D5 10
1178 AC 94 02 D0 1D F0 EB A2
| 1180 | 00 | A1 | FB | AC | 95 | 02 | F0 | 02 |
| 1188 | 31 | FD | C1 | FD | F0 | 0B | 20 | 13 |
| 1190 | 18 | 20 | 52 | 19 | 20 | AE | 18 | F0 |
| 1198 | 01 | 60 | 4C | 93 | 10 | 20 | 01 | 18 |
| 11A0 | 20 | 0B | 18 | 20 | A4 | 18 | 20 | 6F |
| 11A8 | 18 | 90 | 17 | 8D | 89 | 02 | AE | 94 |
| 11B0 | 02 | D0 | 12 | 20 | 13 | 11 | 90 | 0D |
| 11B8 | AD | 89 | 02 | 81 | FB | 20 | 3B | 19 |
| 11C0 | D0 | EC | 4C | 8E | 10 | 4C | 93 | 10 |
| 11C8 | 20 | 01 | 18 | 20 | 0B | 18 | 20 | A4 |
| 11D0 | 18 | A2 | 00 | 20 | A4 | 18 | C9 | 27 |
| 11D8 | D0 | 14 | 20 | A4 | 18 | 9D | A3 | 02 |
| 11E0 | E8 | 20 | 06 | 10 | C9 | 0D | F0 | 22 |
| 11E8 | E0 | 20 | D0 | F1 | F0 | 1C | 8E | 97 |
| 11F0 | 02 | 20 | 77 | 18 | 90 | CC | 9D | A3 |
| 11F8 | 02 | E8 | 20 | 06 | 10 | C9 | 0D | F0 |

| 1200 | 09 | 20 | 6F | 18 | 90 | BC | E0 | 20 |
| 1208 | D0 | EC | 8E | 88 | 02 | 20 | 55 | 19 |
| 1210 | A2 | 00 | A0 | 00 | B1 | FB | DD | A3 |
| 1218 | 02 | D0 | 0A | C8 | E8 | EC | 88 | 02 |
| 1220 | D0 | F2 | 20 | 8E | 11 | 20 | 3B | 19 |
| 1228 | AC | 94 | 02 | D0 | 05 | 20 | 13 | 11 |
| 1230 | B0 | DE | 4C | 93 | 10 | 20 | 39 | 14 |
| 1238 | 20 | 13 | 11 | 90 | 0D | A0 | 2C | 20 |
| 1240 | E7 | 10 | 20 | AB | 12 | 20 | AE | 18 |
| 1248 | D0 | EE | 20 | B3 | 15 | D0 | E3 | 20 |
| 1250 | 47 | 19 | 20 | 13 | 18 | 20 | 52 | 19 |
| 1258 | 20 | 0E | 1E | 48 | 20 | 0B | 13 | 68 |
| 1260 | 20 | 22 | 13 | A2 | 06 | E0 | 03 | D0 |
| 1268 | 14 | AC | 8B | 02 | F0 | 0F | AD | 96 |
| 1270 | 02 | C9 | E8 | B1 | FB | B0 | 1D | 20 |
| 1278 | A1 | 12 | 88 | D0 | F1 | 0E | 96 | 02 |

| 1280 | 90 | 0E | BD | E9 | 1E | 20 | AD | 15 |
| 1288 | BD | EF | 1E | F0 | 03 | 20 | AD | 15 |
| 1290 | CA | D0 | D2 | 60 | 20 | B7 | 12 | AA |
| 1298 | E8 | D0 | 01 | C8 | 98 | 20 | A1 | 12 |
| 12A0 | 8A | 8E | 88 | 02 | 20 | 1A | 18 | AE |
| 12A8 | 88 | 02 | 60 | AD | 8B | 02 | 20 | B6 |
| 12B0 | 12 | 85 | FB | 84 | FC | 60 | 38 | A4 |
Appendix F

13E8 20 03 19 A9 08 20 B9 13
13F0 A9 12 20 09 10 A0 08 A2
13F8 00 A1 FB 29 7F C9 20 B0

1400 02 A9 2E 20 09 10 C9 22
1408 F0 04 C9 62 D0 0A A9 14
1410 20 09 10 A9 22 20 09 10
1418 20 3B 19 88 D0 DB A9 92
1420 4C 09 10 20 01 18 A9 08
1428 20 AC 13 20 B3 15 20 D6
1430 13 A9 3A 8D 6F 02 4C 5C
1438 15 20 01 18 85 FD 86 FE
1440 20 06 10 C9 0D F0 03 20
1448 06 18 4C 55 19 20 4C 18
1450 85 FD 86 FE A2 00 8E A4
1458 02 20 A4 18 C9 20 F0 F4
1460 9D 8D 02 E8 E0 03 D0 F1
1468 CA 30 14 BD 8D 02 38 E9
1470 3F A0 05 4A 6E A4 02 6E
1478 A3 02 88 D0 F6 F0 E9 A2

1480 02 20 06 10 C9 0D F0 22
1488 C9 3A F0 1E C9 20 F0 F1
1490 20 A4 15 B0 0F 20 84 18
1498 A4 FB 84 FC 85 FB A9 30
14A0 9D A3 02 E8 9D A3 02 E8
14A8 D0 D7 8E 92 02 A2 00 8E
14B0 94 02 A2 00 8E 89 02 AD
14B8 94 02 20 C3 12 AE 96 02
14C0 8E 93 02 AA BD 36 1F 20
14C8 84 15 BD F6 1E 20 84 15
14D0 A2 06 E0 03 D0 14 AC 8B
14D8 02 F0 0F AD 96 02 C9 E8
14E0 A9 30 B0 1E 20 81 15 88
14E8 D0 F1 0E 96 02 90 0E BD
14F0 E9 1E 20 84 15 BD EF 1E
14F8 F0 03 20 84 15 CA D0 D2

1500 F0 06 20 81 15 20 81 15
1508 AD 92 02 CD 89 02 F0 03
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 6C 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
Appendix F

1770 20 06 10 C9 0D F0 0A 20
1778 06 10 C9 57 D0 03 EE 8C

1780 02 20 3C 18 AE 94 02 D0
1788 18 20 0A 11 90 13 AC 8C
1790 02 D0 1A B1 FB 20 C3 12
1798 AA BD F6 1E D0 06 20 E7
17A0 10 4C 93 10 AC 8B 02 C0
17A8 02 D0 33 F0 03 8C 8B 02
17B0 88 38 B1 FB AA ED 8D 02
17B8 C8 B1 FB ED 8E 02 90 1E
17C0 88 AD 8F 02 F1 FB C8 AD
17C8 90 02 F1 FB 90 10 88 18
17D0 8A 6D A0 02 91 FB C8 B1
17D8 FB 6D A1 02 91 FB 20 3B
17E0 19 88 10 FA 30 9E 20 4C
17E8 18 85 FD 86 FE 20 5D 18
17F0 8D 92 02 8E 93 02 20 A4
17F8 18 20 60 18 85 FB 86 FC

1800 60 20 4C 18 B0 F6 20 60
1808 18 B0 03 20 5D 18 85 FD
1810 86 FE 60 A5 FC 20 1A 18
1818 A5 FB 48 4A 4A 4A 4A 20
1820 32 18 AA 68 29 0F 20 32
1828 18 48 8A 20 09 10 68 4C
1830 09 10 18 69 F6 90 02 69
1838 06 69 3A 60 A2 02 B5 FA
1840 48 B5 FC 95 FA 68 95 FC
1848 CA D0 F3 60 A9 00 8D 97
1850 02 20 A4 18 C9 20 F0 F9
1858 20 84 18 B0 08 20 A4 18
1860 20 6F 18 90 07 AA 20 6F
1868 18 90 01 60 4C 8E 10 A9
1870 00 8D 97 02 20 A4 18 C9
1878 20 D0 09 20 A4 18 C9 20

1880 D0 0F 18 60 20 99 18 0A
1888 0A 0A 0A 8D 97 02 20 A4

284
Appendix F

1890 18 20 99 18 0D 97 02 38
1898 60 C9 3A 08 29 0F 28 90
18A0 02 69 08 60 20 06 10 C9
18A8 0D D0 F8 4C 93 10 A5 9B
18B0 C9 EF D0 07 08 20 CC FF
18B8 85 9E 28 60 20 C6 18 AD
18C0 13 E8 6A 90 F7 60 20 AE
18C8 18 D0 0B 20 D7 18 A9 03
18D0 85 B0 A9 00 85 AF 60 08
18D8 78 AD 40 E8 09 10 8D 40
18E0 E8 A9 7F 8D 4E E8 A9 3C
18E8 8D 11 E8 A9 3D 8D 13 E8
18F0 AD EE 1F 85 90 AD EF 1F
18F8 85 91 28 60 8D 7B 02 8E

1900 7A 02 60 8D 89 02 A0 00
1908 20 52 19 B1 FB 20 1A 18
1910 20 3B 19 CE 89 02 D0 F0
1918 60 20 6F 18 90 0B A2 00
1920 81 FB C1 FB F0 03 4C 8E
1928 10 20 3B 19 CE 89 02 60
1930 A9 7C 85 FB A9 02 85 FC
1938 A9 05 60 E6 FB D0 07 E6
1940 FC D0 03 EE 94 02 60 98
1948 48 20 55 19 68 A2 2E 20
1950 29 18 A9 20 2C A9 0D 4C
1958 09 10 A2 00 BD 76 1F 20
1960 09 10 E8 E0 1C D0 F5 A0
1968 3B 20 47 19 AD 7A 02 20
1970 1A 18 AD 7B 02 20 1A 18
1978 20 52 19 AD 81 02 20 1A

1980 18 AD 82 02 20 1A 18 20
1988 30 19 20 03 19 4C 93 10
1990 4C 8E 10 20 4C 18 20 FC
1998 18 20 5D 18 8D 82 02 8E
19A0 81 02 20 30 19 8D 89 02
19A8 20 A4 18 20 19 D0 F8
19B0 F0 DB 20 60 1C AE 80 02
19B8 9A 6C 94 00 4C 8E 10 A0
<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>19C0 01</td>
<td>84 84 D4 88 84 D1 84 96</td>
</tr>
<tr>
<td>19C8 01</td>
<td>84 9D A9 02 85 DB A9 A3</td>
</tr>
<tr>
<td>19D0 01</td>
<td>85 DA 20 06 10 C9 20 F0</td>
</tr>
<tr>
<td>19D8 01</td>
<td>F9 C9 0D F0 1A C9 22 D0</td>
</tr>
<tr>
<td>19E0 01</td>
<td>DB 20 06 10 C9 22 F0 36</td>
</tr>
<tr>
<td>19E8 01</td>
<td>C9 0D F0 0B 91 DA E6 D1</td>
</tr>
<tr>
<td>19F0 01</td>
<td>C8 C0 10 F0 C7 D0 EA AD</td>
</tr>
<tr>
<td>19F8 01</td>
<td>87 02 C9 4C D0 E1 AD 00</td>
</tr>
<tr>
<td>1A00 01</td>
<td>C0 C9 40 D0 06 20 22 F3</td>
</tr>
<tr>
<td>1A08 01</td>
<td>4C 12 1A C9 4C D0 AD 20</td>
</tr>
<tr>
<td>1A10 01</td>
<td>56 F3 20 BC 18 A5 96 29</td>
</tr>
<tr>
<td>1A18 01</td>
<td>10 D0 E1 4C 93 10 20 06</td>
</tr>
<tr>
<td>1A20 01</td>
<td>10 C9 0D FA D2 C9 2C D0</td>
</tr>
<tr>
<td>1A28 01</td>
<td>F0 20 6F 18 29 0F F0 C3</td>
</tr>
<tr>
<td>1A30 01</td>
<td>C9 03 F0 FA 85 D4 20 06</td>
</tr>
<tr>
<td>1A38 01</td>
<td>10 C9 0D F0 BA C9 2C D0</td>
</tr>
<tr>
<td>1A40 01</td>
<td>E6 20 F9 17 20 06 10 C9</td>
</tr>
<tr>
<td>1A48 01</td>
<td>2C D0 F4 20 60 18 85 C9</td>
</tr>
<tr>
<td>1A50 01</td>
<td>86 CA 20 06 10 C9 20 F0</td>
</tr>
<tr>
<td>1A58 01</td>
<td>F9 C9 0D D0 EC AD 87 02</td>
</tr>
<tr>
<td>1A60 01</td>
<td>C9 53 D0 F7 AD 00 C0 C9</td>
</tr>
<tr>
<td>1A68 01</td>
<td>40 D0 06 20 A4 F6 4C 93</td>
</tr>
<tr>
<td>1A70 01</td>
<td>10 C9 4C D0 D4 20 E3 F6</td>
</tr>
<tr>
<td>1A78 01</td>
<td>4C 93 10 20 01 18 20 3B</td>
</tr>
<tr>
<td>1A80 01</td>
<td>19 20 3B 19 20 0B 18 20</td>
</tr>
<tr>
<td>1A88 01</td>
<td>52 19 20 13 11 90 0A 98</td>
</tr>
<tr>
<td>1A90 01</td>
<td>D0 15 AD 91 02 30 10 10</td>
</tr>
<tr>
<td>1A98 01</td>
<td>08 C8 D0 0B AD 91 02 10</td>
</tr>
<tr>
<td>1AA0 01</td>
<td>06 20 1A 18 4C 93 10 4C</td>
</tr>
<tr>
<td>1AA8 01</td>
<td>8E 10 20 01 18 20 C0 1A</td>
</tr>
<tr>
<td>1AB0 01</td>
<td>4C 93 10 20 55 19 A2 2E</td>
</tr>
<tr>
<td>1AB8 01</td>
<td>A9 24 20 29 18 20 13 18</td>
</tr>
<tr>
<td>1AC0 01</td>
<td>20 2F 1B 20 E6 1A 20 52</td>
</tr>
<tr>
<td>1AC8 01</td>
<td>19 20 CC 1A 20 CF 1A 20</td>
</tr>
<tr>
<td>1AD0 01</td>
<td>52 19 A2 04 A9 30 18 0E</td>
</tr>
<tr>
<td>1AD8 01</td>
<td>92 02 2E 93 02 69 00 20</td>
</tr>
<tr>
<td>1AE0 01</td>
<td>09 10 CA D0 EF 60 A5 FC</td>
</tr>
<tr>
<td>1AE8 01</td>
<td>A6 FB 8D 93 02 8E 92 02</td>
</tr>
</tbody>
</table>
Appendix F

1AF0 20 52 19 A5 FC 20 FA 1A
1AF8 A5 FB AA 20 52 19 8A 29

1B00 7F C9 20 08 B0 0A A9 12
1B08 20 09 10 8A 18 69 40 AA
1B10 8A 20 09 10 C9 22 F0 04
1B18 C9 62 D0 0A A9 14 20 09
1B20 10 A9 22 20 09 10 28 B0
1B28 05 A9 92 20 09 10 60 20
1B30 52 19 A6 FB A5 FC AC 00
1B38 C0 C0 40 D0 03 4C D9 DC
1B40 C0 4C D0 03 4C 83 CF 4C
1B48 8E 10 20 5B 1B B0 F8 20
1B50 52 19 20 13 18 20 C3 1A
1B58 4C 93 10 A2 04 A9 00 85
1B60 FC 20 17 1C 20 83 1B 85
1B68 FB 20 78 1B 20 92 1B CA
1B70 D0 F7 08 20 52 19 28 60
1B78 20 06 10 C9 0D F0 0F C9

1B80 20 F0 0B C9 30 90 C0 C9
1B88 3A B0 BC 29 0F 60 68 68
1B90 18 60 85 FE A5 FC 48 A5
1B98 FB 48 06 FB 26 FC 06 FB
1BA0 26 FC 68 65 FB 85 FB 68
1BA8 65 FC 85 FC 06 FB 26 FC
1BB0 A5 FE 65 FB 85 FB A9 00
1BB8 65 FC 85 FC 60 20 17 1C
1BC0 8D 93 02 48 48 20 52 19
1BC8 20 52 19 68 20 1A 18 20
1BD0 52 19 68 AA A9 00 20 36
1BD8 1B 20 52 19 20 CC 1A 4C
1BE0 93 10 20 F4 1B 20 52 19
1BE8 20 13 18 20 2F 1B 20 E6
1BF0 1A 4C 93 10 A2 0F A9 00
1BF8 85 FB 85 FC 20 17 1C 20

