CONTENTS

SECTION DESCRIPTIONS

6502 INSTRUCTION SUMMARY TABLES

INTRODUCTION

Your Ohio Scientific microcomputer is in many ways like the computers that write company paychecks and run factories, control space missions and predict the weather. This manual is to show how your microcomputer and its big brothers work.

It is not assumed that you know anything about computers already, just that you want to know the inside story. You will be discovering how to use your computer at a very direct level, one that permits you to control every capability that it has. Convenient access to this level is available through a part of your system known as the 65V Machine Monitor. The 65V Monitor is present in all Ohio Scientific personal computers. The first section will tell what the Monitor is. The remainder of the manual will guide you in using the Monitor to explore and control your computer. There will be twenty sections. Each section will discuss a new topic and will suggest something new for you to do with your system that will help you to understand computers.

Take your time and play around with the new elements introduced in each section before going into the next section. Enjoy yourself.

SECTION 1

COMPUTERS, PROGRAMS AND THE MONITOR

Computers are information processing devices. That is their thing! They accept input information and transforms it into output data that we can read and interpret, or into sequences of actions which we find useful or pleasing.

Not all information processing takes place in computers. One drives a car by giving it the necessary input through the ignition, steering wheel, accelerator, etc. The result is a mechanical amplification of the input information. But, a car is not a computer. There is but one way to process information through a car and that is to drive it. Social benefits aside, it is a single-purpose device.

A computer can be redirected easily to different information processing tasks. It is a general-purpose device. Switching tasks is easy because the description of a task is itself information, the natural food for a computer. Of course, the computer is a machine without intelligence, so the input must be put into a completely prescribed form, leaving nothing to be figured out on the basis of previous experience.

Computer input is of two kinds: the information to be processed and the directions for the processing. Information to be processed is called input data. Several kinds of input data will be introduced later. The directions for processing take the form of a sequence of actions called a program. The actions to complete a task may be called an algorithm. Expressing the algorithm in a form suitable for computer input, we have a program. The algorithm must be described in a language that can be "understood" by the computer, in the sense that the computer takes the right action.

Languages developed for this purpose are called computer languages. One type of computer language spells out the precise way in which each part of the computer is to participate in processing the data. This is the machine language of the computer. Since the parts of different computers are arranged differently, their machine languages are different.

When carrying out a task by executing the program which describes it, a computer is actually processing a machine language program. This does not mean that people who develop programs (programmers) always write their programs in machine language. After all, the translation of a program from another computer language into machine language is just another information processing task for the computer. When a translation is being done, the input computer language is called the source language or source code. The source language is translated by the computer into the object language. Since the objective of the translation is usually machine language, the term object language or object code is often used for machine language.

The 65V Machine Monitor is a machine language program. We will refer to it by using just the term Monitor. It does not have to be fed into your computer as input because it is built-in. Built-in programs control your computer from the time it is switched on or reset (see your operator's manual for specific directions), until the control of the computer is given to a program you have entered as input information. The program in control at start-up places a "prompting" message on the video display requesting that you select which built-in program should be executed next. One of the options in the prompting message is an "M." The program is reading the keyboard for input over and over, just waiting for you to type something.

As you depress the "M" key, control of the computer is turned over to the 65V Machine Monitor. The transfer to the Monitor program occurs before you can release the "M" key. The Monitor writes a message on the video display, to be explained in the next section, and begins to read the keyboard, searching for your next input.

Make sure the 'SHIFT LOCK' key on your keyboard is depressed. Reset the computer (this method varies between different OSI computers—consult your operator's manual). Type "M" in response to the displayed message. This starts execution of the 65V Machine Monitor.

Observe the display while depressing various keys and try to determine which keys the Monitor interprets as commands. You'll find the answer as you read on. Two commands 'G' and 'L' terminate the execution of the Monitor. Avoid them until you are finished playing hide and seek. Soon you will be entering the executing machine language programs with the aid of your humble and ever present servant, the 65V Machine Monitor.

The appendix contains a listing of the Monitor. This manual will teach you how to read and understand that listing.

SECTION 2

EXPLORING SPACE WITH THE MONITOR

Your computer is a system consisting of physical machinery controlled by stored information in the form of programs and data. The physical configuration does not change as programs execute. That part of the system is called hardware—the circuit boards, metal cases, etc. Computer programs are frequently changed and are collectively referred to as software. The hardware components of your system are represented in Figure 2.1.
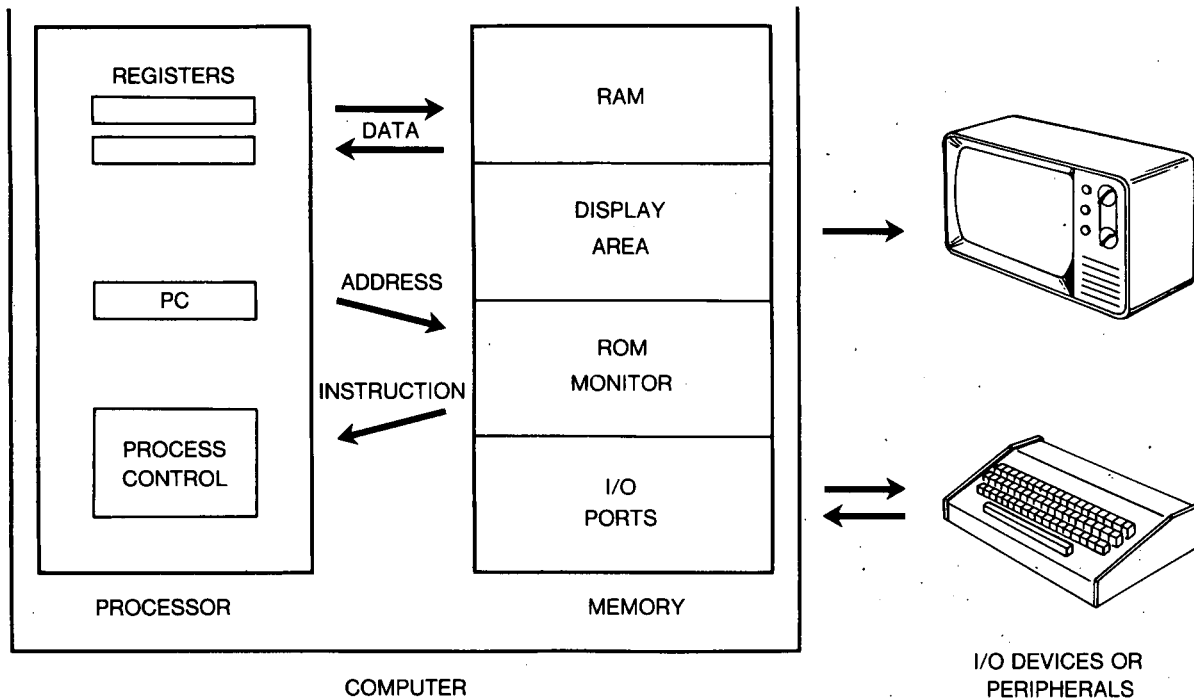
Figure 2.1 Major Hardware Components

The programs in control of the system are stored in the memory. A memory consists of a large number of cells. The contents of a cell are made available to other components by means of its address, a unique number associated only with that cell. One cell contains a piece of information we call a byte. The byte is defined in Section 4. Cells are also called memory locations.

The "action" component that processes data from the memory is the processor. In your system, the processor is a 6502 microprocessor. Information storage locations and processing stations within the processor are called registers. A single action that a processor can take with some register is called an instruction. A machine language program is a series of instructions. When one of the processor's registers, the program counter (PC), is set to the address of the beginning cell of a program, the processor begins to read and execute the machine language instruction contained in those cells. The resulting execution of the program causes some memory cells to be altered and exchanges of information to take place between the memory and input/output (I/O) devices. The result is the activity of the computer system that you can observe, from a loan calculation to video animation.

Memory cells are not all alike. Some portions of memory hold fixed information that cannot be altered as programs are executed or when the computer is turned off. This portion is called Read Only Memory (ROM). The 65V Machine Monitor and other built-in programs are located in ROM. Other parts of memory are for temporary information, like the data being processed by executing programs, or programs which are brought into memory to produce a desired behavior at a particular time. This kind of memory is known as Random Access Memory (RAM) or read/write memory.

As you start the execution of the 65V Machine Monitor, the processor clears the CRT screen (video display) and writes a four-digit number and a two-digit on a small portion of the display. This is done by transferring characters

into the RAM display area as shown in Figure 2.1. Contents of this RAM display area are regularly transmitted to the CRT display screen.

The characters appearing on the screen represent two numbers written in a number system called hexadecimal or base sixteen. In a section to follow you will learn how these numbers are constructed. The left number is the address (location) of a memory cell and the right number shows the information contained in that cell. Two hex digits are enough to express the contents of any 8-bit) memory cell. Addresses or memory cell numbers are two bytes long and are represented by four hex digits or represented by four hexadecimal digits. For reasons we can appreciate shortly, the Monitor ignores most keys reacting only to the decimal digits 0-9, the letters A-G and L, the special symbols: "/" (slash), '.' (period), and the 'RETURN' key.

The decimal digits 0-9, along with the letters A-F are hexadecimal digits. The Monitor has two ways to react to keyed in hexadecimal digits. In the Monitor address mode, digits are rolled into the address value (left or four-digit number) displayed. This changes the right value too, since the new address designates a different memory cell. In the Monitor data mode, digits are rolled into the data value (right or two—digit number). Actually, the addressed memory cell is being altered and the display change reflects the alteration.

The Monitor starts in the address mode. Depressing '/' changes it to data mode. When '.' is depressed, the Monitor returns to address mode. The 'RETURN' key advances the address by one, changing the data value. You can use it to move through memory, reading or writing into cells.

The 65V Machine Monitor program starts at a location whose address is $FE00 (read F, E, zero, zero—the $ indicates that this is a hex value). See if you can get that number (without the $) into the address side of the display. This displays the first instruction of the Monitor on the data side. It should be 'A2'. Now depress '/' and the 'RETURN' key several times. What is happening? You are counting up on the address side and getting a display of the contents of successive memory cells on the data side. Wherever you stop, write down the address and write '22' into that cell using the data mode. Now return to address mode ('.') and key-in the address over itself. Disappointed? The '22' is not there because you are addressing ROM, where the contents of cells cannot be changed.

Can you enter data and see the result immediately? Yes, in the display area. Each character position of the CRT screen has a RAM cell associated with it which you can alter in data mode. Most of them now contain the 'blank' character. Given the address of a cell in the display area, (see the user's manual for specific video display locations for your computer) you can place non-blank characters in nearby cells to create a pattern on the screen. When you learn how to count in hexadecimal, you can systematically explore the display area and find out exactly where screen positions are recorded in the display memory. A starting location for some preliminary exploration is the hexadecimal number $D140, a strange looking number if I ever saw one.

SECTION 3

CREATING A GOOD LISTENER

It may be that good listeners are born, not made. But your computer can be programmed to pay strict attention to every character you type. It cannot nod its head in agreement but it can repeat or echo each character on the television screen.