1C00 83 1B 20 11 1C 20 78 1B
1C08 20 11 1C CA D0 F7 4C 52
Appendix F

1C10 19 4A 26 FB 26 FC 60 20
1C18 A4 18 C9 20 F0 F9 60 A9
1C20 02 4D 4C E8 8D 4C E8 4C
1C28 93 10 20 0B 18 4C F6 17
1C30 20 2A 1C 18 A5 FB 65 FD
1C38 85 FB A5 FC 65 FE 85 FC
1C40 4C 50 1C 20 2A 1C 20 13
1C48 11 84 FC AD 91 02 85 FB
1C50 20 52 19 20 13 18 4C 93
1C58 10 20 60 1C 00 6C EC 1F
1C60 78 AD E5 02 AE E6 02 85
1C68 92 86 93 AD 9E 02 AE 9F
1C70 02 85 90 86 91 58 60 20
1C78 2A 1C 20 3C 18 20 52 19

1C80 A0 00 8C 92 02 8C 93 02
1C88 20 13 11 90 1D AD 94 02
1C90 D0 18 A0 00 18 B1 FB 6D
1C98 92 02 8D 92 02 98 6D 93
1CA0 02 8D 93 02 20 3B 19 4C
1CA8 88 1C AD 93 02 20 1A 18
1CB0 AD 92 02 20 1A 18 4C 93
1CB8 10 AD A2 02 D0 04 A5 9E
1CC0 D0 06 68 A8 68 AA 68 40
1CC8 AD 6F 02 C9 11 D0 7D A5
1CD0 D8 C9 18 D0 ED A5 C4 85
1CD8 FD A5 C5 85 FE A9 19 8D
1CE0 9C 02 A0 01 20 8C 1E C9
1CE8 3A F0 1A C9 2C F0 16 C9
1CF0 24 F0 12 CE 9C 02 F0 CA
1CF8 38 A5 FD E9 28 85 FD B0

1D00 E1 C6 FE D0 DD 8D 87 02
1D08 20 45 1E B0 B5 AD 87 02
1D10 C9 3A D0 11 18 A5 FB 69
1D18 08 85 FB 90 02 E6 FC 20
1D20 D6 13 4C 39 1D C9 24 F0
1D28 1A 20 0E 1E 20 AB 12 A9
1D30 00 8D 8C 02 A0 2C 20 4F
1D38 12 A9 00 85 9E 4C 4A 12
<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D40</td>
<td>4C C2 1C 20 3B 19 20 B3</td>
</tr>
<tr>
<td>1D48</td>
<td>1A 4C 39 1D C9 91 D0 F0</td>
</tr>
<tr>
<td>1D50</td>
<td>A5 D8 D0 EC A5 C4 85 FD</td>
</tr>
<tr>
<td>1D58</td>
<td>A5 C5 85 FE A9 19 8D 9C</td>
</tr>
<tr>
<td>1D60</td>
<td>02 A0 01 20 8C 1E C9 3A</td>
</tr>
<tr>
<td>1D68</td>
<td>F0 1A C9 2C F0 16 C9 24</td>
</tr>
<tr>
<td>1D70</td>
<td>F0 12 CE 9C 02 F0 15 18</td>
</tr>
<tr>
<td>1D78</td>
<td>A5 FD 69 28 85 FD 90 E1</td>
</tr>
<tr>
<td>1D80</td>
<td>E6 FE D0 DD 8D 87 02 20</td>
</tr>
<tr>
<td>1D88</td>
<td>45 1E 90 03 4C C2 1C AD</td>
</tr>
<tr>
<td>1D90</td>
<td>87 02 C9 3A F0 06 C9 24</td>
</tr>
<tr>
<td>1D98</td>
<td>F0 1D D0 27 20 15 1E 38</td>
</tr>
<tr>
<td>1DA0</td>
<td>A5 FB E9 08 85 FB B0 02</td>
</tr>
<tr>
<td>1DA8</td>
<td>C6 FC 20 D9 13 A9 00 85</td>
</tr>
<tr>
<td>1DB0</td>
<td>9E 20 40 1E 4C 96 10 20</td>
</tr>
<tr>
<td>1DB8</td>
<td>15 1E 20 D5 10 20 B6 1A</td>
</tr>
<tr>
<td>1DC0</td>
<td>4C AD 1D 20 15 1E A5 FB</td>
</tr>
<tr>
<td>1DC8</td>
<td>A6 FC 85 FD 86 FE A9 10</td>
</tr>
<tr>
<td>1DD0</td>
<td>8D 9C 02 38 A5 FD ED 9C</td>
</tr>
<tr>
<td>1DD8</td>
<td>02 85 FB A5 FE E9 00 85</td>
</tr>
<tr>
<td>1DE0</td>
<td>FC 20 0E 1E 20 AB 12 20</td>
</tr>
<tr>
<td>1DE8</td>
<td>13 11 F0 07 B0 F3 CE 9C</td>
</tr>
<tr>
<td>1DF0</td>
<td>02 D0 E0 EE 8B 02 AD 8B</td>
</tr>
<tr>
<td>1DF8</td>
<td>02 20 B9 13 A2 00 A1 FB</td>
</tr>
<tr>
<td>1E00</td>
<td>8E 8C 02 A9 2C 20 4D 19</td>
</tr>
<tr>
<td>1E08</td>
<td>20 52 12 4C AD 1D A2 00</td>
</tr>
<tr>
<td>1E10</td>
<td>A1 FB 4C C3 12 A9 83 85</td>
</tr>
<tr>
<td>1E18</td>
<td>C8 85 FE A9 00 85 C7 A9</td>
</tr>
<tr>
<td>1E20</td>
<td>28 85 FD A0 C0 A2 04 88</td>
</tr>
<tr>
<td>1E28</td>
<td>B1 C7 91 FD 98 D0 F8 C6</td>
</tr>
<tr>
<td>1E30</td>
<td>C8 C6 FE CA D0 F1 A2 27</td>
</tr>
<tr>
<td>1E38</td>
<td>A9 20 9D 00 80 CA 10 FA</td>
</tr>
<tr>
<td>1E40</td>
<td>A9 13 4C 09 10 C0 28 D0</td>
</tr>
<tr>
<td>1E48</td>
<td>02 38 60 20 8C 1E C9 20</td>
</tr>
<tr>
<td>1E50</td>
<td>F0 F3 88 20 75 1E AA 20</td>
</tr>
<tr>
<td>1E58</td>
<td>75 1E 85 FB 86 FC A9 FF</td>
</tr>
<tr>
<td>1E60</td>
<td>8D A2 02 85 A7 A5 AA F0</td>
</tr>
<tr>
<td>1E68</td>
<td>0A A5 A9 A4 C6 91 C4 A9</td>
</tr>
</tbody>
</table>
Appendix F

1E70 00 85 AA 18 60 20 8C 1E
1E78 20 99 18 0A 0A 0A 0A 8D

1E80 97 02 20 8C 1E 20 99 18
1E88 0D 97 02 60 B1 FD C8 29
1E90 7F C9 20 B0 02 09 40 60
1E98 40 02 45 03 D0 08 40 09
1EA0 30 22 45 33 D0 08 40 09
1EA8 40 02 45 33 D0 08 40 09
1EB0 40 02 45 B3 D0 08 40 09
1EB8 00 22 44 33 D0 8C 44 00
1EC0 11 22 44 33 D0 8C 44 9A
1EC8 10 22 44 33 D0 08 40 09
1ED0 10 22 44 33 D0 08 40 09
1ED8 62 13 78 A9 00 21 81 82
1EE0 00 00 59 4D 91 92 86 4A
1EE8 85 9D 2C 29 2C 23 28 24
1EF0 59 00 58 24 24 00 1C 8A
1EF8 1C 23 5D 8B 1B A1 9D 8A

1F00 1D 23 9D 8B 1D A1 00 29
1F08 19 AE 69 A8 19 23 24 53
1F10 1B 23 24 53 19 A1 00 1A
1F18 5B 5B A5 69 24 24 AE AE
1F20 A8 AD 29 00 7C 00 15 9C
1F28 6D 9C A5 69 29 53 84 13
1F30 34 11 A5 69 23 A0 D8 62
1F38 5A 48 26 62 94 88 54 44
1F40 C8 54 68 44 E8 94 00 B4
1F48 08 84 74 B4 28 6E 74 F4
1F50 CC 4A 72 F2 A4 8A 00 AA
1F58 A2 A2 74 74 74 22 44 68
1F60 B2 32 B2 00 22 00 1A 1A
1F68 26 26 72 72 88 C8 C4 CA
1F70 26 48 44 44 A2 C8 0D 20
1F78 20 20 20 50 43 20 20 49
Program 3. Checksum For Micromon Plus.

10 DATA 15965,14778,13059,14282,14416,17693,12979,12903,17667,6,21760
20 DATA 14416,17693,12979,12903
100 Q=23296
110 FOR BLOCK=1TO8
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 #"BLOCK:GOTO170
165 PRINT" BLOCK"
170 CK=0:Q=Q+128
180 NEXT BLOCK
190 PRINT"ANY REMAINING PROBLEMS ARE EITHER WITHIN THE FINAL"
200 PRINT"SHORT BLOCK OR WITHIN DATA STATEMENTS IN THIS PROGRAM."
Appendix F


5B00 78 A5 90 A6 91 CD EE 6F
5B08 D0 05 EC EF 6F F0 30 8D
5B10 9E 02 8E 9F 02 AD EE 6F
5B18 AE EF 6F 85 90 86 91 A5
5B20 92 A6 93 8D E5 02 8E E6
5B28 02 AD 3C 5F AE 3D 5F 8D
5B30 E3 02 8E E4 02 AD F0 6F
5B38 AE F1 6F 85 92 86 93 AD
5B40 3E 5F AE 3F 5F E0 80 B0
5B48 08 85 34 86 35 85 30 86
5B50 31 A9 10 8D 84 02 8D 85
5B58 02 A9 00 8D 86 02 8D A2
5B60 02 8D E7 02 8D E8 02 58
5B68 00 A2 0C DD 15 5F D0 13
5B70 8D 87 02 8A 0A AA BD 22
5B78 5F 85 FB BD 23 5F 85 FC

5B80 6C FB 00 CA 10 E5 4C 8E
5B88 60 20 39 64 20 13 61 90
5B90 17 20 EF 60 8E 8C 02 20
5B98 52 62 20 AB 5B 20 AB 62
5BA0 20 93 5C 20 AE 68 D0 E4
5BA8 4C 9B 60 A2 1E 20 F1 60
5BB0 A0 00 B1 FB 20 60 5C CC
5BB8 8B 02 C8 90 F5 60 A5 B0
5BC0 C9 03 D0 19 20 06 60 AA
5BC8 A9 04 E0 0D F0 09 20 6F
5BD0 68 29 1F C9 04 90 AF 20
5BD8 E3 5B 4C 9B 60 20 CC FF
5BE0 4C 93 60 85 B0 85 D4 20
5BE8 09 5C AE 00 C0 E0 40 D0
5BF0 0B 20 BA F0 20 2D F1 A5
5BF8 96 D0 E2 60 E0 4C D0 5D

5C00 20 D5 F0 20 48 F1 4C F7
5C08 5B A9 00 85 96 8D FC 03
5C10 85 0D 8D E8 02 60 20 39
5C18 64 AE 94 02 D0 10 20 13
5C20 61 90 0B 20 31 5C 20 93
5C28 5C 20 AE 68 D0 EB 4C A8
5C30 5B A2 05 20 F1 60 20 13
5C38 68 A2 02 20 F1 60 A9 10
5C40 20 03 69 A9 10 20 B9 63
5C48 A2 04 20 F1 60 A0 10 A2
5C50 00 A1 FB 20 60 5C 20 3B
5C58 69 88 D0 F5 60 4C 8E 60
5C60 29 7F C9 20 BO 02 A9 20
5C68 4C 09 60 20 06 60 C9 0D
5C70 F0 19 C9 20 D0 03 20 17
5C78 6C C9 58 F0 50 20 71 5D

5C80 8E E8 02 A2 02 20 A7 5C
5C88 4C 9B 60 A2 04 20 Cl 5C
5C90 4C 9B 60 20 55 69 AE E7
5C98 02 F0 31 CE E7 02 D0 2C
5CA0 AE E8 02 F0 1A A2 06 20
5CA8 5C A2 14 20 F1 60 BD
5CB0 A3 02 20 09 60 E8 EC E8
5CB8 02 D0 F4 A2 03 D0 02 A2
5CC0 09 20 55 69 CA D0 FA A9
5CC8 39 8D E7 02 60 A9 00 8D
5CD0 E7 02 8D E8 02 4C 9B 60
5CD8 20 09 5C 20 CC FF 20 06
5CE0 60 C9 0D F0 16 C9 24 F0
5CE8 24 48 20 9E 5D 68 20 09
5CF0 60 20 06 60 C9 0D D0 F6
5CF8 4C DD 5B 20 52 69 20 C5

5D00 5D 20 06 60 C9 0D F0 F0
5D08 20 09 60 D0 F4 A2 00 20
5D10 82 5D 20 8B 5D 20 55 69
5D18 20 55 69 A0 03 D0 02 A0
5D20 02 84 D1 A9 08 85 AF 20
5D28 06 60 AA A4 96 D0 36 20
5D30 06 60 A4 96 D0 2F C6 D1
5D38 D0 ED 20 36 6B 20 52 69
5D40 20 06 60 F0 05 20 09 60
<table>
<thead>
<tr>
<th>Code</th>
<th>Hex</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>5D48</td>
<td>D0</td>
<td>F6 20 55 69 A9 00 85</td>
</tr>
<tr>
<td>5D50</td>
<td>AF</td>
<td>20 E4 FF F0 C9 D0 05</td>
</tr>
<tr>
<td>5D58</td>
<td>20</td>
<td>E4 FF F0 FB C9 20 F0</td>
</tr>
<tr>
<td>5D60</td>
<td>F7</td>
<td>C9 03 D0 BA 20 12 5E</td>
</tr>
<tr>
<td>5D68</td>
<td>20</td>
<td>55 69 4C 93 60 20 17</td>
</tr>
<tr>
<td>5D70</td>
<td>6C</td>
<td>C9 22 D0 7B A2 00 20</td>
</tr>
<tr>
<td>5D78</td>
<td>06</td>
<td>60 C9 0D F0 0C C9 22</td>
</tr>
<tr>
<td>5D80</td>
<td>F0</td>
<td>08 9D A3 02 E8 E0 40</td>
</tr>
<tr>
<td>5D88</td>
<td>90</td>
<td>ED 60 86 D1 A9 A3 85</td>
</tr>
<tr>
<td>5D90</td>
<td>DA</td>
<td>A9 02 85 DB 20 CC FF</td>
</tr>
<tr>
<td>5D98</td>
<td>20</td>
<td>F3 5D 4C C9 5D A9 08</td>
</tr>
<tr>
<td>5DA0</td>
<td>85</td>
<td>D4 85 B0 AC 00 C0 C0</td>
</tr>
<tr>
<td>5DA8</td>
<td>40</td>
<td>D0 0B 20 BA F0 A9 6F</td>
</tr>
<tr>
<td>5DB0</td>
<td>20</td>
<td>28 F1 4C F7 5B C0 4C</td>
</tr>
<tr>
<td>5DB8</td>
<td>D0</td>
<td>36 20 D5 F0 A9 6F 20</td>
</tr>
<tr>
<td>5DC0</td>
<td>43</td>
<td>F1 4C F7 5B A9 6F 85</td>
</tr>
<tr>
<td>5DC8</td>
<td>D3</td>
<td>A9 08 85 D4 85 AF AC</td>
</tr>
<tr>
<td>5DD0</td>
<td>00</td>
<td>C0 C0 40 D0 0B 20 B6</td>
</tr>
<tr>
<td>5DD8</td>
<td>F0</td>
<td>A5 D3 20 64 F1 4C F7</td>
</tr>
<tr>
<td>5DE0</td>
<td>5B</td>
<td>C0 4C D0 0B 20 D2 F0</td>
</tr>
<tr>
<td>5DE8</td>
<td>A5</td>
<td>D3 20 93 F1 4C F7 5B</td>
</tr>
<tr>
<td>5DF0</td>
<td>4C</td>
<td>8E 60 A9 08 85 D4 A9</td>
</tr>
<tr>
<td>5DF8</td>
<td>60</td>
<td>85 D3 AD 00 C0 C9 40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Hex</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>5E00</td>
<td>D0</td>
<td>06 20 66 F4 4C F7 5B</td>
</tr>
<tr>
<td>5E08</td>
<td>C9</td>
<td>4C D0 E4 20 A5 F4 4C</td>
</tr>
<tr>
<td>5E10</td>
<td>F7</td>
<td>5B A9 00 85 AF AD 00</td>
</tr>
<tr>
<td>5E18</td>
<td>C0</td>
<td>C9 40 D0 03 4C 8F F3</td>
</tr>
<tr>
<td>5E20</td>
<td>C9</td>
<td>4C D0 CC 4C CE F3 A9</td>
</tr>
<tr>
<td>5E28</td>
<td>02</td>
<td>2C 4C E8 08 A9 0E 28</td>
</tr>
<tr>
<td>5E30</td>
<td>F0</td>
<td>02 09 80 20 09 60 4C</td>
</tr>
<tr>
<td>5E38</td>
<td>93</td>
<td>60 20 09 5C 20 6E 5D</td>
</tr>
<tr>
<td>5E40</td>
<td>20</td>
<td>8B 5D 20 06 60 8D FB</td>
</tr>
<tr>
<td>5E48</td>
<td>00</td>
<td>20 06 60 8D FC 00 20</td>
</tr>
<tr>
<td>5E50</td>
<td>12</td>
<td>5E 20 52 69 A9 24 A2</td>
</tr>
<tr>
<td>5E58</td>
<td>20</td>
<td>20 29 68 20 13 68 4C</td>
</tr>
<tr>
<td>5E60</td>
<td>93</td>
<td>60 20 60 6C 00 6C 3E</td>
</tr>
<tr>
<td>5E68</td>
<td>5F</td>
<td>A0 08 84 D4 A0 4C 8C</td>
</tr>
</tbody>
</table>
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 relocater. 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 kernel 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

If the output is to the screen, then switch the output 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.