Let's think about the program which describes the "good listener" task. In order to write a program we have to specify the desired behavior completely, leaving nothing unspecified. The computer understands nothing and must be told everything. For starters, exactly where on the screen do we want the input to be echoed? Where the Monitor prompt message was? Okay. When our program is executing, the Monitor will be asleep, having no processor to execute it, so these display locations are available for display. Let's extend the display across the screen so we can read the glorious stuff we shall be entering. What happens when we reach the right of the screen? Later you can implement other choices, but for now, let's start pushing characters to the left making room for new input to the right, just as the Monitor does to its address and contents display. Will there be any keys which will be used to cause the "good listener" to alter its behavior, like the '.' and '/' commands affect the Monitor? No, not in the 'first pass' version. How will the good listening end? We can't tell it to stop so let it go on forever! "Forever" means as long as the program has control of the processor. The best way to see what the program does is to try it. The sequence of steps in the flowchart (See Figure 3.1) must be expressed in computer language. This programming step is called coding. Coding in machine language requires the programmer to express the required processing in terms of operations which the processor can do with its registers. Using techniques you will be learning in this manual, a machine language version of the "good listener" was created in hexadecimal numbers, so that you can load it from the keyboard. It should be loaded starting at location 0000, as shown. All key-in's for loading, using the Monitor, are shown below. We use the symbol '*' to represent the 'RETURN' key. Some locations are shown to the left for checking during loading.

LOCATION (HEX)	WHAT YOU SHOULD KEY-IN
0000	.0000/A2*00*20*ED*FE*9D*46*D1*
0008	E8*E0*14*D0*F5*20*ED*FE*
0010	A8*A2*00*BD*47*D1*9D*46*
0018	D1*E8*E0*13*D0*F5*98*8D*
0020	59*D1*4C*0D*00

Before executing the program let's check the loading. One key-in error can make a big difference. How do we check it? Type ".0000/"—this resets the Monitor to the data mode and resets the display to the beginning of the program ($0000). Repeated use of the 'RETURN'' key allows you to "step" through the program examining each location. If you find a mistake, enter the correction.

Computers do one thing at a time, so a program must be organized that way. An important tool for planning or showing the sequence of steps in a program is a flowchart, a diagram such as the one to the right. Arrows in a flowchart mark the path of the processor, boxes represent the processing that it does along the way. Box shapes have meaning to programmers. Rectangular boxes stand for processing steps. Slanted sides signal an input or an output. Diamonds are decision boxes, representing tests which select paths for the processor. In the program a decision step is called a branch. Branches allow sets of steps to be repeated and decisions to be made. A set of repeated steps is called a loop. The six-sided box near the top denotes preparation for a loop. Ovals show the start and termination points of the program.

You probably have realized that the flowchart represents the "good listener." There are two loops, one to fill the display line and another to add characters to a full line. The lower loop has no decision box, therefore, there is no way for it to end. There is usually one start point and there can be several termination points.
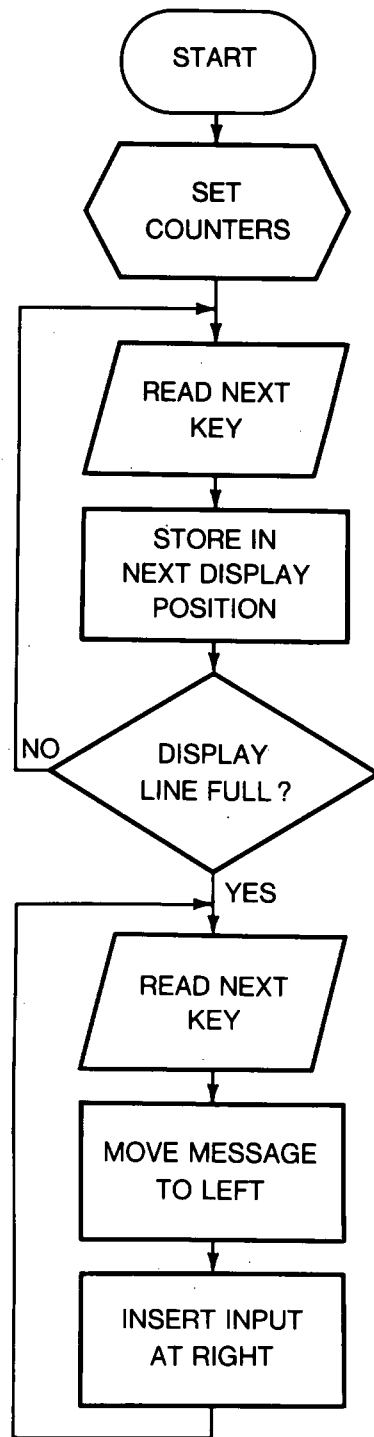
Figure 3.1 A Flowchart

For example, suppose you type ".0000/" and depress the 'RETURN' key three times. Suppose the display reads "0003 DD," this is wrong, it should be "0003 ED." To enter the correction, simply type the correct contents, E followed by D. Note that the display now reads "0003 ED" which is correct. Your correction is now entered. Continue with the 'RETURN' key to check the code. Repeat the above process as necessary until the code on Page 5 is entirely correct.

Now that the program is loaded and checked, it can be executed. The 'G' command tells the Monitor to load the displayed location into the program counter, thus, starting the execution of the program beginning at the displayed cell. So, .0000G will start the "good listener." Is it listening?

6

SECTION 4

THE MIGHTY BIT

Information comes into the computer system through its input/output (I/O) devices and is stored in memory. It moves around inside the processor. Output information appears on the display as numbers, words or pictures. But if we were inside the computer watching the information go by, we would see it in a strange internal form.

A computer is made of electrical circuits and connecting paths which hold, transmit or process the smallest particle of computer information there is, the bit. A bit is a simple yes or no, a true or false, one of two possible values. Internally, this is seen as a one or a zero. It is the universal measure of the amount of information content in any message. A set of bits taken together is a binary code. Just as a bicycle has two wheels, a binary code is based on two values. Like the parallel rails of a railroad track, sets of electrical paths carry binary codes around inside the computer.

Anything that can be said, can be said in a binary code, provided there are enough bits and the bit values have been assigned a meaning. The bits themselves do not carry the meaning. Sitting there in the computer, we would not get any message by reading a passing binary code unless we knew the meaning assigned to the bit values by the sender and receiver of the code.

To represent a binary code on paper, we write it as a string of characters in which each bit is represented by one of two possible characters. Bit values are traditionally represented by the symbols '0' (zero) and '1' (one).

A byte is a binary code composed of eight bits. In your computer each memory cell contains one byte. A character, such as a letter, digit, or punctuation mark, is the type of information which is (forgive the pun) byte size. When reading the keyboard, the processor loads a byte from the keyboard into one of its registers. This byte indicates which key is being depressed. "Good listener" stores each byte it obtaines in memory cells in the display area. A byte in the display area directs the display subsystem to a pattern for the character which is to be seen at that position.

Somebody had to decide what bit pattern should represent each character. Many computer manufacturers use the same code, so that computers can transmit character information to each other. The common code used in almost all small computers is the American Standard Code for Information Interchange or ASCII code.

You can look up the ASCII code for any keyboard character in the table in the appendix at the rear of this manual. We have another way to determine the ASCII value—we can execute a program which reads the keyboard, converts and displays the bit values it is receiving. The load and execute Monitor commands for this conversion program are:

```
.0000/20*ED*FE*85*F0*A9*18*A2*

07*9D*D2*D0*66*F0*3E*D2*

D0*CA*10*F5*30*EA

.0000G
```

As you examine ASCII codes, go through the sequence of digit characters 0,1, . . . ,9 and note what you see.

If it is convenient to leave your computer on while reading the next section, it will save time. The next section suggests a small change in the binary display program, which can be made without keying in the program again, provided the computer has not been turned off or used for any other purposes.

SECTION 5

THE UGLY TRUTH ABOUT BINARY

Numbers are important to computers. Addresses of memory cells are numbers. The contents of registers and memory cells often represent numbers. Computers spend a lot of their working time counting and calculating with numbers. So, numbers have to be represented in binary codes, don't they? One way of doing it could be seen in the ASCII codes for digits Ø-9. Ignoring the left four bits which remain fixed at ØØ11, the digits are represented this way:

Ø—ØØØØ   5—Ø1Ø1

1—ØØØ1   6—Ø11Ø

2—ØØ1Ø   7—Ø111

3—ØØ11   8—1ØØØ

4—Ø1ØØ   9—1ØØ1

To represent a number like 65Ø2, we can string these codes together, packing two-digit codes to a byte like this:

65Ø2 = Ø11ØØ1Ø1 ØØØØØØ1Ø
        6   5   Ø   2

This is a frequently used code for numbers. It is called Binary Coded Decimal or BCD. Can you write the four-bit codes that are not used in BCD for digits? There are six of them. (The unused codes are: 1Ø1Ø, 1Ø11, 11ØØ, 11Ø1, 111Ø and 1111). These unused codes are the reason why BCD is not the most efficient way to represent numbers in a computer. A memory cell containing a BCD byte is storing only 5/8 of the information it could, because its bits are not free to express all of their values. Also, the processor circuits which carry out such pleasing operations as addition and subtraction are complicated by the need to avoid the codes that do not represent digits.

So what is a "natural" number system for computers? One that will allow them to do their thing? We humans are partial to tens. We write the digits 65Ø2 to mean:

$$(6 \times 1ØØØ) + (5 \times 1ØØ) + (Ø \times 1Ø) + (2 \times 1)$$

$$\text{or } (6 \times 1Ø^3) + (5 \times 1Ø^2) + (Ø \times 1Ø^1) + (2 \times 1Ø^Ø)$$

The decimal digits Ø,1, . . . ,9 express numbers in the decimal number system based on the number ten. What's so great about ten? Take off your mittens.

If you are a computer, how many fingers do you have? Two. The characters 'Ø' and '1' stand for your digits and the natural way for you to express the above number is 11ØØ11ØØ1Ø11Ø, meaning:

$$(1 \times 2^{12}) + (1 \times 2^{11}) + (Ø \times 2^{1Ø}) + (Ø \times 2^9) +$$

$$(1 \times 2^8) + (1 \times 2^7) + (Ø \times 2^6) + (Ø \times 2^5) +$$

$$(1 \times 2^4) + (Ø \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (Ø \times 2^Ø)$$

This is the binary number system based on the number two. The BCD codes for numbers up to ten are the binary representations of these numbers.

In the binary number system every bit pattern is used and stands for a unique number, but only a computer could love a number like 11ØØ11ØØ1Ø11Ø. The ugly truth about binary is that it scrambles human brain waves. We lose our place in its maze of zeros and ones. The human brain deals best with a few things at a time. We even put commas in long decimal numbers to mark the place. What we need to deal with binary is something to group those bits into a shorter, more readable pattern.

The answer lies in those six missing four-bit codes in BCD. If we had digits to represent those extra codes, we could write binary compactly because every four-bit group could be replaced by the corresponding digit. A digit is

some character given a numerical meaning. To get six more, we use the letters A-F and assign numbers and their four-bit binary representations to them, arriving at:

0—0000	4—0100	8—1000	C—1100
1—0001	5—0101	9—1001	D—1101
2—0010	6—0110	A—1010	E—1110
3—0011	7—0111	B—1011	F—1111

The number system based on sixteen is called hexadecimal. If our explorations of the universe reveal intelligent life forms with sixteen fingers, they probably will be using hexadecimal.

To see how that ugly binary number 1100110010110 shapes up in its hexadecimal form, just mark off groups of four bits from the right 1/1001/1001/0110 and replace each group by its hexadecimal digit. You should get 1996_{16}.

Another system that is sometimes used to represent binary in computer literature is the system based on eight. It is called octal. In octal, the digits are zero through seven and each digit replaces three bits.

With all these number systems floating around we could get confused. When there is any question about it, the base number of the system is written as a subscript of the number representation. For example:

$$6502_{10} = 1100110010110_2 = 1966_{16}$$

How would our 16_{10}—fingered friends write that? Answer:

$$6502_A = 1100110010110_2 = 1966_{10}$$

Can you figure out why?

You can enlist your computer's help in a little binary to hexadecimal drill by making a small modification in the binary display program of the last section. The program loops forever, always looking for a new key-in. Let's make it return to the Monitor after accepting one key-in. Then you can use the Monitor to place in a display cell what you think is the hexadecimal version of the binary code that you see. If you are right, the character originally keyed in will appear on the screen.

If you haven't turned your computer off since entering the binary display program, it is still there in RAM! Going to the Monitor, alter the program by entering:

.0014/4C*2A*FE*

Now when you enter .0000G, the modified program will display in binary the ASCII code for your next key-in. Select a nearby display location and enter the hexadecimal code. If you make a mistake on the binary to hexadecimal conversion, something other than the character you expected will appear, maybe something you cannot key-in. Does that make you curious? Then why not make some "intentional" mistakes? Who'll know?

SECTION 6

HEXABURGER HELPER

So why does an alien being with eight fingers to the hand (or four fingers on each of four hands) write

$$6502_A = 1100110010110_2 = 1966_{10}$$

with his little alien BIC? Because to him 10 is the one-digit number A. And to him 10 means sixteen. Of course, he would never call it "sixteen." That't decimal-based terminology. In any number system, 10 stands for the number on which the system is based. This number is referred to as the base or the radix of the system. Remember that there is just one set of integer numbers but many systems with which to express them. The equal signs in (back to earth)

$$6502_{10} = 1100110010110_2 = 1966_{16}$$

belong there because we have three representations of the same number.

To find the decimal representation of a hexadecimal number, say, $D1E4_{16}$, you translate its digits to decimal and carry out the decimal calculation it stands for, namely

$$(13 \times 16^3) + (1 \times 16^2) + (14 \times 16^1) + 4 = 53732$$

The calculation is a bit easier if you alternate multiplications and additions like this:

$$((13 \times 16 + 1) \times 16 + 14) \times 16 + 4 = 53732$$

Written out, the calculations look like this:

```
     13        208        209       3344        3358       53728
   × 16      +   1      ×  16      +  14       ×   16      +    4
     78        209       1254       3358       20148       53732
     13                   209                   3358
    208                  3344                  53728
```

The result: $D1E4_{16} = 53732_{10}$.

As you work with machine language programs, you will want to add two hexadecimal numbers and get a hexadecimal result. On way is to convert the numbers to decimal representation, add them, then convert the sum back to hexadecimal. That's the hard way. The easy way is to add in hexadecimal, using the same rules of arithmetic we use in decimal addition. Sounds hard, doesn't it? Well, judge for yourself. First, let's take a close look at what's happening in the decimal addition:

```
   1
  182
 +137
  319
```

Since $8 + 3$ cannot be represented in one decimal digit, you think of it as $10 + 1$ and write the 1, noting the carry into the column to the left, as shown.

Hexadecimal addition works on the same way. Write the digits of $B6_{16} + 89_{16}$ in the format for column-by-column addition, then add the units column digits. The answer, 15_{10}, is represented as F in hexadecimal. In the sixteen's column, we get $11 + 8 = 19_{10}$, which cannot be represented in one hexadecimal digit. So, we think of it as $16 + 3$, write down the 3, and carry the 1 from $16 = 10_{16}$. The whole problem then reads:

```
   B6₁₆
 +  89₁₆
   13F₁₆
```

Hexadecimal subtraction can be done directly, following the same rules as decimal subtraction: In

$$13F_{16}$$
$$-\ B6_{16}$$
$$89_{16}$$

we must borrow 1 to go with 3. But, the borrowed 1 stands for 16_{10}, so we calculate $(16 + 3) - 11 = 8$.

Hexadecimal addition and subtraction are a help in finding out exactly where display positions appear on your screen. The number of display positions in a row is a multiple of 16_{10}. It may be $32_{10} = 20_{16}$ or $64_{10} = 40_{16}$. This means that locations displayed in a vertical column differ by that multiple.

SECTION 7

MAKING UP ORDERS FROM HEADQUARTERS

Remember the flowchart for "good listener?" It showed the sequence of steps for doing the desired processing and display task. Such a sequence of steps is called an algorithm, as we mentioned earlier. To bring the "good listening" algorithm to life in your computer, you loaded a machine language program into memory as a sequence of binary codes, keying in each byte of the program as a pair of hexadecimal digits. How did "good listener" get into that form? By a process that will be described in this section, a process involving two stages called coding and assembly.

In the coding stage, the programmer expresses his chosen algorithm as a series of actions that the computer's processor can perform. Each individual action is an instruction. A processor has a menu of available instructions called the instruction set. Working from a flowchart or some other description of the algorithm, the programmer chooses sequences of instructions. These include instructions which select sequences to be executed next and other instructions which cause sequences to be executed repeatedly.

The structure of the microprocessor determines what instructions are available in the instruction set. Coding in machine language requires a knowledge of that structure because you are expressing the algorithm directly in machine instructions.

Translation of algorithms into more general computer languages (also called higher level languages) is also called coding. This approach does not require the programmer to know the "insides" of the processor. This knowledge is applied in a program called a compiler or an interpreter that translates the higher level computer language into machine language instructions. Some high level languages that are translated into machine language are BASIC, FORTRAN, PASCAL, APL and COBOL.

We are going to be coding in the machine language of the 6502 microprocessor. The 6502 has three registers for which programers use the names X, Y and A. Each of these registers holds one byte of information. The accumulator, A, is the busiest register by virtue of its direct connections with the processing circuitry in the 6502. The X-register and Y-register are called indexing registers. Their usual jobs are counting and maintaining the location (or index) of data being processed.

A translation of the "good listener" flowchart into operations on the contents of 6502 microprocessor registers is given in Figure 7.1. To the left of each box is a description of a way to use the 6502 registers to carry out that step. The programmer may have a choice of several possible ways to use the registers. Of course, Figure 7.1 does not represent the completed coding because the instructions are not specified. Besides spelling out the instructions, the coding step also involves specifying the location and arrangement of data. In the "good listener" coding, for example, access to display locations is done by incrementing the X-register (register X) by one each time a loop is repeated. This approach dictates the arrangement of that data.

Load a zero into Register X.

Use a part of the Monitor which loads the ASCII code for the next key depressed into the accumulator A.

Store from the accumulator A into the memory cell whose address is $D146_{16}$ plus the contents of X.

Increment X by 1 for next store.

Compare X with length of display line. If not yet equal, branch back for next key.

Use the same routine to load the ASCII code for next key depressed into the accumulator.

Load a zero into Register X. Repeat the following loop across the display: load from the contents of $D147_{16}$ plus X into Y, store from Y into $D146_{16}$ plus X. Increment and test X.

Store from the accumulator into the address on the right of the display line. Jump back to the instruction for reading a key.
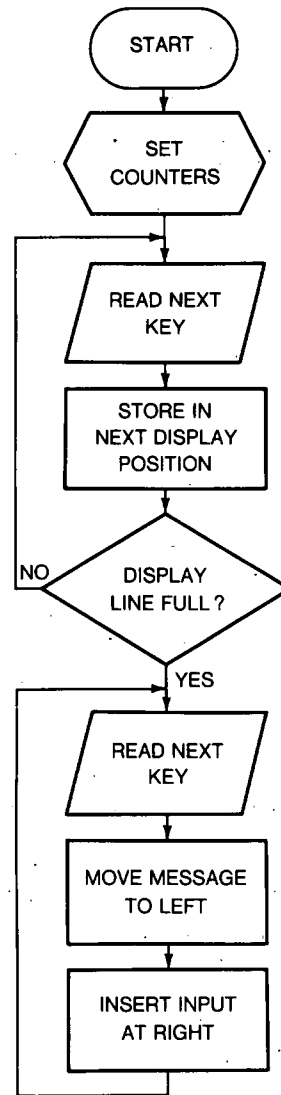
Figure 7.1 "Good Listener" as Register Operations

In the coding stage, some flowchart boxes may be implemented by one or two instructions. Others may take many instructions. In describing the "good listener" algorithm, it is helpful to use a single box for the "move message left" step. Implementing this step with register operations requires a loop, as shown in Figure 7.2. Loops often involve incrementing the testing an index, such as in Figure 7.2. The index must be given an initial value before the loop is executed the first time.
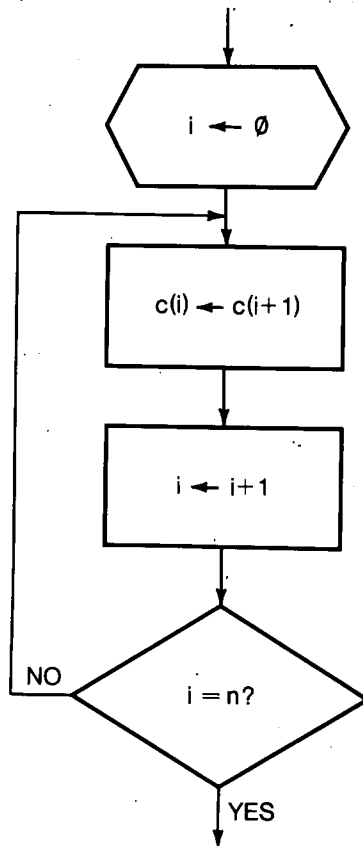
Figure 7.2 A loop for "move message left"

Now getting down to selecting instructions, just what does an instruction contain? An instruction always contains a binary code called an operation code, or opcode, for short. The opcode identifies the operation to take place and the manner in which the operands, the data involved in the operation, are to be accessed. Operands are contained in either microprocessor registers or in memory cells. In the case of memory cells, the instruction selects one of several addressing modes for the operand. Addressing modes are ways of forming the cell address of an operand in memory. In addition to the opcode, instructions often contain the operand address or information that goes into the formation of the address. The operand address derived during the execution of the instruction is called the effective address.

Figure 7.3 shows a coding of "good listener." The symbolic form of the opcode that most programmers use in coding is under the heading Mnemonic. The neumeric form of the same opcode that results from the process of assembly is under the heading Opcode.

LOCATION	OPCODE	OPERAND	LABEL	MNEMONIC	OPERAND		REMARK
0000	A2	00		LDX	#0	;	CLEAR INDEX
0002	20	ED FE	FILL	JSR	GETKEY	;	NEXT PRESSED TO A
0005	9D	46 D1		STA	$D146	;	INTO NEXT LINE CELL
0008	E8			INX		;	INCREMENT INDEX BY 1
0009	E0	14		CPX	#20	;	END OF THE LINE?
000B	D0	F5		BNE	FILL	;	BACK UNTIL EQUAL
000D	20	ED FE	REPEAT	JSR	$FEED	;	NEXT PRESSED TO A
0010	A8			TAY		;	SAVE KEY IN Y
0011	A2	00		LDX	#0	;	CLEAR INDEX
0013	BD	47 D1	MOVE	LDA	$D147	;	LOAD LINE(I+1)
0016	9D	46 D1		STA	$D146	;	STORE INTO LINE (I)
0019	E8			INX		;	INCREMENT INDEX
001A	E0	13		CPX	#19	;	END OF THE LINE?
001C	D0	F5		BNE	MOVE	;	BACK UNTIL EQUAL
001E	98			TYA		;	RESTORE KEY-IN
001F	8D	59 D1		STA	$D159	;	STORE NEW KEY
0022	4C	0D 00		JMP	REPEAT	;	BACK FOR MORE

Figure 7.3 Good Listener as Machine Language Instructions

Compare columns 1-3 with the contents of memory (starting at $0000) resulting from the program entry exercise on page 5. Notice the pattern of correspondence between the assembly listing here and the machine readable form of the program given on page 5.