Regardless of the output, clear the RS-232 channel and set output to the screen.

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Use</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7,6,5</td>
<td>Parity Options</td>
<td>-0</td>
<td>Parity disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1</td>
<td>Odd parity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 1</td>
<td>Even parity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 1</td>
<td>Mark transmitted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 1 1</td>
<td>Space transmitted</td>
</tr>
<tr>
<td>4</td>
<td>Duplex</td>
<td>0</td>
<td>Full duplex</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Half duplex</td>
</tr>
<tr>
<td>3,2,1</td>
<td>Unused</td>
<td>0</td>
<td>3 line</td>
</tr>
<tr>
<td>0</td>
<td>Handshake</td>
<td>1</td>
<td>x line</td>
</tr>
</tbody>
</table>
Appendix F

Control Register Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Use</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stop Bits</td>
<td>0</td>
<td>1 stop bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2 stop bits</td>
</tr>
<tr>
<td>6,5</td>
<td>Word Length</td>
<td>0 0</td>
<td>8 bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1</td>
<td>7 bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0</td>
<td>6 bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 1</td>
<td>5 bits</td>
</tr>
<tr>
<td>4</td>
<td>Unused</td>
<td>0 0 0</td>
<td>User rate</td>
</tr>
<tr>
<td>3,2,1,0</td>
<td>Baud Rate</td>
<td>0 0 0 0</td>
<td>50 Baud</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 1</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 0</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 1</td>
<td>134.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 0 0</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 0 1</td>
<td>300</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 1 0</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 0</td>
<td>1200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 1</td>
<td>1800</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 1 0</td>
<td>2400</td>
</tr>
</tbody>
</table>

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**

```
.7r 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:

```
.7r 2FFF 2FFF 0 7
```

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**

```
.f 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:

```
.f 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

<table>
<thead>
<tr>
<th>VIC Micromon Instruction</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMPLE ASSEMBLER</td>
<td>A</td>
</tr>
<tr>
<td>BREAK SET</td>
<td>B</td>
</tr>
<tr>
<td>COMPARE MEMORY</td>
<td>C</td>
</tr>
<tr>
<td>DISASSEMBLER</td>
<td>D</td>
</tr>
<tr>
<td>EXIT VIC MICROMON</td>
<td>E</td>
</tr>
<tr>
<td>FILL MEMORY</td>
<td>F</td>
</tr>
<tr>
<td>GO RUN</td>
<td>G</td>
</tr>
<tr>
<td>HUNT MEMORY</td>
<td>H</td>
</tr>
<tr>
<td>INITIAL MEMORY &amp; SCREEN PTRS</td>
<td>I</td>
</tr>
<tr>
<td>JUMP TO SUBROUTINE</td>
<td>J</td>
</tr>
<tr>
<td>LOAD MEMORY FROM DEVICE</td>
<td>L</td>
</tr>
<tr>
<td>MEMORY DISPLAY</td>
<td>M</td>
</tr>
<tr>
<td>NEW LOCATER</td>
<td>N</td>
</tr>
<tr>
<td>OFFSET OR BRANCH CALCULATE</td>
<td>O</td>
</tr>
<tr>
<td>PRINT SWITCHER</td>
<td>P</td>
</tr>
<tr>
<td>QUICK TRACE</td>
<td>Q</td>
</tr>
<tr>
<td>REGISTER DISPLAY</td>
<td>R</td>
</tr>
<tr>
<td>SAVE MEMORY TO DEVICE</td>
<td>S</td>
</tr>
<tr>
<td>TRANSFER MEMORY</td>
<td>T</td>
</tr>
<tr>
<td>VERIFY MEMORY FROM DEVICE</td>
<td>V</td>
</tr>
<tr>
<td>WALK CODE</td>
<td>W</td>
</tr>
<tr>
<td>EXIT TO BASIC</td>
<td>X</td>
</tr>
<tr>
<td>ASCII CONVERSION</td>
<td>#</td>
</tr>
<tr>
<td>DECIMAL CONVERSION</td>
<td>$</td>
</tr>
<tr>
<td>HEXADECIMAL CONVERSION</td>
<td>%</td>
</tr>
<tr>
<td>BINARY CONVERSION</td>
<td>&amp;</td>
</tr>
<tr>
<td>CHECKSUM MEMORY</td>
<td>(</td>
</tr>
<tr>
<td>COMMAND END TONE ENABLE</td>
<td>)</td>
</tr>
<tr>
<td>COMMAND END TONE DISABLE</td>
<td>+</td>
</tr>
<tr>
<td>ADDITION</td>
<td>-</td>
</tr>
<tr>
<td>LOAD MEMORY FROM EPROM</td>
<td>£</td>
</tr>
<tr>
<td>PROGRAM EPROM FROM MEMORY</td>
<td>π</td>
</tr>
<tr>
<td>COMPARE EPROM TO MEMORY</td>
<td>=</td>
</tr>
</tbody>
</table>
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.
Appendix F

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
```

<table>
<thead>
<tr>
<th>Location</th>
<th>Old Value</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6018</td>
<td>45</td>
<td>65</td>
</tr>
<tr>
<td>602A</td>
<td>43</td>
<td>63</td>
</tr>
<tr>
<td>6392</td>
<td>4C</td>
<td>6C</td>
</tr>
<tr>
<td>6650</td>
<td>45</td>
<td>65</td>
</tr>
<tr>
<td>66E7</td>
<td>45</td>
<td>65</td>
</tr>
<tr>
<td>6897</td>
<td>43</td>
<td>63</td>
</tr>
</tbody>
</table>

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 kernel initialization has not occurred, any cartridge program using kernel I/O routines must do kernel initialization before using those routines.

VIC Micromon as presented here has the kernel 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.

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>A000</td>
<td>09</td>
</tr>
<tr>
<td>A001</td>
<td>A0</td>
</tr>
<tr>
<td>A002</td>
<td>C7</td>
</tr>
<tr>
<td>A003</td>
<td>FE</td>
</tr>
</tbody>
</table>
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 $BFOO. 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 I. Micromon Checksum.

1 IF PEEK(20478)=67 AND PEEK(20479)=73 THEN RUN 10
2 PRINT"VIC20 MICROMON LOAD &":PRINT"VERIFICATION PROGRAM.";PRINT
Appendix F

3 PRINT:PRINT:PRINT"AT LEAST 4K BYTES OF":PRINT"RAM MUST BE INSTALLED"
4 PRINT"AT 16384 ($4000) ELSE":PRINT"LOAD WILL FAIL.":PRINT
5 PRINT"IF LOADED & VERIFIED":PRINT"OK, MICROMON WILL BE":PRINT"ENTERED AUTOMATICALLY."
6 LOAD"",1,1
10 DATA 13328,16867,15061,13732,14507,13829,13267,12747,16288,13920
20 DATA 14355,11977,11877,13583,11338,15173,1237,14852,14051,15713
30 DATA 13442,15242,14746,15059,13134,15848,15858,17856,13327,8601
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 #"BLOCK:"ERR=1:GOTO170
165 PRINT"BLOCK #"BLOCK|" OK"
170 CK=0:Q=Q+128
180 NEXT BLOCK
190 IF ERR=1 THEN PRINT"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
Appendix F

4058 45 A2 42 A9 2A 4C 3D 49
4060 A9 3F 20 D2 FF A9 00 2C
4068 A9 0F 8D 0E 90 20 AE 45
4070 A9 2E 20 D2 FF A9 00 8D
4078 4E 03 8D 56 03 8D 64 03

4080 A2 7F 9A 20 8C 48 C9 2E
4088 F0 F9 C9 20 F0 F5 A2 24
4090 DD 90 4F D0 13 8D 49 03
4098 8A 0A AA BD B5 4F 85 FB
40A0 BD B6 4F 85 FC 6C FB 00
40A8 CA 10 E5 4C 60 40 A2 02
40B0 D0 02 A2 00 B4 FB D0 09
40B8 B4 FC D0 03 EE 56 03 D6
40C0 FC D6 FB 60 A9 00 8D 4E
40C8 03 20 13 42 A2 09 20 38
40D0 49 CA D0 FA 60 A2 02 B5
40D8 FA 48 BD 53 03 95 FA 68
40E0 9D 53 03 CA D0 F1 60 AD
40E8 54 03 AC 55 03 4C F4 40
40F0 A5 FD A4 FE 38 E5 FB 8D
40F8 53 03 98 E5 FC A8 0D 53

4100 03 60 A9 00 F0 02 A9 01
4108 8D 57 03 20 CB 47 20 AE
4110 45 20 F0 40 20 21 48 90
4118 18 20 E7 40 90 7F 20 59
4120 41 E6 FD D0 02 E6 FE 20
4128 1F 49 AC 56 03 D0 6E F0
4130 E8 20 E7 40 18 AD 53 03
4138 65 FD 85 FD 98 65 FE 85
4140 FE 20 D5 40 20 59 41 20
4148 E7 40 B0 51 20 AE 40 20
4150 B2 40 AC 56 03 D0 46 F0
4158 EB A2 00 A1 FB AC 57 03
4160 F0 02 81 FD C1 FD F0 0B
4168 20 F8 47 20 38 49 20 E1
4170 FF F0 2A 60 20 E6 47 20
4178 A1 49 F0 1E AE 56 03 D0
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4180</td>
<td>1C</td>
<td>20</td>
<td>F0</td>
<td>40</td>
<td>90</td>
<td>17</td>
<td>60</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4188</td>
<td>54</td>
<td>48</td>
<td>8D</td>
<td>4B</td>
<td>03</td>
<td>20</td>
<td>7C</td>
<td>41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4190</td>
<td>AD</td>
<td>4B</td>
<td>03</td>
<td>81</td>
<td>FB</td>
<td>20</td>
<td>1F</td>
<td>49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4198</td>
<td>D0</td>
<td>F3</td>
<td>4C</td>
<td>60</td>
<td>40</td>
<td>4C</td>
<td>68</td>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41A0</td>
<td>20</td>
<td>74</td>
<td>41</td>
<td>20</td>
<td>8C</td>
<td>48</td>
<td>C9</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41A8</td>
<td>D0</td>
<td>12</td>
<td>20</td>
<td>8C</td>
<td>48</td>
<td>9D</td>
<td>65</td>
<td>03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41B0</td>
<td>E8</td>
<td>20</td>
<td>A4</td>
<td>49</td>
<td>F0</td>
<td>20</td>
<td>E0</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41B8</td>
<td>D0</td>
<td>F3</td>
<td>F0</td>
<td>1A</td>
<td>8E</td>
<td>59</td>
<td>03</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41C0</td>
<td>5F</td>
<td>48</td>
<td>90</td>
<td>D6</td>
<td>9D</td>
<td>65</td>
<td>03</td>
<td>E8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41C8</td>
<td>20</td>
<td>A4</td>
<td>49</td>
<td>F0</td>
<td>09</td>
<td>20</td>
<td>57</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41D0</td>
<td>90</td>
<td>C8</td>
<td>E0</td>
<td>20</td>
<td>D0</td>
<td>EE</td>
<td>8E</td>
<td>4A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41D8</td>
<td>03</td>
<td>20</td>
<td>AE</td>
<td>45</td>
<td>A2</td>
<td>00</td>
<td>A0</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41E0</td>
<td>B1</td>
<td>FB</td>
<td>DD</td>
<td>65</td>
<td>03</td>
<td>D0</td>
<td>0A</td>
<td>C8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41E8</td>
<td>E8</td>
<td>EC</td>
<td>4A</td>
<td>03</td>
<td>D0</td>
<td>F2</td>
<td>20</td>
<td>68</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41F0</td>
<td>41</td>
<td>20</td>
<td>1F</td>
<td>49</td>
<td>20</td>
<td>7C</td>
<td>41</td>
<td>B0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41F8</td>
<td>E3</td>
<td>20</td>
<td>2B</td>
<td>44</td>
<td>20</td>
<td>F0</td>
<td>40</td>
<td>90</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4200</td>
<td>0D</td>
<td>A0</td>
<td>2C</td>
<td>20</td>
<td>C4</td>
<td>40</td>
<td>20</td>
<td>6F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4208</td>
<td>42</td>
<td>20</td>
<td>E1</td>
<td>FF</td>
<td>D0</td>
<td>EE</td>
<td>20</td>
<td>B6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4210</td>
<td>45</td>
<td>D0</td>
<td>8A</td>
<td>20</td>
<td>2D</td>
<td>49</td>
<td>20</td>
<td>F8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4218</td>
<td>47</td>
<td>20</td>
<td>38</td>
<td>49</td>
<td>20</td>
<td>C9</td>
<td>4D</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4220</td>
<td>20</td>
<td>CF</td>
<td>42</td>
<td>68</td>
<td>20</td>
<td>E6</td>
<td>42</td>
<td>A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4228</td>
<td>06</td>
<td>E0</td>
<td>03</td>
<td>D0</td>
<td>14</td>
<td>AC</td>
<td>4D</td>
<td>03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4230</td>
<td>F0</td>
<td>0F</td>
<td>AD</td>
<td>58</td>
<td>03</td>
<td>C9</td>
<td>E8</td>
<td>B1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4238</td>
<td>FB</td>
<td>B0</td>
<td>1D</td>
<td>20</td>
<td>65</td>
<td>42</td>
<td>88</td>
<td>D0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4240</td>
<td>F1</td>
<td>0E</td>
<td>58</td>
<td>03</td>
<td>90</td>
<td>0E</td>
<td>BD</td>
<td>E9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4248</td>
<td>4E</td>
<td>20</td>
<td>99</td>
<td>45</td>
<td>BD</td>
<td>EF</td>
<td>4E</td>
<td>F0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4250</td>
<td>03</td>
<td>20</td>
<td>99</td>
<td>45</td>
<td>CA</td>
<td>D0</td>
<td>D2</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4258</td>
<td>20</td>
<td>7B</td>
<td>42</td>
<td>AA</td>
<td>E8</td>
<td>D0</td>
<td>01</td>
<td>C8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4260</td>
<td>98</td>
<td>20</td>
<td>65</td>
<td>42</td>
<td>8A</td>
<td>8E</td>
<td>4A</td>
<td>03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4268</td>
<td>20</td>
<td>FF</td>
<td>47</td>
<td>AE</td>
<td>4A</td>
<td>03</td>
<td>60</td>
<td>AD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4270</td>
<td>4D</td>
<td>03</td>
<td>20</td>
<td>7A</td>
<td>42</td>
<td>85</td>
<td>FB</td>
<td>84</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4278</td>
<td>FC</td>
<td>60</td>
<td>38</td>
<td>A4</td>
<td>FC</td>
<td>AA</td>
<td>10</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4280</td>
<td>88</td>
<td>65</td>
<td>FB</td>
<td>90</td>
<td>01</td>
<td>C8</td>
<td>60</td>
<td>A8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4288</td>
<td>4A</td>
<td>90</td>
<td>0B</td>
<td>4A</td>
<td>B0</td>
<td>17</td>
<td>C9</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4290</td>
<td>F0</td>
<td>13</td>
<td>29</td>
<td>07</td>
<td>09</td>
<td>80</td>
<td>4A</td>
<td>AA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4298</td>
<td>BD</td>
<td>98</td>
<td>4E</td>
<td>B0</td>
<td>04</td>
<td>4A</td>
<td>4A</td>
<td>4A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42A0</td>
<td>4A</td>
<td>29</td>
<td>0F</td>
<td>D0</td>
<td>04</td>
<td>A0</td>
<td>80</td>
<td>A9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42A8</td>
<td>00</td>
<td>AA</td>
<td>BD</td>
<td>DC</td>
<td>4E</td>
<td>8D</td>
<td>58</td>
<td>03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42B0</td>
<td>29</td>
<td>03</td>
<td>8D</td>
<td>4D</td>
<td>03</td>
<td>98</td>
<td>29</td>
<td>8F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix F