Just as binary numbers are hard to interpret, machine language programs in binary form are not easy to deal with. Since programmers want to be able to read and alter programs as well as create them, they usually write them in a symbolic form using names for operations, registers and memory locations. There is also the important matter of getting the new program to work in the first place. This process is called "debugging." There always seem to be some little mistakes or "bugs" in a new program, and finding them requires concentrated study of the program. So a readable form of the program is essential.

Assembly is the process of translating the symbolic form of machine language into binary codes. If the symbolic form of the program is written according to required format rules, a program can do the assembly task. Such a program is called an assembler. The input to the assembler is called assembly language. The symbolic form of Figure 7.3 and other such figures in this manual conform to the standard assembly language mnemonics (symbols) for the 6502 (see list in appendix). (For more information concerning assembly language programming on OSI computers refer to the OSI Assembler/Editor and Extended Monitor Reference Manual.)

In later sections we shall return to Figure 7.3 to learn about every part of it. For the moment, see if you can tell where to change the length of the display line. Compare the assembly language version with the flowchart and description of Figure 7.1. Every instruction is on a separate line and the assembled instruction begins with the opcode. In the assembly language, the numbers relating to display line length are in decimal; the corresponding numbers are in hexadecimal in the assembled code. Can you find them? Now select another line length that will fit on your screen, alter the program and try it. A clue: there are three bytes in the program that must be altered.

SECTION 8

MAKING YOUR MOVE

How is coding a program like sorting potatoes? Answer: It's just one decision after another. Would you like to understand the process and do some coding yourself? Okay, let's examine some of the decision-making that takes place in coding. Mostly, the "good listener" algorithm involves moving data around. For now, we'll concentrate on decisions connected with that.

The part of the 6502 instruction set devoted to moving data around between memory and processor registers X, Y and A appears in Figure 8.1. There you have almost all you need to know about these instructions for coding and assembly. Missing is the time it takes the processor to execute each instruction. We are seldom concerned with that. Many manuals and books provide 6502 instruction tables with execution timing. See the Bibliography for references.

Starting with the leftmost column of Figure 8.1, the operation mnemonic is a name for the operation, a name which is supposed to be easy to remember. Compare the mnemonics to the explanations of the operations. Do mnemonics help you to remember? Next we have hexadecimal opcodes for each of the addressing modes available with that operation. The opcode determines both the operation and the addressing mode. There are many ways to address memory cells with the 6502 processor. Frequently used operations are assigned most of the available addressing modes, so that you will have ways of expressing algorithms effectively, in reasonably small numbers of instructions. Not all combinations of operation and addressing mode are available in the instruction set.

OPCODES FOR VARIOUS MODES

ADDRESS MODES

MNEMONIC	EXPLANATION	immediate	absolute	zero page	(ind,X)	(ind),Y	zero page,X	abs,X	abs,Y	zero page,Y	FLAGS
LDA	Load A	A9	AD	A5	A1	B1	B5	BD	B9		NZ
STA	Store A		8D	85	81	91	95	9D	99		
LDX	Load X	A2	AE	A6					BE	B6	NZ
STX	Store X		8E	86						96	
LDY	Load Y	A0	AC	A4			B4	BC			NZ
STY	Store Y		8C	84			94				

MNEMONIC	EXPLANATION	MODE IS IMPLIED	FLAGS
TAX	Transfer A to X	AA	NZ
TAY	Transfer A to Y	A8	NZ
TXA	Transfer X to A	8A	NZ
TYA	Transfer Y to A	98	NZ

Figure 8.1 Moving Data between Memory, X, Y and A

At the right of Figure 8.1 there is an indication of any flags affected by the operation. This is included here for the sake of completeness. Flags are one-bit registers in the processor whose functions we will go into later.

Most of the addressing modes in Figure 8.1 will be explained in later sections, as we examine different kinds of information processing. Right now, let's review the "moving data" coding decisions in the "good listener" program, Figure 7.3.

In the first instruction, index register X is cleared to zero by loading a zero valued byte from memory. The opcode identifies the addressing mode as immediate mode (A2 in the LDX row, Figure 8.1). In this mode, the operand is the byte in memory following the opcode. It is known as an immediate operand. Note from Figure 8.1 that registers A and Y can also be loaded with immediate operands. There are no immediate mode store-in operations, so immediate bytes must be considered to be constants. Memory locations in which a program stores data can represent variables in the algorithm, quantities that can change during its execution. In the assembler input, an immediate operand is identified by the character '#' preceding the constant. Identify the other immediate operands in the program. A dollar sign indicates the following digits are hexadecimal, not decimal digits.

We must defer explaining the second instruction, the JSR, but its effect here is to place the ASCII code for the next key depressed into the accumulator A.

The natural thing to do is to store from A into the display line. The addressing mode of the STA instruction is absolute, indexed by X. Absolute mode means that a full 16-bit address follows the opcode. For reasons we will cover later, the absolute address bytes are reversed in the instruction. Do you see what we mean? The address intended in the program is the start of the display line, which is D146. The leftmost digits, D and 1, represent the high order byte of the address. This byte is called the high order or high byte because the bits stand for higher powers of two than the low byte (46) bits. As a 16-bit binary number, the address is written D146, with the high byte on the left and the low byte on the right. In an absolute addressing mode instruction, the low byte comes before the high byte in memory. When listing the contents of memory, left to right corresponds to increasing addresses in memory. Thus, the high and low bytes of an absolute address are reversed in the machine language program.

Would you like to change the address of the display line in "good listener?" Then you must alter the program in four places, the absolute addresses with low bytes at hexadecimal 6, 14, 17 and 20.

In the assembly language operands, the characters ',X' denote indexing by X, which means that the effective address is the sum of the absolute address and the contents of register X. What happens to the effective address as X is incremented (by one) with each repeat of the loop? This is a very important question. The answer reveals how "good listener" works.

There is an instruction in the program using an absolute addressing mode without indexing. It is the JMP instruction at 22_{16}. Guess what JMP stands for? In assembly language, names like REPEAT, FILL and MOVE are created by the programmer to stand for addresses of instruction or data locations. Such names are called labels. Each label appears to the left of the opcode mnemonic and in the symbolic form of instructions in which the address affects the assembled code. Labels may be used whenever they are needed.

Continuing with the coding decisions of "good listener," there is a problem in the second loop with having the key code in register A. We would like to use the sequence

```
MOVE   LDY   $D147,X

       STY   $D146,X
```

to accomplish the left face, forward march of the display. The problem is that STY has no absolute indexed addressing mode. No such opcode in Figure 8.1, right? If we used register Y instead of register X as the index register, would we have the same problem? You bet. To get around the problem, the program saves the key code in the Y register and uses the accumulator A for the leftward bunny hop. Could this little inelegance be avoided by bunny hopping first, then getting the new key? The answer is no. Figure out why, then test your theory by arranging the program that way and observing what happens.

Do you get the impression that coding can be fun? Let's see if you can reverse the roles of X and Y in "good listener" and get a working program. To compare contents of Y with an immediate line length, the instruction reads

```
CPY   #20     ;END OF THE LINE?
```

and the opcode is C0.

SECTION 9

FETCH AND STEP IT

Many of the 6502 instructions are easier to understand if you know how a computer's processor goes from instruction to instruction during program execution. The processor keeps track of its location in the program by means of a register caled the program counter (PC). The program counter is the size of an effective memory address. After the execution of an instruction, it holds the address of the next instruction to be executed.

Each instruction is processed as it is encountered, in a two-part cycle, the fetch-and-execute cycle. In the first part, the program counter contents are sent to the memory unit which sends the opcode back to the processor regis-

```
+-----------------------------+         +-----------------------------+
|                             |         |                             |
|   PC  [    .    |        ]  | ---- ADDRESS ---->                   |
|                             |         |                             |
|                             |         |        [          ]         |
|                             |         |                             |
|   IR  [          ]          | <---- OPCODE ----                     |
|                             |         |                             |
+-----------------------------+         +-----------------------------+
         PROCESSOR                                  MEMORY
```
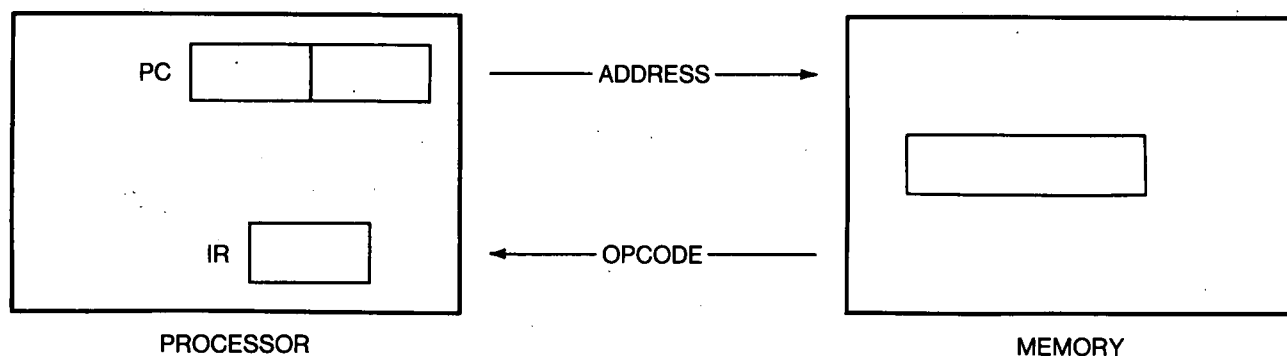
ter called the instruction register (IR). The program counter is incremented by one. This is the fetch cycle. Now the processor determines the operation and address mode and carries out the execution cycle. In the execution of the instruction the program counter advances to the next opcode. It may be further affected by the execution of some instructions, as we shall see.

It is the program counter that provides access to the immediate operands described in the last section. It contains the effective address at the beginning of the execution cycle. It is efficient to access constants this way, but not variables. For a variable, some additional mechanism would be necessary to get back to the stored value, once the program counter had gone past it. Besides, programs in ROM would have to avoid immediate "stores" anyway. Remember why? For these reasons, the immediate addressing mode is for constants only.

In absolute and absolute indexed addressing modes, the program counter is used to fetch a full two-byte address. Because the address bytes are stored in reverse order, the low byte is fetched first. This is handy because in indexing the processor must add a register byte to the absolute address to form the effective address. It gets the low byte first and adds the index register contents to it. The sum can be too large for a byte but there is no real problem. With your knowledge of binary, you can confirm that at worst, only one extra bit is needed. The extra bit is called, appropriately enough, the carry bit. The processor retains the carry bit, advances the program counter and fetches the high address byte, then adds the carry to it, forming the full effective address.

Branch or jump instructions provide an effective address which can replace the contents of the program counter. The replacement causes the processor to start on a sequence of instructions at another location in memory.

With the 65V Machine Monitor, you start the execution of a program by entering its start location into the address display and depressing the 'G' key. This transfers the displayed address into the program counter. From that point the processor is at the mercy of the program you keyed in. Program bugs or loading errors may bring the program counter to the address of an invalid or unintended opcode. When that happens, the program loses control of the processor and, if anything occurs next, it isn't what you had in mind. Fortunately, the reset key forces the program counter to the starting address of a program that will listen to you, so that you can get back to the Monitor and look for the problem.