42B8 AA 98 AØ Ø3 EØ 8A FØ ØB
42C0 4A 9Ø Ø8 4A 4A Ø9 20 88
42C8 DØ FA C8 88 DØ F2 6Ø B1
42D0 FB 20 65 42 A2 Ø1 20 CE
42D8 40 CC 4D Ø3 C8 9Ø FØ A2
42E0 Ø3 CØ Ø3 9Ø F1 6Ø A8 B9
42E8 F6 4E 8D 54 Ø3 B9 36 4F
42F0 8D 55 Ø3 A9 ØØ AØ Ø5 ØE
42F8 55 Ø3 2E 54 Ø3 2A 88 DØ

4300 F6 69 3F 20 D2 FF CA DØ
4308 E8 4C 38 49 20 E6 47 A9
4310 Ø3 20 9E 43 AØ 2C 4C 3C
4318 45 0Ø 0Ø 0Ø A9 3C 8D 13
4320 91 20 3A 43 A9 FF 8D 12
4328 91 A5 FB AØ 18 20 34 43
4330 A5 FF AØ 14 8D 10 91 8C
4338 11 91 AØ 1C 8C 11 91 6Ø
4340 20 54 48 85 FF 20 AE 45
4348 20 6E 41 20 7C 41 20 1C
4350 43 AD 49 Ø3 ØA Ø8 9Ø 17
4358 A1 FB 8D 10 91 78 A9 C4
4360 8D 19 91 A9 3C 8D 11- 91
4368 A9 20 2C 1D 91 FØ FB 20
4370 3A 43 58 8E 12 91 A9 ØC
4378 8D 11 91 AD 10 91 28 BØ

4380 Ø4 1Ø Ø2 81 FB C1 FB FØ
4388 Ø3 2Ø 68 41 2Ø 1F 49 DØ
4390 B7 A9 4C 48 A9 77 48 Ø8
4398 48 48 48 6C 6Ø Ø3 8D 4B
43A0 Ø3 48 20 8C 48 20 ØØ 49
43A8 DØ F8 68 49 FF 4C 72 42
43B0 20 2B 44 AE 56 Ø3 DØ ØD
43B8 20 FØ 4Ø 9Ø Ø8 2Ø C8 43
43C0 20 E1 FF DØ EE 4C ØE 42
43C8 20 AE 45 A2 2E A9 3A 2Ø
43D0 ØE 48 2Ø 38 49 2Ø F8 47
43D8 A9 Ø8 2Ø EA 48 A9 Ø8 2Ø
43E0 AB 43 2Ø 38 49 2Ø 38 49
43E8 A9 12 2Ø D2 FF AØ Ø8 A2
Appendix F

43FØ ØØ A1 FB 29 7F C9 2Ø BØ
43F8 Ø2 A9 2E 2Ø D2 FF A9 ØØ

440Ø 85 D4 EA EA EA EA EA EA
4408 EA EA EA EA EA 2Ø 1F 49 88
4410 DØ DF 4C DF 4A 2Ø E6 47
4418 A9 Ø8 2Ø 9E 43 2Ø B6 45
4420 2Ø C8 43 A9 3A 8D 77 Ø2
4428 4C 48 45 2Ø E6 47 85 FD
4430 86 FE 2Ø A4 49 FØ Ø3 2Ø
4438 EB 47 4C AE 45 2Ø 31 48
4440 85 FD 86 FE A2 ØØ 8E 66
4448 Ø3 2Ø 8C 48 C9 2Ø FØ F4
4450 9D 4F Ø3 E8 EØ Ø3 DØ F1
4458 CA 3Ø 14 BD 4F Ø3 38 E9
4460 3F AØ Ø5 4A 6E 66 Ø3 6E
4468 65 Ø3 88 DØ F6 FØ E9 A2
4470 Ø2 2Ø A4 49 FØ 22 C9 3A
4478 FØ 1E C9 2Ø FØ F3 2Ø 9Ø

4480 45 BØ ØF 2Ø 6C 48 A4 FB
4488 84 FC 85 FB A9 3Ø 9D 65
4490 Ø3 E8 9D 65 Ø3 E8 DØ D9
4498 8E 54 Ø3 A2 ØØ 8E 56 Ø3
44AØ A2 ØØ 8E 4B Ø3 AD 56 Ø3
44A8 2Ø 87 42 AE 58 Ø3 8E 55
44BØ Ø3 AA BD 36 4F 2Ø 7Ø 45
44B8 BD F6 4E 2Ø 7Ø 45 A2 Ø6
44CØ EØ Ø3 DØ 14 AC 4D Ø3 FØ
44C8 ØF AD 58 Ø3 C9 E8 A9 3Ø
44DØ BØ 1E 2Ø 6D 45 88 DØ F1
44DB ØE 58 Ø3 9Ø ØE BD E9 4E
44EØ 2Ø 7Ø 45 BD EF 4E FØ Ø3
44E8 2Ø 7Ø 45 CA DØ D2 FØ Ø6
44FØ 2Ø 6D 45 2Ø 6D 45 AD 54
44F8 Ø3 CD 4B Ø3 DØ 7F 2Ø 21

450Ø 48 AC 4D Ø3 FØ 2F AD 55
4508 Ø3 C9 9D DØ 2Ø 2Ø FØ 4Ø
451Ø 9Ø Ø1 88 C8 DØ 6F 98 2A
<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>4648</code></td>
<td>Ø3 90 D2</td>
<td>AE 42 Ø3 9A A9</td>
</tr>
<tr>
<td><code>4650</code></td>
<td>45 48 A9</td>
<td>BA 48 4C Ø6 47</td>
</tr>
<tr>
<td><code>4658</code></td>
<td>20 AE 45</td>
<td>20 14 49 8D 4B</td>
</tr>
<tr>
<td><code>4660</code></td>
<td>Ø3 AØ</td>
<td>Ø0 20 F2 48 AD 3D</td>
</tr>
<tr>
<td><code>4668</code></td>
<td>Ø3 AE 3C</td>
<td>Ø3 85 FB 86 FC</td>
</tr>
<tr>
<td><code>4670</code></td>
<td>20 38 49 A9</td>
<td>24 8D 4E Ø3</td>
</tr>
<tr>
<td><code>4678</code></td>
<td>20 16 42</td>
<td>20 E4 FF FØ FB</td>
</tr>
<tr>
<td><code>4680</code></td>
<td>C9 Ø3 DØ</td>
<td>Ø3 4C 68 4Ø C9</td>
</tr>
<tr>
<td><code>4688</code></td>
<td>4A DØ</td>
<td>4E A9 Ø1 8D 48 Ø3</td>
</tr>
<tr>
<td><code>4690</code></td>
<td>DØ 47 CE</td>
<td>5F Ø3 CE 5E Ø3</td>
</tr>
<tr>
<td><code>4698</code></td>
<td>AD 21 91 C9</td>
<td>FE DØ 3A A2</td>
</tr>
<tr>
<td><code>46A0</code></td>
<td>53 4C</td>
<td>5B 4Ø A9 ØØ FØ 12</td>
</tr>
<tr>
<td><code>46A8</code></td>
<td>AD 5C</td>
<td>Ø3 AE 5D Ø3 8D 5E</td>
</tr>
<tr>
<td><code>46B0</code></td>
<td>Ø3 8E</td>
<td>5F Ø3 A9 4Ø DØ Ø2</td>
</tr>
<tr>
<td><code>46B8</code></td>
<td>A9 8Ø 8D</td>
<td>48 Ø3 20 A4 49</td>
</tr>
<tr>
<td><code>46C0</code></td>
<td>FØ ØF</td>
<td>C9 20 DØ 56 20 45</td>
</tr>
<tr>
<td><code>46C8</code></td>
<td>48 20 E3</td>
<td>48 20 A4 49 DØ</td>
</tr>
<tr>
<td><code>46D0</code></td>
<td>4B 20 AE</td>
<td>45 AD 48 Ø3 FØ</td>
</tr>
<tr>
<td><code>46D8</code></td>
<td>1F 78 A9</td>
<td>AØ 8D 2E 91 A9</td>
</tr>
<tr>
<td><code>46E0</code></td>
<td>5F 8D 2E</td>
<td>91 A9 DF A2 45</td>
</tr>
<tr>
<td><code>46E8</code></td>
<td>8D 44 Ø3 8E</td>
<td>43 Ø3 A9 49</td>
</tr>
<tr>
<td><code>46F0</code></td>
<td>A2 ØØ 8D</td>
<td>28 91 8E 29 91</td>
</tr>
<tr>
<td><code>46F8</code></td>
<td>AE 42 Ø3</td>
<td>9A 78 AD 44 Ø3</td>
</tr>
<tr>
<td><code>4700</code></td>
<td>AE 43 Ø3</td>
<td>20 98 48 AD 3C</td>
</tr>
<tr>
<td><code>4708</code></td>
<td>Ø3 48 AD</td>
<td>3D Ø3 48 AD 3E</td>
</tr>
<tr>
<td><code>4710</code></td>
<td>Ø3 48 AD</td>
<td>3F Ø3 AE 40 Ø3</td>
</tr>
<tr>
<td><code>4718</code></td>
<td>AC 41 Ø3</td>
<td>4Ø 4C 6Ø 4Ø 2Ø</td>
</tr>
<tr>
<td><code>4720</code></td>
<td>31 48 8D 5A</td>
<td>Ø3 8E 5B Ø3</td>
</tr>
<tr>
<td><code>4728</code></td>
<td>A9 ØØ 8D 5C</td>
<td>Ø3 8D 5D Ø3</td>
</tr>
<tr>
<td><code>4730</code></td>
<td>20 42 48 8D</td>
<td>5C Ø3 8E 5D</td>
</tr>
<tr>
<td><code>4738</code></td>
<td>Ø3 4C 68 4Ø</td>
<td>20 CB 47 8D</td>
</tr>
<tr>
<td><code>4740</code></td>
<td>62 Ø3 8E 63</td>
<td>Ø3 2Ø 4Ø 48</td>
</tr>
<tr>
<td><code>4748</code></td>
<td>8D 4F Ø3 8E</td>
<td>5Ø Ø3 2Ø 42</td>
</tr>
<tr>
<td><code>4750</code></td>
<td>48 8D 51 Ø3 8E 52 Ø3 2Ø</td>
<td></td>
</tr>
<tr>
<td><code>4758</code></td>
<td>A4 49 FØ ØA</td>
<td>2Ø CF FF C9</td>
</tr>
<tr>
<td><code>4760</code></td>
<td>57 DØ Ø3 EE 4E Ø3 2Ø 21</td>
<td></td>
</tr>
<tr>
<td><code>4768</code></td>
<td>48 AE 56 Ø3 DØ 18 2Ø E7</td>
<td></td>
</tr>
<tr>
<td><code>4770</code></td>
<td>4Ø 9Ø 13 AC</td>
<td>4E Ø3 DØ 1A</td>
</tr>
<tr>
<td><code>4778</code></td>
<td>B1 FB 2Ø</td>
<td>87 42 AA BD F6</td>
</tr>
</tbody>
</table>
Appendix F

4780 4E D0 06 20 C4 40 4C 68
4788 40 AC 4D 03 C0 02 D0 33
4790 F0 03 8C 4D 03 88 38 B1
4798 FB AA ED 4F 03 C8 B1 FB
47A0 ED 50 03 90 1E 88 AD 51
47A8 03 F1 FB C8 AD 52 03 F1
47B0 FB 90 10 88 18 8A 6D 62
47B8 03 91 FB C8 B1 FB 6D 63
47C0 03 91 FB 20 1F 49 88 10
47C8 FA 30 9E 20 31 48 85 FD
47D0 86 FE 20 42 48 8D 54 03
47D8 8E 55 03 20 8C 48 20 45
47E0 48 85 FB 86 FC 60 20 31
47E8 48 B0 F6 20 45 48 B0 03
47F0 20 42 48 85 FD 86 FE 60
47F8 A5 FC 20 FF 47 A5 FB 48

4800 4A 4A 4A 4A 20 17 48 AA
4808 68 29 0F 20 17 48 48 8A
4810 20 D2 FF 68 4C D2 FF 18
4818 69 F6 90 02 69 06 69 3A
4820 60 A2 02 B5 FA 48 B5 FC
4828 95 FA 68 95 FC CA D0 F3
4830 60 A9 00 8D 59 03 20 8C
4838 48 C9 20 F0 F9 20 6C 48
4840 B0 08 20 8C 48 20 57 48
4848 90 07 AA 20 57 48 90 01
4850 60 4C 60 40 20 74 41 A9
4858 00 8D 59 03 20 8C 48 C9
4860 20 D0 09 20 8C 48 C9 20
4868 D0 0F 18 60 20 81 48 0A
4870 0A 0A 0A 8D 59 03 20 8C
4878 48 20 81 48 0D 59 03 38

4880 60 C9 3A 08 29 0F 28 90
4888 02 69 08 60 20 A4 49 D0
4890 FA 4C 65 40 A9 91 A2 43
4898 8D 14 03 8E 15 03 60 20
48A0 A4 49 F0 37 20 E6 47 A5
48A8 FB 05 FC F0 22 A5 9A C9
48B0 03 D0 9E A5 FB 8D 93 02
Appendix F

48B8 A5 FC 8D 94 02 A9 02 AA
48C0 A8 20 BA FF 20 C0 FF A2
48C8 02 20 C9 FF 4C 75 40 A9
48D0 02 20 C3 FF A9 03 85 9A
48D8 4C 68 40 A5 9A C9 03 F0
48E0 DC D0 F1 8D 3D 03 8E 3C
48E8 03 60 8D 4B 03 A0 00 20
48F0 38 49 B1 FB 20 FF 47 20
48F8 1F 49 CE 4B 03 D0 F0 60

4900 20 57 48 90 08 A2 00 81
4908 FB C1 FB D0 69 20 1F 49
4910 CE 4B 03 60 A9 3E 85 FB
4918 A9 03 85 FC A9 05 60 E6
4920 FB D0 09 E6 FF E6 FC D0
4928 03 EE 56 03 60 98 48 20
4930 AE 45 68 A2 2E 20 0E 48
4938 A9 20 4C D2 FF 20 0E 48
4940 A2 00 BD 76 4F 20 D2 FF
4948 E8 E0 1C D0 F5 A0 3B 20
4950 2D 49 AD 3C 03 20 FF 47
4958 AD 3D 03 20 FF 47 20 38
4960 49 AD 43 03 20 FF 47 AD
4968 44 03 20 FF 47 20 14 49
4970 20 EA 48 4C 68 40 4C 60
4978 40 20 31 48 20 E3 48 20