With the program of Figure 9.1, you can execute a series of instructions one at a time and observe their effects on memory and the processor registers. The program contains a "box" of three "no operation" (mnemonic is NOP, pronounced no op, like co-op) opcodes in its middle. A NOP instruction does nothing but advance the program counter by one. To execute one instruction, you insert it in the box where the program counter will reach it as the

program executes. Any instruction will fit since none are longer than three bytes, but be sure that the unused box bytes are filled with the EA opcode of a NOP instruction.

The instructions preceding the box load the processor registers and flags from memory locations $00F0 through $00F3. After the execution of the inserted instruction, the contents of all processor registers are stored in the corresponding memory locations and the program returns to the address mode of the Monitor. At this point you can examine memory to verify the effects of the executed instruction. Where do you look for the contents of X, Y and A? (Answer=$F1–$F3)

```
                        ;RESTORE PROCESSOR TO EXECUTE BOX INSTRUCTION

00D0    A5 F0           LDA SAVE        ; LOAD FLAGS AND

00D2    48              PHA             ; PUSH THEM ONTO THE STACK

00D3    A6 F1           LDX SAVE+1      ; LOAD X

00D5    A4 F2           LDY SAVE+2      ; LOAD Y

00D7    A5 F3           LDA SAVE+3      ; LOAD A

00D9    28              PLP             ; PULL FLAGS FROM STACK

                        ;THE BOX

00DA    EA              NOP             ; A BOX OF THREE BYTES

00DB    EA              NOP             ; TO HOLD LARGEST INSTRUCTION

00DC    EA              NOP             ; ALWAYS RESTORE TRAILING NOP'S

                        ;SAVE PROCESSOR STATE AFTER EXECUTION

00DD    08              PHP             ; PUSH FLAGS

00DE    86 F1           STX SAVE+1      ; SAVE X

00E0    84 F2           STY SAVE+2      ; SAVE Y

00E2    85 F3           STA SAVE+3      ; SAVE A

00E4    68              PLA             ; PULL FLAGS

00E5    BA              TSX

00E6    86 FF           STX SAVE+4      ; SAVE STACK POINTER

00E8    85 F0           STA SAVE        ; SAVE FLAGS

00EA    4C 47 FE        JMP $FE47       ; RETURN TO MONITOR ADDRESS
                                          MODE
```

Figure 9.1 Execute-in-a-Box Program

When you have looked around enough you are ready to put the next instruction in the box. At the beginning of a sequence of instructions, you may want to set up initial values for the registers by setting the corresponding memory locations.

One restriction you must observe: avoid placing a branch or jump instruction in the box. Such an instruction would allow the processor to escape from the box, with unpredictable results. No problem though. Branches and jumps effect only the sequencing of instructions, which you are handling anyway, so you would leave them out of any sequence of instruction you were executing step by step.

Think of memory (RAM) as divided into blocks of 256 bytes each. These blocks are called pages. The high byte of an effective address can be considered as identifying a page, and the low byte as identifying a location within the page. The first page in memory has the identifying number 00 (zero) and is therefore known as the zero page. By having an opcode for a zero page addressing mode, the processor can be told to supply a zero for the left byte of the effective address and the zero byte can be left out of the program. This saves memory space and the execution cycle time it would take to fetch the zero byte from memory. This feature makes zero page memory locations very valuable. The zero page indexed addressing modes work just as you would suspect. The carry from the indexing sum is added to the zero byte supplied by the processor to form the left byte of the effective address.

To load the "execute-in-a-box" program shown in Figure 9.1, type the following (remember "*" stands for Return):

.ØØDØ/A5*FØ*48*A6*F1*A4*F2*A5*F3*28*

EA*EA*EA*Ø8*86*F1*84*F2*85*F3*

68*BA*86*FF*85*FØ*4C*47*FE.

To run the program, simply type .ØØDØG.

The program will return to the Monitor address mode after executing the instruction located at $DA-$DC. This makes it easy to explore the effect of any instruction on the X, Y, A and S internal 65Ø2 registers. You may experiment by inserting opcodes of various instructions into locations $DA-$DC, running the program and examining locations $FØ-$FA to determine that instructions effect on the 65Ø2's internal registers.

As an example, suppose we want to simulate "good listener" instruction by instruction. We might preset registers X and A by

.ØØF1/ØØ .ØØF3/2B   or   .ØØF1/ØØ**2B

and then insert

STA $D146

in the box and execute "execute-in-a-box," with

.ØØDA/9D*46*D1.ØØDØG

then examine $D146 to see if it changed.

We are not ready to fully explain the "execute-in-a-box" program now, but one important feature we can consider now is the zero page addressing mode used to access the data beginning at $ØØFØ (SAVE).

Load the program and try it out. It may be your best friend when you are trying to find out what is going on in some other program.

SECTION 10

START WAVING YOUR FLAG

Decisions are implemented in programs by branch instructions. In these instructions, the test of a condition determines whether or not the program counter is loaded with a new value. In the 6502, branching is controlled by a set of one-bit registers called flags. The important flags for branching are given the symbolic names N, V, Z and C by 6502 programmers. They are known as the negative (N), overflow (V), zero (Z), and carry (C) flags.

Flags are set (to 1) and cleared (to 0) by the action of instructions. In fact, programmers need to know how each instruction affects the flags. Look at the Figure 8.1 summary of the 6502 move instructions. The letters at the right indicate when the corresponding flag is affected by the instruction. If you know that an operation affects a certain flag, you'll know whether it sets or clears the flag, because flags have consistent, easy-to-learn meanings. Interpret the N, V and Z flags as follows:

N = 1 means the result is negative (0 means positive or zero)

V = 1 means the result is invalid (more about this later)

Z = 1 means the result is zero

The carry flag represents the carry bit or some other one-bit extension of the result. As we go through more of the instruction set you will see how useful the carry bit extension is.

MNEMONIC	EXPLANATION	OPCODE	ADDRESS MODE
BCC	Branch if carry is clear (C = 0)	90	Relative
BCS	Branch if carry is set	B0	Relative
BEQ	Branch if equal (Z = 1)	F0	Relative
BNE	Branch if not equal (Z = 0)	D0	Relative
BMI	Branch if minus (N = 1)	30	Relative
BPL	Branch if plus	10	Relative
JMP	Unconditional jump	4C	Absolute
		6C	Indirect

FOR A BACKWARD JUMP, SUBTRACT THE NUMBER OF BYTES FROM 255 AND PUT THAT NUMBER AFTER THE OPCODE *

FOR A FORWARD JUMP TAKE THE NUMBER OF BYTES YOU WANT TO JUMP STARTING FROM AFTER THE OPERAND

		ADDRESS MODE								
MNEMONIC	EXPLANATION	immediate	absolute	zero page	(ind,X)	(ind),Y	zero page,X	abs,X	abs,Y	FLAGS
CMP	Set flag by A − Memory	C9	CD	C5	C1	D1	D5	DD	D9	N,Z,C
CPX	Set flag by X − Memory	E0	EC	E4						N,Z,C
CPY	Set flag by Y − Memory	C0	CC	C4						N,Z,C

START COUNTING THE Figure 10.1 6502 Branch and Compare Instructions
BYTES AT THE OPCODE OF THE BRANCH INSTRUCTION ?
(COUNT THE OPCODE AS A BYTE)

21

? CHECK OUT - MAY BE THE OPERAND INSTEAD OF THE OPCODE

Figure 1Ø.1 summarizes the 65Ø2 branch instructions. The conditional branch instructions check for a particular flag condition and loads the program counter if it is met. If the condition is not met, no branch occurs. The program counter simply advances to the next opcode. The conditional branches use the relative address mode. In this mode, a full effective address is loaded into the program counter with a new value formed from a single byte following the opcode. This is done by adding a single byte, the displacement byte, to the program counter. A negative instruction byte allows the processor to branch backwards in the program, as you can see happening in "good listener." More details on relative addressing will come up in the next section.

The mnemonic JMP is used for an unconditional branch or jump. No flag testing is involved and the branch is taken every time. Absolute, rather than relative, addressing mode is used. For an example, see "good listener." The absolute address ØØØD would have to be changed if "good listener" were moved to some other location in memory. It should always be the address of the instruction with the label REPEAT. If the symbolic program were given to an assembler program, the label REPEAT would be assigned the address as a numerical value. If the assembler is told where "good listener" is to be placed in memory, it will know the absolute address for the JMP instruction. The Monitor's machine code provides a good illustration of the JMP indirect mode shown in Figure 1Ø.1. In indirect addressing, the address provided by the instruction is used to fetch the address of the operand. In the case of the JMP, the operand is the jump destination, the address to replace the program counter. When you depress the 'G' key, the Monitor branches to the machine language instruction 6C FE ØØ. Can you figure out what is in locations $ØØFE_{16}$ and $ØØFF_{16}$ when the Monitor is running? Elementary, my dear Watson. The answer is on page 23.

Some instructions are devoted to setting flags and do nothing else. The are called "compare" instructions. In the 65Ø2 instruction set, compare instructions subtract a memory byte from the contents of a processor register and set the flags according to the result, leaving the operands unchanged. The 65Ø2 compares are included in Figure 1Ø.1. Many decisions represented by diamond boxes in flowcharts are implemented by a compare instruction, followed by a conditional branch.

When the distance between a branch instruction and the intended branch destination is too great for one byte, the JMP instruction with its absolute address mode can come to the rescue. For example, in

       CMP #_____

       BEQ AWAY,

if the location represented by AWAY is too far away, requiring too large a value for the displacement byte after the opcode FØ, then use

          CMP #_____

          BNE SKIP

          JMP AWAY

      SKIP

There are all sorts of interesting variations to be made in "good listener," based on recognizing a particular character as a command. For example, one could have "good listener" return to the address mode of the Monitor on command. In the Monitor you could change the location or length of the display line, then restart "good listener." This version could be used to compose character pictures on the screen. For your JMP into the Monitor, $FE43_{16}$ is a good address, but remember to reverse the bytes. For the compare instruction to recognize the new command you'll need the ASCII code for the command's key. If necessary, you can call upon the binary display program of Section 4, or consult the appendix.

Another idea is to have "good listener" advance the display area to a new line, all by itself, on command. After all, the line location in the program is in RAM when the program is loaded and can be treated as variable data.

When making a change in a program, the insertion of instructions changes the location of instructions following the insertion point. Most instructions are not affected by such relocations, but some are. Absolute address mode instructions may have to be adjusted. In relative addressing, any insertion or deletion between a branch instruction and its branch destination effects the displacement byte following the branch opcode. Watch your step when making changes. Bugs are easier to prevent than to find.

When executing in a box with the program of Figure 9.1, you can follow the changes in the flags. The program restores and saves all flags in location $ØØFØ_{16}$, in the binary format

7	6	5	4	3	2	1	Ø
N	V					Z	C

For example, if the Monitor command .00F0 displays A6 in the data display, then the binary code

      1010 0110

reveals that N = 1, V = 0, Z = 1, C = 0 at that point.

*Answer: 00FE,00FF contain the display address.

SECTION 11

A TOTAL MYSTERY

How about a little challenge? Load the following program, double check the loading and start the execution.