4980 42 48 8D 44 03 8E 43 03
4988 20 14 49 8D 4B 03 20 8C
4990 48 20 00 49 D0 F8 F0 DB
4998 20 CF FF C9 20 F0 F9 D0
49A0 06 20 F0 47 20 CF FF C9
49A8 0D 60 A0 01 84 BA A9 00
49B0 A2 65 A0 03 20 BD FF A8
49B8 20 E6 47 AD 49 03 C9 53
49C0 D0 08 20 A4 49 F0 AF 20
49C8 EB 47 20 98 49 F0 29 C9
49D0 22 D0 A3 20 CF FF C9 22
49D8 F0 0B 91 BB E6 B7 C8 C0
49E0 51 90 F0 B0 91 20 A4 49
<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Code</th>
<th>Value</th>
<th>Code</th>
<th>Value</th>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>4B10</code></td>
<td><code>4B</code></td>
<td><code>85</code></td>
<td><code>FB</code></td>
<td><code>20</code></td>
<td><code>22</code></td>
<td><code>4B</code></td>
<td><code>20</code></td>
</tr>
<tr>
<td><code>4B18</code></td>
<td><code>4B</code></td>
<td><code>CA</code></td>
<td><code>D0</code></td>
<td><code>F7</code></td>
<td><code>08</code></td>
<td><code>20</code></td>
<td><code>38</code></td>
</tr>
<tr>
<td><code>4B20</code></td>
<td><code>28</code></td>
<td><code>60</code></td>
<td><code>20</code></td>
<td><code>A4</code></td>
<td><code>49</code></td>
<td><code>F0</code></td>
<td><code>0F</code></td>
</tr>
<tr>
<td><code>4B28</code></td>
<td><code>20</code></td>
<td><code>F0</code></td>
<td><code>0B</code></td>
<td><code>C9</code></td>
<td><code>30</code></td>
<td><code>90</code></td>
<td><code>0B</code></td>
</tr>
<tr>
<td><code>4B30</code></td>
<td><code>3A</code></td>
<td><code>B0</code></td>
<td><code>07</code></td>
<td><code>29</code></td>
<td><code>0F</code></td>
<td><code>60</code></td>
<td><code>68</code></td>
</tr>
<tr>
<td><code>4B38</code></td>
<td><code>18</code></td>
<td><code>60</code></td>
<td><code>4C</code></td>
<td><code>60</code></td>
<td><code>40</code></td>
<td><code>85</code></td>
<td><code>FE</code></td>
</tr>
<tr>
<td><code>4B40</code></td>
<td><code>FC</code></td>
<td><code>48</code></td>
<td><code>A5</code></td>
<td><code>FB</code></td>
<td><code>48</code></td>
<td><code>06</code></td>
<td><code>FB</code></td>
</tr>
<tr>
<td><code>4B48</code></td>
<td><code>FC</code></td>
<td><code>06</code></td>
<td><code>FB</code></td>
<td><code>26</code></td>
<td><code>FC</code></td>
<td><code>68</code></td>
<td><code>65</code></td>
</tr>
<tr>
<td><code>4B50</code></td>
<td><code>85</code></td>
<td><code>FB</code></td>
<td><code>68</code></td>
<td><code>65</code></td>
<td><code>FC</code></td>
<td><code>85</code></td>
<td><code>FC</code></td>
</tr>
<tr>
<td><code>4B58</code></td>
<td><code>FB</code></td>
<td><code>26</code></td>
<td><code>FC</code></td>
<td><code>A5</code></td>
<td><code>FE</code></td>
<td><code>65</code></td>
<td><code>FB</code></td>
</tr>
<tr>
<td><code>4B60</code></td>
<td><code>FB</code></td>
<td><code>A9</code></td>
<td><code>00</code></td>
<td><code>65</code></td>
<td><code>FC</code></td>
<td><code>85</code></td>
<td><code>FC</code></td>
</tr>
<tr>
<td><code>4B68</code></td>
<td><code>20</code></td>
<td><code>C2</code></td>
<td><code>4B</code></td>
<td><code>8D</code></td>
<td><code>55</code></td>
<td><code>03</code></td>
<td><code>48</code></td>
</tr>
<tr>
<td><code>4B70</code></td>
<td><code>20</code></td>
<td><code>38</code></td>
<td><code>49</code></td>
<td><code>20</code></td>
<td><code>38</code></td>
<td><code>49</code></td>
<td><code>68</code></td>
</tr>
<tr>
<td><code>4B78</code></td>
<td><code>FF</code></td>
<td><code>47</code></td>
<td><code>20</code></td>
<td><code>38</code></td>
<td><code>49</code></td>
<td><code>68</code></td>
<td><code>AA</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>4B80</code></td>
<td><code>00</code></td>
<td><code>20</code></td>
<td><code>F1</code></td>
<td><code>4A</code></td>
<td><code>20</code></td>
<td><code>38</code></td>
<td><code>49</code></td>
</tr>
<tr>
<td><code>4B88</code></td>
<td><code>86</code></td>
<td><code>4A</code></td>
<td><code>4C</code></td>
<td><code>68</code></td>
<td><code>40</code></td>
<td><code>20</code></td>
<td><code>9F</code></td>
</tr>
<tr>
<td><code>4B90</code></td>
<td><code>20</code></td>
<td><code>38</code></td>
<td><code>49</code></td>
<td><code>20</code></td>
<td><code>F8</code></td>
<td><code>47</code></td>
<td><code>20</code></td>
</tr>
<tr>
<td><code>4B98</code></td>
<td><code>4A</code></td>
<td><code>20</code></td>
<td><code>A0</code></td>
<td><code>4A</code></td>
<td><code>4C</code></td>
<td><code>68</code></td>
<td><code>40</code></td>
</tr>
<tr>
<td><code>4BA0</code></td>
<td><code>0F</code></td>
<td><code>A9</code></td>
<td><code>00</code></td>
<td><code>85</code></td>
<td><code>FB</code></td>
<td><code>85</code></td>
<td><code>FC</code></td>
</tr>
<tr>
<td><code>4BA8</code></td>
<td><code>C2</code></td>
<td><code>4B</code></td>
<td><code>20</code></td>
<td><code>2B</code></td>
<td><code>4B</code></td>
<td><code>20</code></td>
<td><code>BC</code></td>
</tr>
<tr>
<td><code>4BB0</code></td>
<td><code>20</code></td>
<td><code>22</code></td>
<td><code>4B</code></td>
<td><code>20</code></td>
<td><code>BC</code></td>
<td><code>4B</code></td>
<td><code>CA</code></td>
</tr>
<tr>
<td><code>4BB8</code></td>
<td><code>F7</code></td>
<td><code>4C</code></td>
<td><code>38</code></td>
<td><code>49</code></td>
<td><code>20</code></td>
<td><code>FB</code></td>
<td><code>26</code></td>
</tr>
<tr>
<td><code>4BC0</code></td>
<td><code>FC</code></td>
<td><code>60</code></td>
<td><code>20</code></td>
<td><code>8C</code></td>
<td><code>48</code></td>
<td><code>C9</code></td>
<td><code>20</code></td>
</tr>
<tr>
<td><code>4BC8</code></td>
<td><code>F9</code></td>
<td><code>60</code></td>
<td><code>20</code></td>
<td><code>54</code></td>
<td><code>48</code></td>
<td><code>8D</code></td>
<td><code>88</code></td>
</tr>
<tr>
<td><code>4BD0</code></td>
<td><code>A6</code></td>
<td><code>FB</code></td>
<td><code>A4</code></td>
<td><code>FC</code></td>
<td><code>20</code></td>
<td><code>8A</code></td>
<td><code>FE</code></td>
</tr>
<tr>
<td><code>4BD8</code></td>
<td><code>FD</code></td>
<td><code>A4</code></td>
<td><code>FE</code></td>
<td><code>20</code></td>
<td><code>7B</code></td>
<td><code>FE</code></td>
<td><code>20</code></td>
</tr>
<tr>
<td><code>4BE0</code></td>
<td><code>E5</code></td>
<td><code>20</code></td>
<td><code>A4</code></td>
<td><code>E3</code></td>
<td><code>4C</code></td>
<td><code>68</code></td>
<td><code>40</code></td>
</tr>
<tr>
<td><code>4BE8</code></td>
<td><code>F0</code></td>
<td><code>47</code></td>
<td><code>4C</code></td>
<td><code>DB</code></td>
<td><code>47</code></td>
<td><code>20</code></td>
<td><code>E7</code></td>
</tr>
<tr>
<td><code>4BF0</code></td>
<td><code>18</code></td>
<td><code>A5</code></td>
<td><code>FB</code></td>
<td><code>65</code></td>
<td><code>FD</code></td>
<td><code>85</code></td>
<td><code>FB</code></td>
</tr>
<tr>
<td><code>4BF8</code></td>
<td><code>FC</code></td>
<td><code>65</code></td>
<td><code>FE</code></td>
<td><code>85</code></td>
<td><code>FC</code></td>
<td><code>4C</code></td>
<td><code>0D</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>4C00</code></td>
<td><code>20</code></td>
<td><code>E7</code></td>
<td><code>4B</code></td>
<td><code>20</code></td>
<td><code>F0</code></td>
<td><code>40</code></td>
<td><code>84</code></td>
</tr>
<tr>
<td><code>4C08</code></td>
<td><code>AD</code></td>
<td><code>53</code></td>
<td><code>03</code></td>
<td><code>85</code></td>
<td><code>FB</code></td>
<td><code>20</code></td>
<td><code>38</code></td>
</tr>
<tr>
<td><code>4C10</code></td>
<td><code>20</code></td>
<td><code>F8</code></td>
<td><code>47</code></td>
<td><code>4C</code></td>
<td><code>68</code></td>
<td><code>40</code></td>
<td><code>A9</code></td>
</tr>
<tr>
<td><code>4C18</code></td>
<td><code>2C</code></td>
<td><code>A9</code></td>
<td><code>00</code></td>
<td><code>8D</code></td>
<td><code>0B</code></td>
<td><code>90</code></td>
<td><code>4C</code></td>
</tr>
<tr>
<td><code>4C20</code></td>
<td><code>40</code></td>
<td><code>78</code></td>
<td><code>20</code></td>
<td><code>52</code></td>
<td><code>FD</code></td>
<td><code>58</code></td>
<td><code>A9</code></td>
</tr>
<tr>
<td><code>4C28</code></td>
<td><code>85</code></td>
<td><code>B2</code></td>
<td><code>AE</code></td>
<td><code>42</code></td>
<td><code>03</code></td>
<td><code>9A</code></td>
<td><code>A5</code></td>
</tr>
<tr>
<td><code>4C30</code></td>
<td><code>C9</code></td>
<td><code>E6</code></td>
<td><code>F0</code></td>
<td><code>95</code></td>
<td><code>6C</code></td>
<td><code>00</code></td>
<td><code>C0</code></td>
</tr>
<tr>
<td><code>4C38</code></td>
<td><code>E7</code></td>
<td><code>4B</code></td>
<td><code>20</code></td>
<td><code>21</code></td>
<td><code>48</code></td>
<td><code>20</code></td>
<td><code>38</code></td>
</tr>
<tr>
<td><code>4C40</code></td>
<td><code>A0</code></td>
<td><code>00</code></td>
<td><code>8C</code></td>
<td><code>54</code></td>
<td><code>03</code></td>
<td><code>8C</code></td>
<td><code>55</code></td>
</tr>
</tbody>
</table>
4D80  4D  A5  FB  A6  FC  85  FD  86
4D88  FE  A9  10  8D  5E  03  38  A5
4D90  FD  ED  5E  03  85  FB  A5  FE
4D98  E9  00  85  FC  20  C9  4D  20
4DA0  6F  42  20  F0  40  F0  07  B0
4DA8  F3  CE  5E  03  D0  E0  EE  4D
4DB0  03  AD  4D  03  20  AB  43  A2
4DB8  00  A1  FB  8E  4E  03  A9  2C
4DC0  20  33  49  20  16  42  4C  68
4DC8  4D  A2  00  A1  FB  4C  87  42
4DD0  A6  D2  20  D7  4D  A6  F4  E8
4DD8  86  AD  86  FE  A2  00  86  AC
4DE0  A9  2C  85  FD  A0  CE  E8  88
4DE8  B1  AC  91  FD  98  D0  F8  C6
4DF0  AD  C6  FE  CA  10  F1  A9  20
4DF8  A6  D2  86  FE  84  FD  A0  2B

4E00  91  FD  88  10  FB  A9  13  4C
4E08  D2  FF  C0  16  D0  02  38  60
4E10  20  51  4E  C9  20  F0  F3  88
4E18  20  3A  4E  AA  20  3A  4E  85
4E20  FB  86  FC  A9  FF  8D  64  03
4E28  85  CC  A5  CF  F0  0A  A5  CE
4E30  A4  D3  91  D1  A9  00  85  CF
4E38  18  60  20  51  4E  20  81  48
4E40  0A  0A  0A  0A  8D  59  03  20
4E48  51  4E  20  81  48  0D  59  03
4E50  60  B1  FD  C8  29  7F  C9  20
4E58  B0  02  09  40  60  BD  98  4D
4E60  20  D2  FF  E8  D0  F7  60  00
4E68  00  00  00  00  00  00  00  0D  56
4E70  49  43  32  30  20  4D  49  43
4E78  52  4F  4D  4F  4E  20  56  31

4E80  2E  32  20  20  20  42  49  4C
4E88  4C  20  59  45  45  20  32  32
4E90  20  4A  41  4E  20  20  38  33
4E98  40  02  45  03  D0  08  40  09
4EA0  30  22  45  33  D0  08  40  09
4EA8  40  02  45  33  D0  08  40  09
4EB0  40  02  45  B3  D0  08  40  09
<table>
<thead>
<tr>
<th>Address</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>4EB8</td>
<td>00 22 44 33 D0 8C 44 00</td>
</tr>
<tr>
<td>4EC0</td>
<td>11 22 44 33 D0 8C 44 9A</td>
</tr>
<tr>
<td>4EC8</td>
<td>10 22 44 33 D0 08 40 09</td>
</tr>
<tr>
<td>4ED0</td>
<td>10 22 44 33 D0 08 40 09</td>
</tr>
<tr>
<td>4ED8</td>
<td>62 13 78 A9 00 21 81 82</td>
</tr>
<tr>
<td>4EE0</td>
<td>00 00 59 4D 91 92 86 4A</td>
</tr>
<tr>
<td>4EE8</td>
<td>85 9D 2C 29 2C 23 28 24</td>
</tr>
<tr>
<td>4EF0</td>
<td>59 00 58 24 24 00 1C 8A</td>
</tr>
<tr>
<td>4EF8</td>
<td>1C 23 5D 8B 1B A1 9D 8A</td>
</tr>
<tr>
<td>4F00</td>
<td>1D 23 9D 8B 1D A1 00 29</td>
</tr>
<tr>
<td>4F08</td>
<td>19 AE 69 A8 19 23 24 53</td>
</tr>
<tr>
<td>4F10</td>
<td>1B 23 24 53 19 A1 00 1A</td>
</tr>
<tr>
<td>4F18</td>
<td>5B 5B A5 69 24 24 AE AE</td>
</tr>
<tr>
<td>4F20</td>
<td>A8 AD 29 00 7C 00 15 9C</td>
</tr>
<tr>
<td>4F28</td>
<td>6D 9C A5 69 29 53 84 13</td>
</tr>
<tr>
<td>4F30</td>
<td>34 11 A5 69 23 A0 D8 62</td>
</tr>
<tr>
<td>4F38</td>
<td>5A 48 26 62 94 88 54 44</td>
</tr>
<tr>
<td>4F40</td>
<td>C8 54 68 44 E8 94 00 B4</td>
</tr>
<tr>
<td>4F48</td>
<td>08 84 74 B4 28 6E 74 F4</td>
</tr>
<tr>
<td>4F50</td>
<td>CC 4A 72 F2 A4 8A 00 AA</td>
</tr>
<tr>
<td>4F58</td>
<td>A2 A2 74 74 74 72 44 68</td>
</tr>
<tr>
<td>4F60</td>
<td>B2 32 B2 00 22 00 1A 1A</td>
</tr>
<tr>
<td>4F68</td>
<td>26 26 72 72 88 C8 C4 CA</td>
</tr>
<tr>
<td>4F70</td>
<td>26 48 44 44 A2 C8 0D 20</td>
</tr>
<tr>
<td>4F78</td>
<td>20 20 20 50 43 20 20 49</td>
</tr>
<tr>
<td>4F80</td>
<td>52 51 20 20 53 52 20 41</td>
</tr>
<tr>
<td>4F88</td>
<td>43 20 58 52 20 59 52 20</td>
</tr>
<tr>
<td>4F90</td>
<td>53 50 41 42 43 44 46 47</td>
</tr>
<tr>
<td>4F98</td>
<td>48 4C 4D 4E 51 52 28 54</td>
</tr>
<tr>
<td>4FA0</td>
<td>57 58 2C 3A 3B 24 23 22</td>
</tr>
<tr>
<td>4FA8</td>
<td>2B 2D 4F 49 4A 25 26 45</td>
</tr>
<tr>
<td>4FB0</td>
<td>56 29 3D 5C FF AA 49 9F</td>
</tr>
<tr>
<td>4FB8</td>
<td>48 3D 44 1F 47 02 41 F9</td>
</tr>
<tr>
<td>4FC0</td>
<td>41 87 41 A4 46 A0 41 AA</td>
</tr>
</tbody>
</table>
Appendix F