LOCATION	PROGRAM
.0000/	D8*A9*00*85*FF*85*FE*85*
0008	FC*20*ED*FE*C9*2B*D0*09*
0010	A5*FE*18*65*FC*50*36*70*
0018	0B*C9*2D*D0*24*38*A5*FE*
0020	E5*FC*50*29*A2*03*B5*54*
0028	9D*D0*D0*CA*10*F8*20*ED*
0030	FE*C9*0D*D0*F9*A2*03*A9*
0038	20*9D*D0* D0*CA*10*FA*30*
0040	C0*20*93*FE*30*C3*A2*00*
0048	20*DA*FE*50*02*85*FE*20*
0050	AC*FE*D0*B5*54*49*4C*54
.0000 G	

Now key-in data and try to discover what the program does. Here are vital clues: the program reacts only to the hexadecimal digits 0-F, the signs '+' and '−', and the 'RETURN' key. The 'RETURN' key is ignored at one point but is vital at another.

You will find an explanation of the program, starting on the next page, but don't spoil the fun by peeking. Try all sorts of input, watch and record what happens, make guesses and test them. There is a special message which can appear but it goes away. The program loops forever.

Bumfuzzled? If so, here is some information that may clear things up. Do the results you cannot explain involve data having a left hexadecimal digit of eight or greater? To the 6502 processor, such binary codes represent negative numbers. Try starting with a zero and subtracting a positive value, one that reads $7F (a "$" preceding a number indicates hexadecimal) or less. The result represents the negative number of the same size. Perhaps now you can explain everything the program is doing, before looking at the flowchart of Figure 11.1, which revals all. Try to determine the largest positive sum and the smallest negative sum that can be obtained.

There are many ways to represent negative numbers in binary codes. For binary integer arithmetic, most computers use the system that the 6502, the one illustrated by the mystery program. It is called two's complement representation. To change the binary representation of a number to the two's complement negative, you first change every bit to its opposite value. This is referred to as complementing the bits. Then add 1 (one) to the result. Since the process amounts to changing the sign of a number, repeating it should produce the original binary code. Does it?

Two's complementing can be done directly in hexadecimal. Replace each hex digit by a complement digit that you obtain by subtracting the original digit from 15_{10}. This does the complementing. Then add 1 (one) to the result.

Trace through the flowchart of Figure 11.1 and try out any parts you did not get into with your blind exploration of the program. The flowchart shows something important about two's complement representation. There are no tests in the algorithm to determine the sign of the data values. With two's complement representation, positive and negative values are processed in exactly the same way.

24
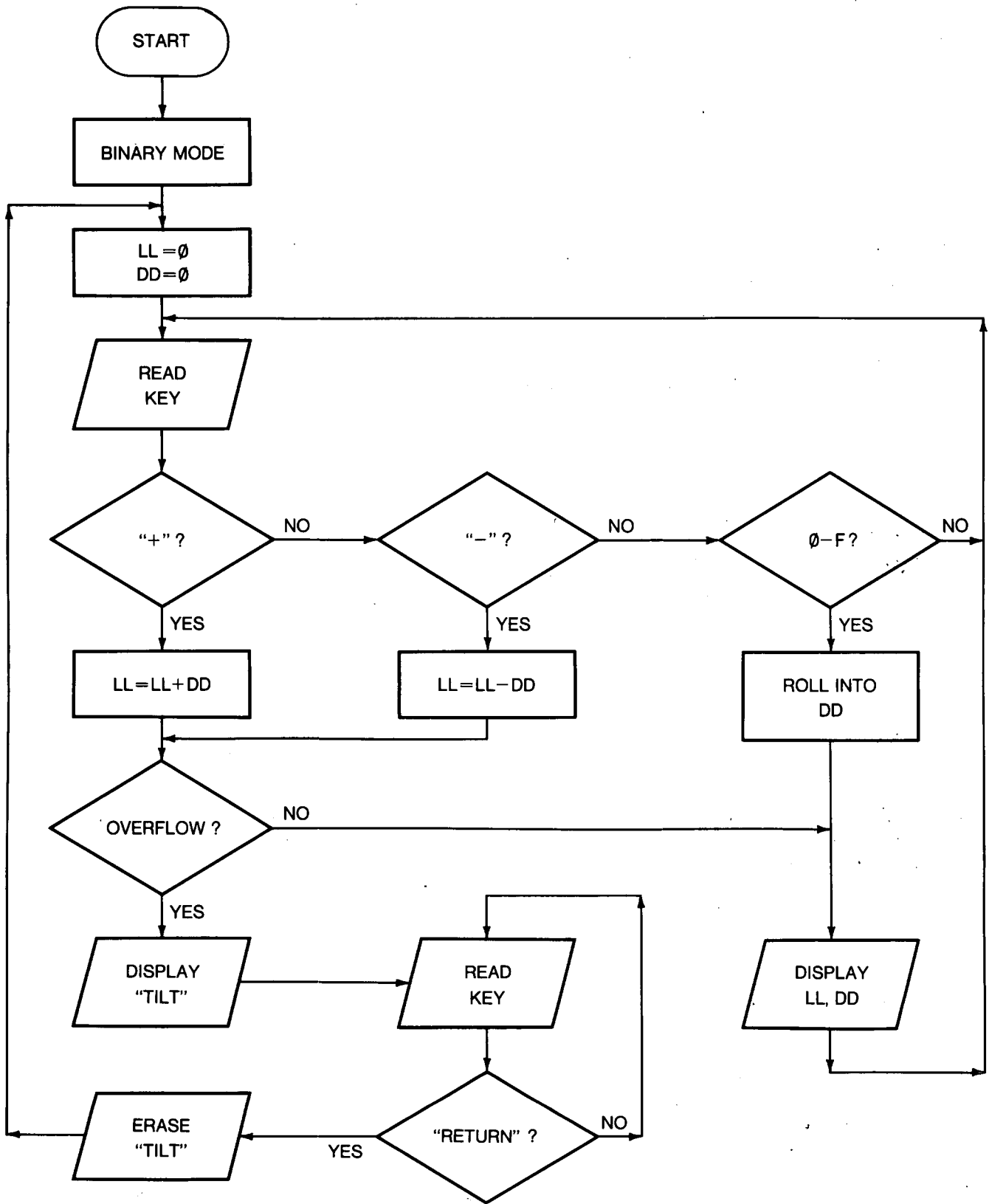
Figure 11.1 Flowchart for "Total Mystery"

Did you get the 'TILT' message? It appears when the overflow flag is set by the add or subtract operations. In either case, the message means that the correct result cannot be represented in one byte.

Now that we have reached two's complement, we can understand the 6502 backward branches in relative addressing. In the Figure 7.3, coding of "good listener," the test for the end of the loop is

BNE FILL

The reference to the label FILL is assembled as $F5, the two's complement representation of a negative displacement. In the execution of the instruction, the program counter is first advanced to $000D, then the negative byte $F5 is expanded to a 16-bit representation of the same number and is added to the updated value of the program counter to produce

$$
\begin{array}{cc}
000D_{16} & \\
 & \$000D \\
+FFF5_{16} \quad \text{or} \quad & +\$FFF5 \\
\hline
0002_{16} & \$0002
\end{array}
$$

as the branch destination. The addition is ordinary garden-variety binary addition whether the displacement is negative or positive. Does the backward branch to MOVE work the same way?

SECTION 12

A LOOP YOU CAN COUNT ON

In the "good listener" code of Figure 7.3, there are three loops. One pair of loops is nested, one within the other. Two of good listener's loops are controlled by counters to execute a prescribed number of times. The index register X is used as the loop counter within each of these counting loops.

Figure 12.1 shows the usual arrangement of the parts of a counting loop. The body of the loop is the part that is repeated. It is a set of instructions that may contain other loops. The loop counter is given an initial value outside of the loop. Within the loop it is incremented or decremented until it reaches some test value. The test value may be a constant or a variable, depending on when the number of times through the loop is determined.
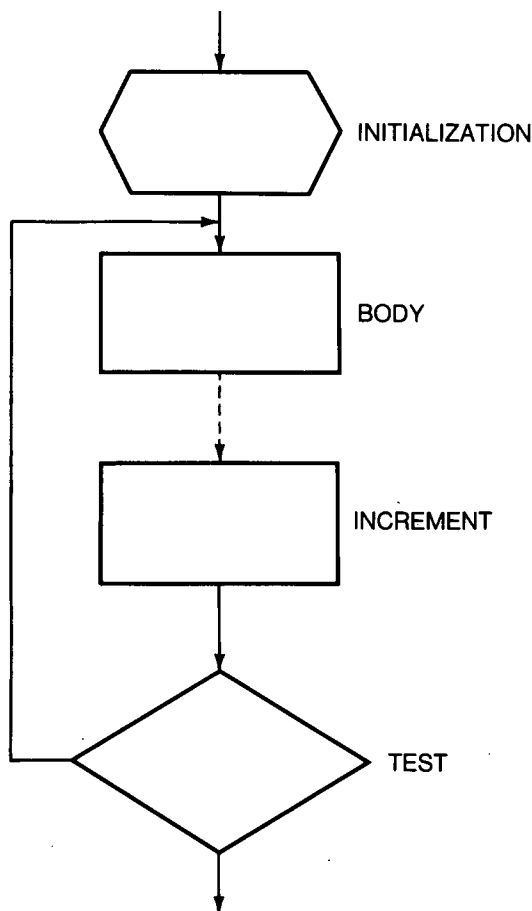
Figure 12.1 Parts of a counting loop

Many a program bug has hatched when the programmer forgot to initialize the loop counter or chose the wrong test value or branch instruction, causing the loop body to be executed the wrong number of times. A wise precaution is to double check immediately after coding a loop to see what values the loop counter will have on the first and last executions of the body. If undetected, this kind of bug crawls off into some other part of the program and causes something apparently unrelated to go awry. A good way to save yourself a lot of effort looking for bugs of this type is to make it a practice to play processor with a new program, going through it instruction by instruction, calculating the changes that take place and looking for surprises. Programmers call this desk checking because it is done at the desk and not on the computer. You can play processor with "execute-in-a-box."

The 6502 instructions for incrementing and decrementing are shown in Figure 12.2. The index registers X and Y make the best loop counters, but a way is provided to use a memory cell as a counter as well. The reason is that with

27

loops nested within one another, so that several are repeating at one time, an index register would not be available to control each loop. X and Y are preferred as loop counters, primarily because of their ability to access data through indexing. As loop bodies are repeated, the algorithm is often moving through blocks of data in a systematic way. Why do you suppose that increment and decrement instructions are not provided for the accumulator? Generally, it would be occupied with something in the body of the loop and not be available as an index register or loop counter.

ADDRESS MODES

MNEMONIC	EXPLANATION	absolute	zero page	zero page,X	absolute,X	FLAGS
DEC	Decrement memory by 1	CE	C6	D6	DE	N,Z
INC	Increment memory by 1	EE	E6	F6	FE	N,Z
DEX	Decrement X by 1	CA	Implied			N,Z
DEY	Decrement Y by 1	88	Implied			N,Z
INX	Increment X by 1	EB	Implied			N,Z
INY	Increment Y by 1	C8	Implied			N,Z

Figure 12.2 Counter Increment and Decrement Instruction

You probably noticed that the increment and decrement instructions affect the flags N and Z. This makes it possible to detect when a loop counter reaches a zero or negative value without a compare instruction. When you have a choice, it may be better to count backwards to take advantage of this feature. As an example, look at the program of Figure 12.3 which displays all the graphic symbols with a given high (left) hexadecimal digit in their codes. With this program you can rapidly survey the graphic symbols available to your computer.

```
                    ; USER KEYS HEX DIGIT, D, PROGRAM DISPLAYS ALL GRAPHICS, DK
                    ; WHERE K IS A HEX DIGIT
0000 20 ED FE BEGIN JSR  $FEED    ; GET A KEY
0003 20 93 FE       JSR  $FE93    ; STRIP TO HEX DIGIT
0006 30 F8          BMI  BEGIN    ; IGNORE NON-DIGITS
0008 0A             ASL  A
0009 0A             ASL  A
000A 0A             ASL  A
000B 0A             ASL  A
000C 85 1B          STA  TEMP     ; SAVE
000D A2 0F          LDX  #15      ; INITIALIZE COUNTER
0010 8A        LOOP TXA           ; COUNT TO A FOR LOW DIGIT
0011 05 1B          ORA  TEMP     ; APPEND HIGH DIGIT
0013 9D 46 D1       STA  $D146,X  ; STORE IN DISPLAY LINE
0016 CA             DEX           ; INCR, TEST
0017 10 F7          BPL  LOOP     ; DISPLAY LOADING
0019 30 E5          BMI  BEGIN    ; WAIT FOR ANOTHER INPUT
001B                TEMP
```

To enter this program, type:

```
.0000  /  20*ED*FE*20*93*FE*30*F8*
           0A*0A*0A*0A*85*1B*A2*0F*
           8A*05*1B*9D*46*D1*CA*10*
           F7*30*E5—*—.
```

The blanks at $001B are determined by where you locate TEMP. For example, to locate TEMP at $D400, put 00 and D4 in the blanks.
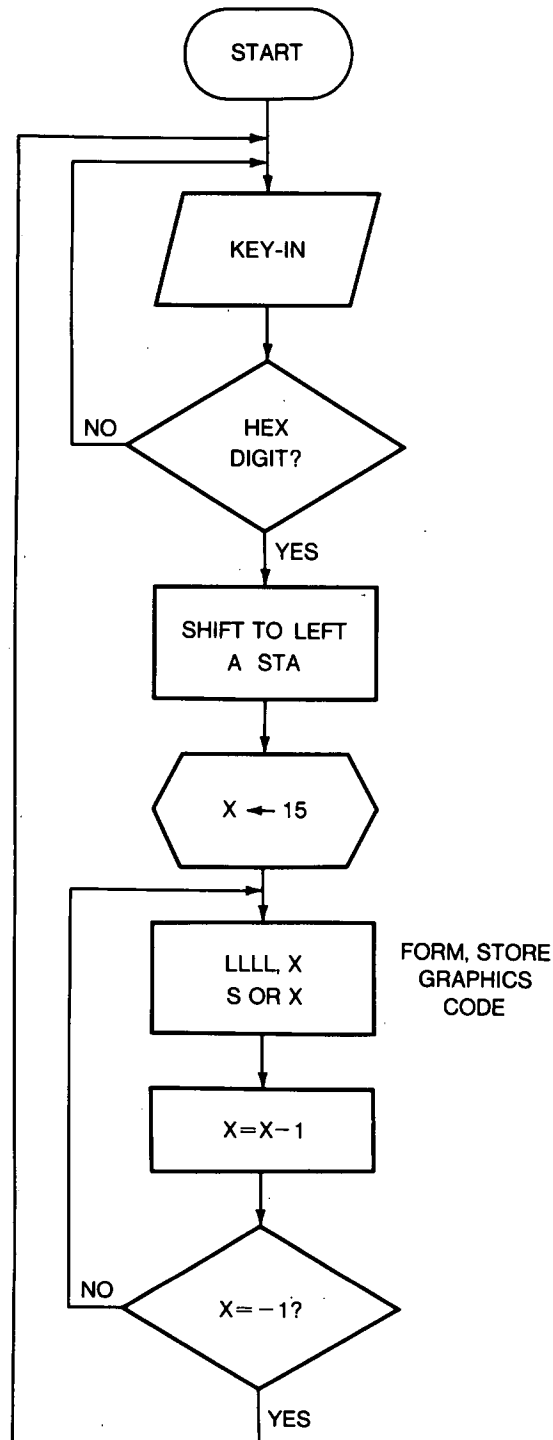
Figure 12.3 A graphics display program

In the algorithm, it doesn't matter which order the graphic codes are loaded into display locations, so they are loaded right-to-left, decreasing order. The BPL instruction allows a repeat of the body of the loop with X = $\emptyset$. A BNE instruction would make the X = 1 iteration the last. The BMI instead of the JMP is a trick to save a byte of program. TEMP is assigned a byte at the end of the program instructions.

If you are looking for a programming challenge involving counting loops, you could attempt a program to produce the same output as this one, but with the displayed symbols arranged in a table of four rows by four columns, with single blank spaces between all of the symbols.

SECTION 13

A STACK OF LETTERS

Beware. You are in Section 13. To avoid bad luck while in this section, spell 'abracadabra' backwards. Use the program of Figure 13.1. This program will save a word in memory without letting you see it until you depress the space bar. Then it moves the word into the display area in reverse order. Without modification, the program allows up to 256 characters in an input word.

```
                    ;
                    ;BACKWARDS SPELLER: ENTER A WORD, FOLLOWED BY A BLANK.
                    ;THE WORD THEN APPEARS, BACKWARDS.
                    ;TO ERASE, PRESS ANY KEY, THEN START A NEW WORD.
                    ;

0000   A2  40          GO     LDX   #64       ; LONGEST WORD
0002   A9  20          GO     LDA   #C'
0004   9D  45  D1      BLA    STA   $D145,X   ; BLANK OLD WORD
0007   CA                     DEX
0008   D0  FA                 BNE   BLA       ; TEST LEAVES X=0
000A   20  ED  FE      BUILD  JSR   $FEED     ; GET A LETTER
000D   C9  20                 CMP   #C'       ; IF A BLANK
000F   F0  04                 BEQ   DUMP      ; WORD ENDS
0011   48                     PHA             ; IF NOT, PUSH IT
0012   E8                     INX             ; COUNT PUSHES
0013   D0  F5                 BNE   BUILD     ; RETURN FOR ANOTHER
0015   A0  00          DUMP   LDY   #0        ; INDEX FRONTWARDS
0017   68             POP     PLA             ; POP A LETTER
0018   99  46  D1             STA   $D146,Y   ; SPELL IT OUT
001B   C8                     INY             ; NEXT LETTER POSITION
001C   CA                     DEX             ; COUNT POPS
001D   D0  F8                 BNE   POP
001F   20  ED  FE             JSR   $FEED     ; DELAY BLANKOUT UNTIL NEXT KEY
0022   4C  00  00             JMP   GO        ; IS PRESSED ON RELEASE, REPEAT
```

Figure 13.1 The Backwards Speller

This little trick can be coded a number of ways on the 6502, but our program uses a feature of this processor that we have not yet considered, the stack pointer. A stack is a type of data structure, an arrangement of data that provides access to an item of data in a particular way. The stack provides last-in first-out (LIFO) access, meaning that it makes available one item at a time, and the available item is the latest one that was placed on the stack. Take that item from the stack and the previously entered item becomes available.

In computer terminology, the available item is called the top of the stack. The operation of placing an item on the stack is a push; removal of an item is a pop or a pull. The backwards speller pushes characters on the stack as you

enter them, then pulls them all when it sees a blank. They come out in reverse order.

In the 65Ø2, each item on the stack is a byte of information. The data on the stack is not kept in the processor but in memory. The processor has a one-byte register called the stack pointer, S. The stack pointer contains the location of the top of the stack. A value of 1 (one) is always used as the page number in stack operations, so the stacked data is always contained in page one of memory (the second block of 256 bytes). A push involves writing the byte into the page one memory byte specified by the stack pointer and decrementing the stack pointer. A pull involves incrementing the stack pointer and reading the byte from the page one memory byte specified by the stack pointer.

Figure 13.2 shows several 65Ø2 stack instructions. The first group is used to access the stack. In the second group, the TXS instruction is the means provided to set the stack pointer to its initial value. In your system, the Monitor has taken care of that, so a TXS need not appear in your programs. Whenever a program uses the stack, it should pull from the stack just what it pushed, no more and no less. This leaves the stack in its beginning state. The backwards speller uses a loop counter to see that this is done.

MNEMONIC	EXPLANATION	OPCODE	ADDRESS MODE	FLAGS
PHA	Push A onto the stack	48	Implied	None
PHP	Push flags onto the stack	Ø8	Implied	None
PLA	Pull from the stack into A	68	Implied	N,Z
PLP	Pull from the stack into flags	28	Implied	All
TSX	Copy stack pointer into X	BA	Implied	N,Z
TXS	Copy X into stack pointer	9A	Implied	None

Figure 13.2 Explicit 65Ø2 Stack Operations

The "execute-in-a-box" program of Section 9 takes advantage of available stack operations to keep track of changing flags. The set of flags is considered collectively as a register, P, which can be pushed or pulled from the stack. That represents the 65Ø2's only access to the flags as a group. Once on the stack, a pull into the accumulator is a means of transferring the flags to a memory cell for your inspection.

You are allowed to use push and pull operations within the "execute-in-a-box" program because the segments of the program before and after the box leave the stack as they found it. The TSX instruction is used after the contents of X have been saved, to allow you to follow changes in the stack pointer contents.

SECTION 14

IT'S THE SAME OLD ROUTINE

You must have noticed the same instruction occupying an important spot in "good listener," the graphics display and backwards speller, namely

   JSR $FEED

From the remarks in the programs, the effect of the instruction appears to be to put the ASCII code for the next key depressed into the accumulator. Have you tried executing this instruction in a box yet? When you put the code 20 ED FE in the box and execute the program, there is no return to the Monitor, as usual. That is, not until you depress a key and release it. It's an extraordinary instruction that can make the processor wait around all day until you depress that key, isn't it?

To tell the truth, the JSR doesn't do all this. Rather, it branches to a sequence of instructions that does. The set of instructions is called a closed subroutine or simply, a subroutine. The processor begins to execute code starting at $FEED, continuing until it encounters the opcode $60. (RTS) Then, magically enough, it branches right back to the instruction following the JSR instruction. With this information and the help of Figure 14.1, you can do some detective work and figure out what the subroutine at $FEED really does.

Figure 14.1 is a disassembly table. Take the hexadecimal code in a location and select a row in the table by the left digit and a column by the right digit. If the code is an opcode, the row and column intersection will show the operation and the addressing mode. What makes it fun is that now you must find which bytes represents the next opcode, and the correct interpretation of the bytes in between. Happy hunting!

Where the subroutine at $FEED appears to be reading a memory location, it is not a memory location at all. It is what is called a port, a connection to an input or output device. In this case, it is an input device, the computer's keyboard. There is a signal transmitted to the port when a key is depressed. When all keys are released, the data received through the port is loaded into the accumulator.

A program containing a JSR is said to call the subroutine starting at the branch address in the instruction. The JSR makes provisions for returning the processor to the calling program. It does so by pushing the value contained in the advanced program counter onto the stack before replacing it to execute the branch. The pushed address is called the return address. A subroutine may contain any number of return instructions, written as a mnemonic RTS in the symbolic form of the subroutine. When the RTS ($60) is encountered, the return address is pulled from the stack into the program counter producing the branch to the point immediately after the JSR instruction.

When are subroutines useful? Primarily, when the same processing is needed in more than one place. "Good listener," for example, reads the next key-in in two places. The single copy of the processing code is executed wherever it is needed. Subroutine calling is an important feature that multiplies the power of computer systems. Collections of subroutines can be built to carry out frequently used functions in users' programs.