4FC8 49 B0 43 3C 47 A8 46 40
4FD0 49 16 4C 06 41 B8 46 2A
4FD8 4C 0C 43 15 44 79 49 64
4FE0 4A F4 4A 68 4B ED 4B 00
4FE8 4C 35 4A CA 4B 2C 4A 8D
4FF0 4B 37 4C 21 4C AA 49 19
4FF8 4C 40 43 40 43 40 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
```
Appendix F

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
Appendix F

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. Supermon64 will then disassemble that page again.

• Printing disassembler

.P 2000,2040
2000 A9 12      LDA  #$12
2002 9D 00 80      STA  $8000,X
2005 AA           TAX

... ...

203F A2 00      LDX  #$00

To engage printer, set up beforehand:

OPEN 4,4:CMD4

• 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. 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.
Appendix F

. S "0: PROGRAM NAME", 08, 1200, 1F50

SAVE to disk drive #0 memory from 1200 hex up to but not including 1F50 hex and name it PROGRAM NAME.

• Transfer memory

. T 1000 1100 5000

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.

| Ó800 00 1A 04 64 00 99 22 93 |
| Ó808 12 1D 1D 1D 53 55 50 |
| Ó810 45 52 20 36 34 2D 4D 4F |
| Ó818 4E 00 31 04 6E 00 99 22 |
| Ó820 11 20 20 20 20 20 20 20 |
| Ó828 20 20 20 20 20 20 20 20 |
| Ó830 00 4B 04 78 00 99 22 11 |
| Ó838 20 2E 2E 4A 49 4D 20 42 |
| Ó840 55 54 54 45 52 46 49 45 |
| Ó848 4C 44 00 66 04 82 00 9E |
| Ó850 28 C2 28 34 33 29 AA 32 |
| Ó858 35 36 AC C2 28 34 34 29 |
| Ó860 AA 31 32 37 29 00 00 00 |
| Ó868 AA AA AA AA AA AA AA AA |
| Ó870 AA AA AA AA AA AA AA AA |
| Ó878 AA AA AA AA AA AA AA AA |

324
Appendix F

0880 A5 2D 85 22 A5 2E 85 23
0888 A5 37 85 24 A5 38 85 25
0890 A0 00 A5 22 D0 02 C6 23
0898 C6 22 B1 22 D0 3C A5 22
08A0 D0 02 C6 23 C6 22 B1 22
08A8 F0 21 85 26 A5 22 D0 02
08B0 C6 23 C6 22 B1 22 18 65
08B8 24 AA A5 26 65 25 48 A5
08C0 37 D0 02 C6 38 C6 37 68
08C8 91 37 8A 48 A5 37 D0 02
08D0 C6 38 C6 37 68 91 37 18
08D8 90 B6 C9 4F D0 ED A5 37
08E0 85 33 A5 38 85 34 6C 37
08E8 00 4F 4F 4F AD E6 FF
08F0 00 8D 16 03 AD E7 FF 00
08F8 8D 17 03 A9 80 20 90 FF

0900 00 00 D8 68 8D 3E 02 68
0908 8D 3D 02 68 8D 3C 02 68
0910 8D 3B 02 68 AA 68 A8 38
0918 8A E9 02 8D 3A 02 98 E9
0920 00 00 8D 39 02 BA 8E 3F
0928 02 20 57 FD 00 A2 42 A9
0930 2A 20 57 FA 00 A9 52 D0
0938 34 E6 C1 D0 06 E6 C2 D0
0940 02 E6 26 60 20 CF FF C9
0948 0D D0 F8 68 68 A9 90 20
0950 D2 FF A9 00 00 85 26 A2
0958 0D A9 2E 20 57 FA 00 A9
0960 05 20 D2 FF 20 3E F8 00
0968 C9 2E F0 F9 C9 20 F0 F5
0970 A2 0E DD B7 FF 00 D0 0C
0978 8A 0A AA BD C7 FF 00 48
Appendix F

0980 BD C6 FF 00 48 60 CA 10
0988 EC 4C ED FA 00 A5 C1 8D
0990 3A 02 A5 C2 8D 39 02 60
0998 A9 08 85 1D A0 00 00 20
09A0 54 FD 00 B1 C1 20 48 FA
09A8 00 20 33 F8 00 C6 1D D0
09B0 F1 60 20 88 FA 00 90 0B
09B8 A2 00 00 81 C1 C1 C1 F0
09C0 03 4C ED FA 00 20 33 F8
09C8 00 C6 1D 60 A9 3B 85 C1
09D0 A9 02 85 C2 A9 05 60 98
09D8 48 20 57 FD 00 68 A2 2E
09E0 4C 57 FA 00 A9 90 20 D2
09E8 FF A2 00 00 BD EA FF 00
09F0 20 D2 FF E8 E0 16 D0 F5
09F8 A0 3B 20 C2 F8 00 AD 39

0A00 02 20 48 FA 00 AD 3A 02
0A08 20 48 FA 00 20 B7 F8 00
0A10 20 8D F8 00 F0 5C 20 3E
0A18 F8 00 20 79 FA 00 90 33
0A20 20 69 FA 00 20 3E F8 00
0A28 20 79 FA 00 90 28 20 69
0A30 FA 00 A9 90 20 D2 FF 20
0A38 E1 FF F0 3C A6 26 D0 38
0A40 A5 C3 C5 C1 A5 C4 E5 C2
0A48 90 2E A0 3A 20 C2 F8 00
0A50 20 41 FA 00 20 8B F8 00
0A58 F0 E0 4C ED FA 00 20 79
0A60 FA 00 90 03 20 80 F8 00
0A68 20 B7 F8 00 D0 07 20 79
0A70 FA 00 90 EB A9 0B 85 1D
0A78 20 3E F8 00 20 A1 F8 00

0A80 D0 F8 4C 47 F8 00 20 CF
0A88 FF C9 0D F0 0C C9 20 D0
0A90 D1 20 79 FA 00 90 03 20
<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0A98</td>
<td>80 F8 00 A9 90 20 D2 FF</td>
<td></td>
</tr>
<tr>
<td>0AA0</td>
<td>AE 3F 02 9A 78 AD 39 02</td>
<td></td>
</tr>
<tr>
<td>0AA8</td>
<td>48 AD 3A 02 48 AD 3B 02</td>
<td></td>
</tr>
<tr>
<td>0AB0</td>
<td>48 AD 3C 02 AE 3D 02 AC</td>
<td></td>
</tr>
<tr>
<td>0AB8</td>
<td>3E 02 40 A9 90 20 D2 FF</td>
<td></td>
</tr>
<tr>
<td>0AC0</td>
<td>AE 3F 02 9A 6C 02 A0 A0</td>
<td></td>
</tr>
<tr>
<td>0AC8</td>
<td>01 84 BA 84 B9 88 84 B7</td>
<td></td>
</tr>
<tr>
<td>0AD0</td>
<td>84 90 84 93 A9 40 85 BB</td>
<td></td>
</tr>
<tr>
<td>0AD8</td>
<td>A9 02 85 BC 20 CF FF C9</td>
<td></td>
</tr>
<tr>
<td>0AE0</td>
<td>20 F0 F9 C9 0D F0 38 C9</td>
<td></td>
</tr>
<tr>
<td>0AE8</td>
<td>22 D0 14 20 CF FF C9 22</td>
<td></td>
</tr>
<tr>
<td>0AF0</td>
<td>F0 10 C9 0D F0 29 91 BB</td>
<td></td>
</tr>
<tr>
<td>0AF8</td>
<td>E6 B7 C8 C0 10 D0 EC 4C</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B00</td>
<td>ED FA 00 20 CF FF C9 0D</td>
<td></td>
</tr>
<tr>
<td>0B08</td>
<td>F0 16 C9 2C D0 DC 20 88</td>
<td></td>
</tr>
<tr>
<td>0B10</td>
<td>FA 00 29 0F F0 E9 C9 03</td>
<td></td>
</tr>
<tr>
<td>0B18</td>
<td>F0 E5 85 BA 20 CF FF C9</td>
<td></td>
</tr>
<tr>
<td>0B20</td>
<td>0D 60 6C 30 03 6C 32 03</td>
<td></td>
</tr>
<tr>
<td>0B28</td>
<td>20 96 F9 00 D0 D4 A9 90</td>
<td></td>
</tr>
<tr>
<td>0B30</td>
<td>20 D2 FF A9 00 00 20 EF</td>
<td></td>
</tr>
<tr>
<td>0B38</td>
<td>F9 00 A5 90 29 10 D0 C4</td>
<td></td>
</tr>
<tr>
<td>0B40</td>
<td>4C 47 F8 00 20 96 F9 00</td>
<td></td>
</tr>
<tr>
<td>0B48</td>
<td>C9 2C D0 BA 20 79 FA 00</td>
<td></td>
</tr>
<tr>
<td>0B50</td>
<td>20 69 FA 00 20 CF FF C9</td>
<td></td>
</tr>
<tr>
<td>0B58</td>
<td>2C D0 AD 20 79 FA 00 A5</td>
<td></td>
</tr>
<tr>
<td>0B60</td>
<td>C1 85 AE A5 C2 85 AF 20</td>
<td></td>
</tr>
<tr>
<td>0B68</td>
<td>69 FA 00 20 CF FF C9 0D</td>
<td></td>
</tr>
<tr>
<td>0B70</td>
<td>D0 98 A9 90 20 D2 FF 20</td>
<td></td>
</tr>
<tr>
<td>0B78</td>
<td>F2 F9 00 4C 47 F8 00 A5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B80</td>
<td>C2 20 48 FA 00 A5 C1 48</td>
<td></td>
</tr>
<tr>
<td>0B88</td>
<td>4A 4A 4A 4A 20 60 FA 00</td>
<td></td>
</tr>
<tr>
<td>0B90</td>
<td>AA 68 29 0F 20 60 FA 00</td>
<td></td>
</tr>
<tr>
<td>0B98</td>
<td>48 8A 20 D2 FF 68 4C D2</td>
<td></td>
</tr>
<tr>
<td>0BA0</td>
<td>FF 09 30 C9 3A 90 02 69</td>
<td></td>
</tr>
<tr>
<td>0BA8</td>
<td>06 60 A2 02 B5 C0 48 B5</td>
<td></td>
</tr>
</tbody>
</table>
0BB0 C2 95 C0 68 95 C2 CA D0
0BB8 F3 60 20 88 FA 00 90 02
0BC0 85 C2 20 88 FA 00 90 02
0BC8 85 C1 60 A9 00 00 85 2A
0BD0 20 3E F8 00 C9 20 D0 09
0BD8 20 3E F8 00 C9 20 D0 0E
0BE0 18 60 20 AF FA 00 0A 0A
0BE8 0A 0A 85 2A 20 3E F8 00
0BF0 20 AF FA 00 05 2A 38 60
0BF8 C9 3A 90 02 69 08 29 0F

0C00 60 A2 02 2C A2 00 00 B4
0C08 C1 D0 08 B4 C2 D0 02 E6
0C10 26 D6 C2 D6 C1 60 20 3E
0C18 F8 00 C9 20 F0 F9 60 A9
0C20 00 00 8D 00 00 01 20 CC
0C28 FA 00 20 8F FA 00 20 7C
0C30 FA 00 90 09 60 20 3E F8
0C38 00 20 79 FA 00 B0 DE AE
0C40 3F 02 9A A9 90 20 D2 FF
0C48 A9 3F 20 D2 FF 4C 47 F8
0C50 00 20 54 FD 00 CA D0 FA
0C58 60 E6 C3 D0 02 E6 C4 60
0C60 A2 02 B5 C0 48 B5 27 95
0C68 C0 68 95 27 CA D0 F3 60
0C70 A5 C3 A4 C4 38 E9 02 B0
0C78 0E 88 90 0B A5 28 A4 29

0C80 4C 33 FB 00 A5 C3 A4 C4
0C88 38 E5 C1 85 1E 98 E5 C2
0C90 A8 05 1E 60 20 D4 FA 00
0C98 20 69 FA 00 20 E5 FA 00
0CA0 20 0C FB 00 20 E5 FA 00
0CA8 20 2F FB 00 20 69 FA 00
0CB0 90 15 A6 26 D0 64 20 28
0CB8 FB 00 90 5F A1 C1 81 C3
0CC0 20 05 FB 00 20 33 F8 00
0CC8 D0 EB 20 28 FB 00 18 A5
0CD0 1E 65 C3 85 C3 98 65 C4
0CD8 85 C4 20 0C FB 00 A6 26
0CE0 D0 3D A1 C1 81 C3 20 28
Appendix F

\[ \text{0CE8 FB 00 B0 34 20 B8 FA 00} \]
\[ \text{0CF0 20 BB FA 00 4C 7D FB 00} \]
\[ \text{0CF8 20 D4 FA 00 20 69 FA 00} \]

\[ \text{0D00 20 E5 FA 00 20 69 FA 00} \]
\[ \text{0D08 20 3E F8 00 20 88 FA 00} \]
\[ \text{0D10 90 14 85 1D A6 26 D0 11} \]
\[ \text{0D18 20 2F FB 00 90 0C A5 1D} \]
\[ \text{0D20 81 C1 20 33 F8 00 D0 EE} \]
\[ \text{0D28 4C ED FA 00 4C 47 F8 00} \]
\[ \text{0D30 20 D4 FA 00 20 69 FA 00} \]
\[ \text{0D38 20 E5 FA 00 20 69 FA 00} \]
\[ \text{0D40 20 3E F8 00 A2 00 00 20} \]
\[ \text{0D48 3E F8 00 C9 27 D0 14 20} \]
\[ \text{0D50 3E F8 00 9D 10 02 E8 20} \]
\[ \text{0D58 CF FF C9 0D F0 22 E0 20} \]
\[ \text{0D60 D0 F1 F0 1C 8E 00 00 01} \]
\[ \text{0D68 20 8F FA 00 90 C6 9D 10} \]
\[ \text{0D70 02 E8 20 CF FF C9 0D F0} \]
\[ \text{0D78 09 20 88 FA 00 90 B6 E0} \]

\[ \text{0D80 20 D0 EC 86 1C A9 90 20} \]
\[ \text{0D88 D2 FF 20 57 FD 00 A2 00} \]
\[ \text{0D90 00 A0 00 00 B1 C1 DD 10} \]
\[ \text{0D98 02 D0 0C C8 E8 E4 1C D0} \]
\[ \text{0DA0 F3 20 41 FA 00 20 54 FD} \]
\[ \text{0DA8 00 20 33 F8 00 A6 26 D0} \]
\[ \text{0DB0 8D 20 2F FB 00 B0 DD 4C} \]
\[ \text{0DB8 47 F8 00 20 D4 FA 00 85} \]
\[ \text{0DC0 20 A5 C2 85 21 A2 00 00} \]
\[ \text{0DC8 86 28 A9 93 20 D2 FF A9} \]
\[ \text{0DD0 90 20 D2 FF A9 16 85 1D} \]
\[ \text{0DD8 20 6A FC 00 20 CA FC 00} \]
\[ \text{0DE0 85 C1 84 C2 C6 1D D0 F2} \]
\[ \text{0DE8 A9 91 20 D2 FF 4C 47 F8} \]
\[ \text{0DF0 00 A0 2C 20 C2 F8 00 20} \]
\[ \text{0DF8 54 FD 00 20 41 FA 00 20} \]

\[ \text{0E00 54 FD 00 A2 00 00 A1 C1} \]
\[ \text{0E08 20 D9 FC 00 48 20 1F FD} \]
<table>
<thead>
<tr>
<th>ASCII Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F40</td>
<td>A1</td>
</tr>
<tr>
<td>0F48</td>
<td>85</td>
</tr>
<tr>
<td>0F50</td>
<td>FC</td>
</tr>
<tr>
<td>0F58</td>
<td>FF</td>
</tr>
<tr>
<td>0F60</td>
<td>FA</td>
</tr>
<tr>
<td>0F68</td>
<td>CC</td>
</tr>
<tr>
<td>0F70</td>
<td>03</td>
</tr>
<tr>
<td>0F78</td>
<td>06</td>
</tr>
<tr>
<td>0F80</td>
<td>F6</td>
</tr>
<tr>
<td>0F88</td>
<td>C9</td>
</tr>
<tr>
<td>0F90</td>
<td>F5</td>
</tr>
<tr>
<td>0F98</td>
<td>9C</td>
</tr>
<tr>
<td>0FA0</td>
<td>C1</td>
</tr>
<tr>
<td>0FA8</td>
<td>10</td>
</tr>
<tr>
<td>0FB0</td>
<td>00</td>
</tr>
<tr>
<td>0FBB</td>
<td>F0</td>
</tr>
<tr>
<td>0FC0</td>
<td>26</td>
</tr>
<tr>
<td>0FC8</td>
<td>29</td>
</tr>
<tr>
<td>0FD0</td>
<td>FF</td>
</tr>
<tr>
<td>0FD8</td>
<td>A2</td>
</tr>
<tr>
<td>0FE0</td>
<td>F0</td>
</tr>
<tr>
<td>0FE8</td>
<td>B0</td>
</tr>
<tr>
<td>0FF0</td>
<td>20</td>
</tr>
<tr>
<td>0FF8</td>
<td>EB</td>
</tr>
<tr>
<td>1000</td>
<td>00</td>
</tr>
<tr>
<td>1008</td>
<td>00</td>
</tr>
<tr>
<td>1010</td>
<td>20</td>
</tr>
<tr>
<td>1018</td>
<td>FE</td>
</tr>
<tr>
<td>1020</td>
<td>D0</td>
</tr>
<tr>
<td>1028</td>
<td>F0</td>
</tr>
<tr>
<td>1030</td>
<td>20</td>
</tr>
<tr>
<td>1038</td>
<td>04</td>
</tr>
<tr>
<td>1040</td>
<td>00</td>
</tr>
<tr>
<td>1048</td>
<td>A4</td>
</tr>
<tr>
<td>1050</td>
<td>91</td>
</tr>
<tr>
<td>1058</td>
<td>C1</td>
</tr>
<tr>
<td>1060</td>
<td>C2</td>
</tr>
<tr>
<td>1068</td>
<td>20</td>
</tr>
</tbody>
</table>

Appendix F

<table>
<thead>
<tr>
<th>ASCII Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F40</td>
<td>A1</td>
</tr>
<tr>
<td>0F48</td>
<td>85</td>
</tr>
<tr>
<td>0F50</td>
<td>FC</td>
</tr>
<tr>
<td>0F58</td>
<td>FF</td>
</tr>
<tr>
<td>0F60</td>
<td>FA</td>
</tr>
<tr>
<td>0F68</td>
<td>CC</td>
</tr>
<tr>
<td>0F70</td>
<td>03</td>
</tr>
<tr>
<td>0F78</td>
<td>06</td>
</tr>
<tr>
<td>0F80</td>
<td>F6</td>
</tr>
<tr>
<td>0F88</td>
<td>C9</td>
</tr>
<tr>
<td>0F90</td>
<td>F5</td>
</tr>
<tr>
<td>0F98</td>
<td>9C</td>
</tr>
<tr>
<td>0FA0</td>
<td>C1</td>
</tr>
<tr>
<td>0FA8</td>
<td>10</td>
</tr>
<tr>
<td>0FB0</td>
<td>00</td>
</tr>
<tr>
<td>0FBB</td>
<td>F0</td>
</tr>
<tr>
<td>0FC0</td>
<td>26</td>
</tr>
<tr>
<td>0FC8</td>
<td>29</td>
</tr>
<tr>
<td>0FD0</td>
<td>FF</td>
</tr>
<tr>
<td>0FD8</td>
<td>A2</td>
</tr>
<tr>
<td>0FE0</td>
<td>F0</td>
</tr>
<tr>
<td>0FE8</td>
<td>B0</td>
</tr>
<tr>
<td>0FF0</td>
<td>20</td>
</tr>
<tr>
<td>0FF8</td>
<td>EB</td>
</tr>
<tr>
<td>1000</td>
<td>00</td>
</tr>
<tr>
<td>1008</td>
<td>00</td>
</tr>
<tr>
<td>1010</td>
<td>20</td>
</tr>
<tr>
<td>1018</td>
<td>FE</td>
</tr>
<tr>
<td>1020</td>
<td>D0</td>
</tr>
<tr>
<td>1028</td>
<td>F0</td>
</tr>
<tr>
<td>1030</td>
<td>20</td>
</tr>
<tr>
<td>1038</td>
<td>04</td>
</tr>
<tr>
<td>1040</td>
<td>00</td>
</tr>
<tr>
<td>1048</td>
<td>A4</td>
</tr>
<tr>
<td>1050</td>
<td>91</td>
</tr>
<tr>
<td>1058</td>
<td>C1</td>
</tr>
<tr>
<td>1060</td>
<td>C2</td>
</tr>
<tr>
<td>1068</td>
<td>20</td>
</tr>
</tbody>
</table>

1000 00 BD 2A FF 00 20 B9 FE
1008 00 D0 B5 CA D0 D1 F6 0A
1010 20 B8 FE 00 D0 AB 20 B8
1018 FE 00 D0 A6 A5 28 C5 1D
1020 D0 A0 20 69 FA 00 A4 1F
1028 F0 28 A5 29 C9 9D D0 1A
1030 20 1C FB 00 90 0A 98 D0
1038 04 A5 1E 10 0A 4C ED FA
1040 00 C8 D0 FA A5 1E 10 F6
1048 A4 1F D0 03 B9 C2 00 00
1050 91 C1 88 D0 F8 A5 26 91
1058 C1 20 CA FC 00 85 C1 84
1060 C2 A9 90 20 D2 FF A0 41
1068 20 C2 F8 00 20 54 FD 00

331
<table>
<thead>
<tr>
<th>Hex</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1070</td>
<td>20 41 FA 00 20 54 FD 00</td>
</tr>
<tr>
<td>1078</td>
<td>A9 05 20 D2 FF 4C B0 FD</td>
</tr>
<tr>
<td>1080</td>
<td>00 A8 20 BF FE 00 D0 11</td>
</tr>
<tr>
<td>1088</td>
<td>98 F0 0E 86 1C A6 1D DD</td>
</tr>
<tr>
<td>1090</td>
<td>10 02 08 E8 86 1D A6 1C</td>
</tr>
<tr>
<td>1098</td>
<td>28 60 C9 30 90 03 C9 47</td>
</tr>
<tr>
<td>10A0</td>
<td>60 38 60 40 02 45 03 D0</td>
</tr>
<tr>
<td>10A8</td>
<td>08 40 09 30 22 45 33 D0</td>
</tr>
<tr>
<td>10B0</td>
<td>08 40 09 40 02 45 33 D0</td>
</tr>
<tr>
<td>10B8</td>
<td>08 40 09 40 02 45 B3 D0</td>
</tr>
<tr>
<td>10C0</td>
<td>08 40 09 00 00 22 44 33</td>
</tr>
<tr>
<td>10C8</td>
<td>D0 8C 44 00 00 11 22 44</td>
</tr>
<tr>
<td>10D0</td>
<td>33 D0 8C 44 9A 10 22 44</td>
</tr>
<tr>
<td>10D8</td>
<td>33 D0 08 40 09 10 22 44</td>
</tr>
<tr>
<td>10E0</td>
<td>33 D0 08 40 09 62 13 78</td>
</tr>
<tr>
<td>10E8</td>
<td>A9 00 00 21 81 82 00 00</td>
</tr>
<tr>
<td>10F0</td>
<td>00 00 59 4D 91 92 86 4A</td>
</tr>
<tr>
<td>10F8</td>
<td>85 9D 2C 29 2C 23 28 24</td>
</tr>
<tr>
<td>1100</td>
<td>59 00 00 58 24 24 00 00</td>
</tr>
<tr>
<td>1108</td>
<td>1C 8A 1C 23 5D 8B 1B A1</td>
</tr>
<tr>
<td>1110</td>
<td>9D 8A 1D 23 9D 8B 1D A1</td>
</tr>
<tr>
<td>1118</td>
<td>00 00 29 19 AE 69 A8 19</td>
</tr>
<tr>
<td>1120</td>
<td>23 24 53 1B 23 24 53 19</td>
</tr>
<tr>
<td>1128</td>
<td>A1 00 00 1A 5B 5B A5 69</td>
</tr>
<tr>
<td>1130</td>
<td>24 24 AE AE A8 AD 29 00</td>
</tr>
<tr>
<td>1138</td>
<td>00 7C 00 00 15 9C 6D 9C</td>
</tr>
<tr>
<td>1140</td>
<td>A5 69 29 53 84 13 34 11</td>
</tr>
<tr>
<td>1148</td>
<td>A5 69 23 A0 D8 62 5A 48</td>
</tr>
<tr>
<td>1150</td>
<td>26 62 94 88 54 44 C8 54</td>
</tr>
<tr>
<td>1158</td>
<td>68 44 E8 94 00 00 B4 08</td>
</tr>
<tr>
<td>1160</td>
<td>84 74 B4 28 6E 74 F4 CC</td>
</tr>
<tr>
<td>1168</td>
<td>4A 72 F2 A4 8A 00 00 AA</td>
</tr>
<tr>
<td>1170</td>
<td>A2 A2 74 74 74 72 44 68</td>
</tr>
<tr>
<td>1178</td>
<td>B2 32 B2 00 00 22 00 00</td>
</tr>
<tr>
<td>1180</td>
<td>1A 1A 26 26 72 72 88 C8</td>
</tr>
<tr>
<td>1188</td>
<td>C4 CA 26 48 44 44 A2 C8</td>
</tr>
</tbody>
</table>
Program 3. Supermon64 Checksum.

100 REM SUPERMON64 CHECKSUM PROGRAM
110 DATA 10170,13676,15404,14997,15136,16221,16696,12816,16228,14554
120 DATA 14677,15039,14551,15104,15522,16414,15914,8958,11945 :S=2048
130 FOR B=1 TO 19:READ X:FOR I=STOS+128:N=PEEK(I):Y=Y+N
140 NEXT I:IFY<>X THEN PRINT "ERROR IN BLOCK #"B:GOTO 160
150 PRINT "BLOCK #"B "IS CORRECT"
160 S=I:Y=0:NEXT B: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 CHRGGET differs on the various computers:

<table>
<thead>
<tr>
<th>Computer</th>
<th>Decimal Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PET (Original BASIC)</td>
<td>194-217</td>
</tr>
<tr>
<td>PET/CBM (Upgrade &amp; 4.0)</td>
<td>112-135</td>
</tr>
<tr>
<td>VIC:</td>
<td>115-138</td>
</tr>
<tr>
<td>64:</td>
<td>115-138</td>
</tr>
<tr>
<td>Apple:</td>
<td>177-200</td>
</tr>
</tbody>
</table>

The CHRGGET 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 0D  SBC #$0D
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 CHRGGET location as listed above for the different computers). Here is how the replacement looks in the example CHRGGET routine:

```
0070  4C 00 75  JMP $7500
0073  02  ???
```
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 "wedge" 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:

```
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
```

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

A or AC register (see Accumulator)
Absolute addressing 25, 40-42, 45, 46, 48, 51, 56, 68, 69, 75, 81
Absolute, X and Absolute, Y addressing 48, 51, 68, 69, 75, 81
Accumulator 19, 26, 31, 33, 39, 56, 66
Accumulator mode 51
ADC 20, 56, 58, 68, 149
Addresses 1, 2, 19, 20, 47, 54, 77, 85, 99, 124, 127, 128, 130, 139, 140,
get a character address 1
last key pressed 77
safe place address 1, 2
start of RAM 1, 99
start print address 1
which key is pressed? 1, 54, 127, 128
Addressing 18, 22, 40
Addressing modes 12, 33-34, 37-51, 68, 69, 75, 81, 149-166, 223, 224
Absolute 25, 40-42, 45, 46, 48, 51, 56, 68, 69, 75, 81
Absolute, X and Absolute, Y 48, 51, 68, 69, 75, 81
Accumulator mode 51
Immediate 25, 33, 34, 43, 51, 66, 68, 69
Implied 43-45, 55, 81
Indirect Indexed 74, 125, 141
Indirect X 51, 68, 69
Indirect Y 42, 49, 51, 57, 58, 69, 70, 74, 77, 85
Relative 25, 45-47, 69
Zero Page 33, 34, 42-43, 51, 55, 65, 68, 69, 75
Zero Page, X 48, 68, 69, 75
Zero Page, Y 51
"Alphabetic" mode 54
AND 39, 88, 89, 149
Arcade game programming in ML vi
Argument viii, 40, 55, 69, 70, 77, 81, 223, 224
ASCII code 3, 9, 53, 70, 78, 131, 144
ASL 51, 59, 68, 89, 149
ASM mode (Atari monitor) 27, 28, 110
Assembler vii, 2, 35, 45, 46, 61, 140, 223
assembler program 18
traditional conventions, list of 224
two-pass assemblers 72, 223, 225
Index

Assembler Editor (Atari) 23, 26, 28, 110, 130, 143
Assembly language vii (see machine language)
Assignment of value (see LET)
Atari monitor (see Assembler Editor; DEBUG)
Atari source code 143
ATASCII 3, 144
Attract mode 124
Auto-booting 125

B

BASIC v-vi, vii-xii, 1-4, 7, 19, et passim
advantages of xii
commands vii, 63, 121-147
ASC 144
CHR$ 144
CLR 121-22
CONT 86, 122
DATA xii, 122-23, 140
DIM 123
END 63, 124-25
FOR-NEXT 125-26
FOR-NEXT-STEP 126-27
GET 40, 93, 127-28, 131
GOSUB 81, 128-29, 141, 142
GOTO 18, 84, 85, 129-30
IF-THEN 69, 71, 131
INPUT 131-32, 133
LEFT$ 144, 145
LEN 145
LET 132-34
LIST xi, 134
LOAD 30, 92, 134-35
MID$ 145
NEW 121, 135-36
ON GOSUB 71, 136, 137
ON GOTO 69, 71, 74, 137
PRINT x, 40, 137-40
READ 140
REM 140
RETURN 5, 131, 141
RIGHT$ 145-46
RUN 141-43
SAVE 30, 110, 143
SPC 146-47
STOP 122, 124, 143
TAB 146, 147
loaders 19
Microsoft BASIC vii, 2, 4, 17, 91, 93, 105, 135, 141, 144, 224, 335
words xi
BCC 45, 59, 61, 69, 71, 74, 75, 131, 150
BCS 45, 59, 61, 69, 71, 74, 75, 150
BEQ 25, 45, 47, 59, 69, 71, 77, 131, 151
Binary numbers 7, 8, 9, 15, 243-50
program for printing table of 16
BIT 89, 151
Bits and bytes 8, 9, 10, 12-15
BMI 45, 59, 61, 68, 71, 74, 75, 151
BNE 24, 45, 59, 61, 69, 71, 72, 74, 75, 77, 131, 151
BPL 24, 45, 59, 61, 68, 71, 74, 75, 152
Branch address 47
Branches:
   ON-GOTO 74
   forward 78
Branching instructions 25, 45, 46, 47, 59, 67, 68, 69, 71-72, 73, 88
BRANCHTARGET 72, 74
Breakpoints 86, 87, 143
BRK 29, 30, 34, 37, 45, 61, 67, 86-87, 90, 122, 124, 134, 143, 152
Buffer 42, 98
BUG 28, 142
Bugs 31, 33-34
BVC 45, 63, 68, 71, 152
BVS 45, 68, 71, 152

C

CALL instruction viii, x, xi, 23, 65, 141
Carriage return 54, 144
Carry flag 37, 39, 45, 56, 58, 68, 69
Cassette buffer 1
CHRGET loop 336
CHRGET ML program 336
Circumflex 26, 223
CLC 43, 56, 58, 68, 109, 153
CLD 43, 56, 68, 153
CLI 89, 153
CLV 63, 153
CMP instruction 8, 33, 34, 61, 69, 70, 71, 75, 77, 89, 127, 128, 131, 136,
   145, 154
Code 53
"Cold start" address 124
Index

Comma, use of 79, 223-24
Commands:
   BASIC 63, 121-47
   machine language 63, 64-90, 149-66
Commodore character codes 144
Commodore Monitor Extension 26
Comparisons 70
Compiled code 92
Compilers 92-93
Conditional branch 129, 131
Control characters 273, 276
Counter variable 125
Counters 125-26
CPU (central processing unit) 8, 18, 37, 39
CPX 70, 154
CPY 70, 154
Cursor address 77
Cursor controls (PET) 30
Cursor management 77
Cursor position 140, 147

D
Data table 31, 121, 225
Debug xii
DEBUG (Atari monitor mode) 26, 28, 29, 142, 143
   commands in 28-30
Debugger 23, 36
Debugging 86
   methods 87-88
DEC 75, 155
Decimal address 19
Decimal flag 56
Decimal numbers 8, 9, 10, 14, 16, 243-50
Default 92
Delay loop 83-84, 125-26, 129
Delimiter 78, 138, 145, 146
DEX 45, 75, 84, 155
DEY 45, 75, 155
Dimensioned memory 123
Direct mode 4, 335
Disassembler viii, 20, 134
Disassembly viii, ix, 20, 140
Disassembly listings 25
Dollar sign ($) 10, 12, 28, 223
Do-nothing loop (see Delay loop)
DOS (disk operating system) xi, 135
Double compare 59

E

Echo 131
EDIT mode (Atari monitor) 27, 28, 30, 143
Effective address 85
END 4 (see Pseudo-ops)
Endless loop 33, 54, 74, 76, 124, 125, 145
EOR 39, 88, 156
Equate 72
Error messages 26, 48, 77

F

Fields 25, 27
Filename 143
FILL instruction 67
‘Fill memory’ option 135
FIND command (Atari) 110-118
Flags 8, 12, 31, 37, 39, 45, 56, 61, 66-67, 68, 69, 77, 131
  B flag 68, 86
  C or Carry flag 37, 39, 45, 56, 58, 68, 69
  D flag 68
  I flag 68, 86
  interrupt flag 89
  N or Negative flag 64, 66, 68, 69, 75, 89
  status register flags 68, 131
  V or overthrow flag 45, 68, 89
  Z or Zero flag 39, 64, 66, 67, 68, 69, 71, 75, 77, 78, 89, 130,
      131, 138, 145
  Floating point accumulator 141
  Floating point arithmetic 59, 142

G

GET# 93

H

Hexadecimal numbers viii, 1-2, 7, 8, 9, 10, 11-12, 16-17, 45, 243-50
  conventions of 12
Hex dump 19, 20, 24
Index

I

Immediate addressing 25, 33, 34, 43, 51, 66, 68, 69
Immediate mode 141
Implied addressing 43-45, 55, 81
INC 75, 156
Indirect-indexed addressing 74, 125, 141
Indirect jump 85, 224
Indirect X addressing 51, 68, 69
Indirect Y addressing 42, 49, 51, 57, 68, 69, 70, 74, 77, 85
Initialization routine 121
"Instant action"' keys 54
Instruction field (see Fields)
INT mode 25
Interactivity 34-35
Interpreter 125, 133
Interrupt request 31, 86
Interrupts:
   maskable 89
   non-maskable 90
INX 45, 55, 75, 156
INY 45, 75, 157
IRQ 31, 37, 39, 86

J

JMP instruction 18, 24, 34, 81, 82, 84-85, 91, 128, 129, 130, 157
JSR 24, 25, 45, 67, 71, 72, 80, 81, 82, 91, 92, 124, 129, 130, 136, 141, 143, 157

K

Kernal 91
Kernal jump table 91, 92, 94, 128

L

Label table 72
Languages vii, xi
   FORTH 82
      (see also BASIC; Machine language)
LDA 20, 25, 26, 29, 33, 39, 40, 43, 45, 48, 55, 61, 63, 64, 66, 69, 71, 158
LDX 51, 64, 66, 158
LDY 33, 34, 64, 66, 159
Loaders ix-x
Index

Loops 67, 75-84, 125-28, 335
  delay 83-84, 125-26, 129
  endless 33, 54, 74, 76, 124, 125, 145
  FOR-NEXT 46, 47, 75, 125-26
  indexed 48
  nested 76, 127
  timing 76
LSB (Least Significant Byte) 49, 51, 58, 70, 85, 126, 139, 141, 243-50
LSR 51, 59, 68, 89, 159

M

Machine language (ML)
  advantages of viii, xi
  equivalents of BASIC commands 121-47
  INPUT subroutine 131-32
  instruction groups 64-90
    arithmetic 39, 68-69
    debuggers 86-90
    decision-makers 69-75
    loop 75-81
    subroutine and jump 81-86
    transporters 64-68
  instructions vii, 121-47, 149-66
  monitor 253, 269-333
  strings 77-80, 144-47
  subroutines 31, 91-96
Maps 42
  Atari Memory Map 205
  Commodore 64 Memory Map 193-204
  PET/CBM 4.0 BASIC ROM Routines 175-80
  PET/CBM 4.0 BASIC. Zero Page 172-75
  PET Original and Upgrade BASIC 167-69
  Upgrade PET/CBM 169-72
  VIC Zero Page and BASIC ROMs 181-92
Masking 88-89
Mass-move 80
Memory addresses 1, 2, 20
Memory dump, 24, 28-29, 30, 275
Memory map 1 (see also Maps)
Memory mapped video 70
Memory zones 133
Message table 138
Message zone 77
Micromon 23, 31, 130, 269-333
  VIC Micromon 296-318

345
Index

Mnemonics 18, 20, 149-66
Modes:
  BASIC mode 142
  monitor mode 19, 26, 142, 143
  (see also Addressing modes)
Monitor 18, 22, 23-37
  Apple II monitor 23-26, 143
  Atari monitor 26-28 (see also ASM; Assembler Editor; Debugger)
  interactive monitors 34-35
  monitor extensions 253-334
  PET, VIC, and Commodore 64 monitor 30
  “resident” monitor 30, 253
  (see also Micromon; Supermon)
  “Move it” routine 130
MSB (Most Significant Byte) 49, 51, 58, 70, 85, 99, 126, 138, 139, 141,
  243-50
Multiple branch test 136

N
Natural numbers 7
NOP 45, 86, 169
  uses of 87-88
Number tables (hex, binary, decimal) 243-50

O
Object code 18, 22, 28, 47, 225, 226
Opcode 18, 20, 55, 66, 223, 224
Operand 55 (see Argument)
ORA 160
OS (operating system) 42

P
Page 33
  page one 42
  page six 68, 110
  page zero 33, 42, 51, 56, 57, 98, 139, 140, 336
Parameters 67, 83, 128, 141
PET ASCII 3
PHA 45, 67, 81, 82, 160
PHP 45, 67, 160
PLA 3, 4, 5, 45, 67, 82, 161
PLP 45, 67, 161
Pointers 49, 51, 57, 98, 109
   zero page 77
Pound sign (#) 25, 43, 55
Powers of a number 7-9
PRINT routines 140
Program counter 37, 39, 55, 65, 141
Programs:
   Adding the Conversion Pseudo-op 226
   Apple Version (of Search BASIC Loader) 119-20
   Atari Disassembler 240-42
   Atari Hex-Decimal Converter 11
   BASIC Loader 19
   Binary Quiz for All Computers 15-16
   CHRGET ML program 336
   Decimal to Hex, Microsoft BASIC 17
   Disassembler 237-40
   Double Compare 60
   FIND Utility for Atari BASIC 112-18
   for printing out table of binary numbers 16
   Full Assembly Listing 21
   Labelled Assembly 21
   Micromon 269-333
   Microsoft Hex-Decimal Converter 10-11
   Microsoft Table Printer 250-51
   PET Search (4.0 BASIC Version) 100-104
   Simple Assembler 227-36
      Atari Version 231-36
      VIC, PET, Apple, 64 Version 227-30
   64 Search BASIC Loader 119
   The Source Code by Itself 22
   Supermon 253-68
   Supermon64 319-33
   VIC Micromon 303-18
   VIC-20 Search BASIC Loader 120
Prompts 31
Pseudo-ops 4, 27, 28, 29, 47, 224-26
PUT#6 93

R

RAM (Random Access Memory) viii, xi, 1, 2, 4, 9, 12, 19, 31, 33, 37, 42, 80, 97, 98, 225
Reference sources 221
Registers 26, 28, 30-31, 57, 66, 70, 82
Relative addressing 25, 45-47, 69
REM statements xii, 20
"Resolving" labels 225

347
Index

ROL 51, 89, 161
ROM (Read Only Memory) xi, 1, 12, 23, 25, 26, 128, 253
ROR 51, 63, 89, 162
RTI 63, 89, 90, 162
RTS 20, 25, 45, 67, 81, 124, 129, 136, 141, 143, 162

S

Safe areas 2-3, 42, 68, 97-98, 99, 105
SBC 61, 68, 163
Screen address 139, 140
Screen position (see STA)
Search bloader 119-20
Search routine 88
SEC 43, 58, 61, 68, 163
SED 43, 56, 63, 163
SEI 89-90, 164
SGN command 63
Simple Assembler 1, 2, 3, 4, 5, 10, 17, 22, 25, 26, 27, 28, 35, 43, 47, 57, 73, 74, 79, 223-36
Single-stepping 87
6502 machine language (see Machine language)
Softkey 132
Source code 18, 19, 22, 28, 49, 72, 225
Source program 140, 225
Spaces, important 224
STA 20, 40, 49, 51, 55, 56, 57, 63, 64, 65, 67, 164
Stack 42, 67-68, 81-83, 141
Stack pointers 26, 28, 37, 39
Status Register 8, 26, 28, 31, 39, 56, 66, 68, 82
Step 26, 29-30, 31, 126-27
String handling 77-80, 144-47
Structured programming 85
STX 51, 64, 67, 164
STY 64, 67, 165
Subroutines 31, 91-96
Supermon 23, 31, 130, 253-68
Supermon64 319-333
Symbols 53
SYS instruction v, viii, x, xi, 19, 25, 30, 65, 124, 141

T

TAN command 63
Target address 130, 139, 146
TAX 64, 66, 165

348
TAY 64, 66, 165
TIM (terminal interface monitor) 142, 270, 271, 272, 275
Toggle 88-89
Trace 26, 29, 31
TRACE 87
Transfer 130
"Truth tables" 89
TSX 67, 165
Two-pass assemblers 72, 223, 225
TXA 43, 45, 55, 64, 66, 166
TXS 67, 166
TYA 39, 43, 64, 65, 66, 166

U
Unconditional branch 129, 130
Unmatched RTS 141, 143
Upward arrow 223 (see also Circumflex)
USR instruction v, vii, x, xi, 3, 4, 19, 63, 64, 67, 110, 111, 124, 141, 142

V
Variable x-xi, 132-34, 140
storing 57
Vector 86

W
"Warm start" address 124
Wedge 335-37

X
X register 46, 51, 67, 75, 125, 126
X and Y registers 26, 31, 39, 45, 48, 66, 75, 93, 94

Y
Y register 26, 34, 39, 57, 70, 147

Z
Zero address 47
Zero page 33, 42, 51, 55, 56, 57, 98, 139, 140, 336
Index

Zero page addressing 33, 34, 42-43, 51, 55, 65, 68, 69, 75
Zero page locations 49, 99
Zero page snow 68
Zero page, X addressing 48, 68, 69, 75
Zero page, Y addressing 51
Zone of variables 133, 134
If you've enjoyed the articles in this book, you'll find the same style and quality in every monthly issue of COMPUTE! Magazine. Use this form to order your subscription to COMPUTE!

For Fastest Service,
Call Our Toll-Free US Order Line
800-334-0868
In NC call 919-275-9809

COMPUTE!
P.O. Box 5406
Greensboro, NC 27403

My Computer Is:
☐ PET ☐ Apple ☐ Atari ☐ VIC ☐ Other _______ ☐ Don't yet have one...

☐ $20.00 One Year US Subscription
☐ $36.00 Two Year US Subscription
☐ $54.00 Three Year US Subscription

Subscription rates outside the US:
☐ $25.00 Canada
☐ $38.00 Europe, Australia, New Zealand/Air Delivery
☐ $48.00 Middle East, North Africa, Central America/Air Mail
☐ $68.00 Elsewhere/Air Mail
☐ $25.00 International Surface Mail (lengthy, unreliable delivery)

Name ____________________________
Address __________________________
City ______________________________ State _________ Zip _______

Country ___________________________

Payment must be in US Funds drawn on a US Bank; International Money Order, or charge card.

☐ Payment Enclosed
☐ MasterCard  ☐ VISA
☐ American Express

Acc t. No. ______________________ Expires _____ / _____

233111
If you've enjoyed the articles in this book, you'll find
the same style and quality in every monthly issue of
COMPUTE!'s Gazette for Commodore.

For Fastest Service
Call Our Toll-Free US Order Line
800-334-0868
In NC call 919-275-9809

COMPUTE!'s Gazette
P.O. Box 5406
Greensboro, NC 27403

My computer is:
☐ VIC-20  ☐ Commodore 64  ☐ Other ________________________

☐ $20 One Year US Subscription
☐ $36 Two Year US Subscription
☐ $54 Three Year US Subscription

Subscription rates outside the US:
☐ $25 Canada
☐ $45 Air Mail Delivery
☐ $25 International Surface Mail

Name ________________________________

Address ____________________________________________________________

City ___________________________ State ___________ Zip __________

Country ___________________________________________________________

Payment must be in US Funds drawn on a US Bank, International Money
Order, or charge card.

☐ Payment Enclosed  ☐ VISA
☐ MasterCard  ☐ American Express

Acct. No. ___________________________ Expires ___________ / ___________

233111
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

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Title</th>
<th>Price</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The Beginner's Guide to Buying A Personal</td>
<td>$3.95**</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Computer</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMPUTE!'s First Book of Atari</td>
<td>$12.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inside Atari DOS</td>
<td>$19.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMPUTE!'s First Book of PET/CBM</td>
<td>$12.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Programming the PET/CBM</td>
<td>$24.95***</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Every Kid's First Book of Robots and Computers</td>
<td>$4.95**</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMPUTE!'s Second Book of Atari</td>
<td>$12.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMPUTE!'s First Book of VIC</td>
<td>$12.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMPUTE!'s First Book of Atari Graphics</td>
<td>$12.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mapping the Atari</td>
<td>$14.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Home Energy Applications On Your Personal</td>
<td>$14.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Computer</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Machine Language for Beginners</td>
<td>$12.95*</td>
<td></td>
</tr>
</tbody>
</table>

* 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 pseudooops, 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.