Figure 14.2 describes some of the subroutines in the Monitor. The display subroutine at $FEAC can be used to make an improvement in "execute-in-a-box" without adding much to its length. When storing register contents in $00F0 through $00F4, also store key register contents in $00FC, $00FE and $00FF. Or, replace a key memory location in the simulated program by one of these addresses. When "execute-in-a-box" returns to the Monitor, a call to $FEAC is the first instruction executed, so the display of desired trace data is automatic.

OPCODE TABLE

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

LSD—Least Significant Digit
MSD—Most Significant Digit

Figure 14.1 6502 Disassembly Table

ENTRY	EFFECT REGISTERS	EFFECT
$FE93	A	Replaces ASCII hexadecimal digit with its binary value, N=Ø If not a digit, returns $8Ø and N=1
$FEAC	A,X,Y	Displays as hexadecimal digits LLLL DD the contents of ØØFF and ØØFE (LLLL) and ØØFC (DD)
$FECA	A,Y	Stores the ASCII code for the right hexadecimal digit A in LLLL,Y and increments Y by 1
$FEDA	A,Y	Shifts right digit of A into ØØFD,X and ØØFC,X as shown. Clears A,Y
$FEE9	A	Get next ASCII character from keyboard or UART, depending on location ØØFB
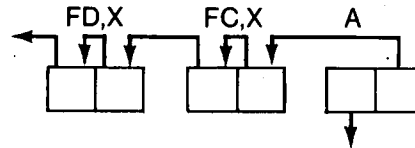
Figure 14.2 Some Useful Monitor Subroutines

SECTION 15

EXTENDING THE MYSTERY

It's time to come forward with the whole story on the "total mystery" program of Section 11. Since "total mystery" calls many monitor subroutines, it seemed only right to keep its workings under wraps until the subroutines were explained. Now Figure 15.1 exposes all of total mystery's secrets.

```
                ;
                ;TOTAL MYSTERY
                ;
0000    D8              BEGIN    CLD                ; CLEAR DECIMAL MODE
0001    A9  00          REPEAT   LDA     #0         ; CLEAR LLLL, DD DISPLAY
0003    85  FF                   STA     $FF
0005    85  FE                   STA     $FE
0007    85  FC                   STA     $FC
0009    20  ED  FE      GET      JSR     $FEED      ; GET KEY
000C    C9  2B                   CMP     #$2B       ; A '+'?
000E    D0  09                   BNE     NEG        ; NO, TEST FOR MINUS
0010    A5  FE                   LDA     $FE        ; FOR
0012    18                       CLC                ; YES, CLEAR CARRY
0013    65  FC                   ADC     $FC        ; ONE-BYTE ADD
0015    50  36                   BVC     STORE      ; NO OVERFLW, STORE AND DISPLAY
0017    70  0B                   BVS     TILT
0019    C9  2D          NEG      CMP     #$2D       ; A '−'?
001B    D0  24                   BNE     DIG        ; NO, TEST FOR DIGIT
001D    38                       SEC                ; SET CARRY
001E    A5  FE                   LDA     $FE        ;
0020    E5  FC                   SBC     $FC        ; ONE-BYTE SUBTRACT
0022    50  29                   BVC     STORE      ; NO OVRFLW, STORE AND DISPLAY
0024    A2  03          TILT     LDX     #3
0026    B5  54          MLP      LDA     MESS,X     ; WRITE MESSAGE TO DISPLAY
0028    9D  D0  D0               STA     $D0D0,X
002B    CA                       DEX
002C    10  F8                   BPL     MLP
002E    20  ED  FE      WAIT     JSR     $FEED      ; WAIT FOR RETURN KEY
0031    C9  0D                   CMP     #$0D
```

36

0033	D0 F9		BNE	WAIT	
0035	A2 03		LDX	#3	
0037	A9 20		LDA	#$20	; CLEAR MESSAGE
0039	9D D0 D0	ERASE	STA	$D0D0,X	
003C	CA		DEX		
003D	10 FA		BPL	ERASE	
003F	30 C0		BMI	REPEAT	
0041	20 93 FE	DIG	JSR	$FE93	; STRIP TO DIGIT
0044	30 C3		BMI	GET	; IF NOT A DIGIT, IGNORE
0046	A2 00		LDX	#0	
0048	20 DA FE		JSR	$FEDA	; ROLL DIGIT INTO DD
004B	50 02		BVC	DISP	
004D	85 FE	STORE	STA	$FE	; STORE RESULT IN LLLL+1
004F	20 AC FE	DISP	JSR	$FEAC	; DISPLAY LLLL DD, CLEAR Z
0052	D0 B5		BNE	GET	
0054	54 49	MESS	.BYTE	'TILT'	
0056	4C 54				

Figure 15.1 The Total Mystery Program

The heart of "total mystery" is the add and subtract instructions ADC and SBC. Since they are the basis of all arithmetic processing on the 6502, an ample set of addressing modes is provided for these instructions, as seen in Figure 15.2. Addition and subtraction are carried out in the 6502 by loading the accumulator, then using ADC or SBC to form the sum or difference in the accumulator. The carry flag gets in on the act: ADC sums accumulator, operand byte and carry flag. SBC subtracts the operand from the accumulator and adds in the complement of the carry as a "borrow." To get a correct one-byte addition, you must make sure that the carry flag is cleared beforehand. An instruction CLC is available to do this. It was not necessary in "total mystery" because the compare instruction at $001C clears the carry flag so that no borrow is assumed. The SEI instruction of Figure 15.2 serves that purpose.

ADDRESS MODES

MNEMONIC	EXPLANATION	immediate	absolute	zero page	(ind,X)	(ind),Y	zero page,X	absolute,X	absolute Y	FLAGS AFFECTED
ADC	Add with carry to accumulator	69	6D	65	61	71	75	7D	79	NZCV
SBC	Subtract with carry from accumulator	E9	ED	E5	E1	F1	F5	FD	F9	NZCV
CLC	Clear carry flag	18	Implied							C=0
SEC	Set carry flag	38	Implied							C=1

Figure 15.2 Add and Subtract and Carry Flag Set Up

The reason for involving the carry flag in ADC and SBC is to enable addition and subtraction to be extended to numbers larger than one byte. Starting at the rightmost byte of the numbers, the value of C, resulting from each addition or subtraction represents the carry or borrow required into the next byte to the left.

```
        FE                    FF    FE
       ┌──┐                  ┌────┬────┐
       └──┘                  └────┴────┘

        FC                    FD    FC
       ┌──┐     is to be     ┌────┬────┐
    +  └──┘     replaced   + └────┴────┘
    ─ ────        by         ─ ────────

        FE                    FF    FE
       ┌──┐                  ┌────┬────┐
       └──┘                  └────┴────┘
```
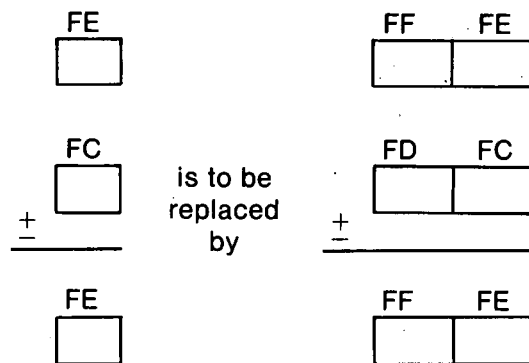
Replace ADC $FC at ØØ13 with:

ØØ13	9Ø	43		BCC	ADD2

and add to the end of the program (ØØ58):

ØØ58	65	FC	ADD2	ADC	$FC
ØØ5A	85	FE		STA	$FE
ØØ5C	A5	FF		LDA	$FF
ØØ5E	65	FD		ADC	$FD
ØØ6Ø	4C	15	ØØ	JMP	$ØØ15

Replace SBC $FC at ØØ2Ø with:

ØØ2Ø	BØ	41		BCS	SUB 2

and add to the end of the program (ØØ63):

ØØ63	E5	FC	SUB2	SBC	$FC
ØØ65	85	FE		STA	$FE
ØØ67	A5	FF		LDA	$FF
ØØ69	E5	FD		SBC	$FD
ØØ6B	4C	22	ØØ	JMP	$ØØ22

and change STA $FE at ØØ4D to

ØØ4D	85	FF	STORE	STA	$FF

Figure 15.3 Extending Total Mystery to Two-Byte Arithmetic

Figure 15.3 shows how to extend the "total mystery" program to two-byte arithmetic. The carry produced by adding $ØØFE and $ØØFC contributes to the sum of $ØØFF and $ØØFD. The overflow flag has meaning only when the most significant, or leftmost, bytes of the numbers have been processed. The arrangement of the numbers in memory may not seem very natural to you. It is done this way in order to make use of the Monitor subroutines at $FEAC and $FEDA. A more natural arrangement is illustrated in the subroutines of Figure 15.4. Can you assemble machine language versions of these routines?

```
              ;
              ; ADD INTEGERS IN BYTES LONG, STARTING AT $EØ AND $E8
              ; WORKS FOR N UP TO 8.
              ; CALL WITH X=N-1. AFFECTS A.
              ;
18                ADDN CLC
B5  EØ            ANLP LDA $EØ,X
75  E8                ADC $E8,X
95  EØ            STA $EØ,X
CA                DEX
1Ø  F7            BPL ANLP
6Ø                RTS



              ;
              ; UNSIGNED ADD, OF A TO N-BYTE ACCUMULATOR AT $EØ
              ; CALL WITH X=N-1.
              ;
18                ADD1 CLC
75  EØ            A1LP ADC $EØ,X
95  EØ                STA $EØ,X
CA                DEX
A9  ØØ            LDA #Ø
BØ  F7            BCS A1LP
6Ø                RTS
```

Figure 15.4 Two Subroutines for N-Byte Addition

SECTION 16

BIT BY BIT

The 6502 processor can address bytes, but not bits within a byte. Two groups of instructions handle information at the bit level: shift instructions move bits to where they are needed, and bit logical instructions operate on bits individually.

6502 shift instructions are illustrated in Figure 16.1. The byte contents of the accumulator, or a memory cell, can be shifted one bit position, left or right. The carry flag acts as an extension of the shifted byte, receiving the bit value shifted out of the byte. The ASL and LSR instructions bring a cleared bit into the opposite end of the byte. The rotate instructions bring in the previous value of the carry flag. Generally, "arithmetic" shifts are those that produce multiplication or division by two, for both positive and negative values. Since shifting right to divide by two would not work for two's complement negative values, the right shift is called a logical shift rather than an arithmetic shift.

DIAGRAM	MNEMONIC	ADDRESS MODES					FLAGS
		Absolute	Zero Page	Accumulator	Z Page, X	Abs, X	
ASL	ASL	0E	06	0A	16	1E	N,Z,C
LSR	LSR	4E	46	4A	56	5E	N=0,Z,C
ROL	ROL	2E	26	2A	36	3E	N,Z,C
ROR	ROR	6E	66	6A	76	7E	N,Z,C

Figure 16.1 Shift instructions

The binary display program of Section 4 is based on shift instructions, using them to extract and use one bit of the displayed value at a time. The symbolic form is shown in Figure 16.2.

; BINARY DISPLAY PROGRAM OF SECTION 4

```
20  ED  FE  LOOP        JSR  $FEED    ; GET A KEY
85  F0                 STA  $F0      ; SAVE IN $F0
A9  18                 LDA  #$18     ; HEX 30, SHIFTED RIGHT
A2  07                 LDX  #7
9D  D2  D0  OUT        STA  $D0D2,X  ; SET UP DISPLAY CELL
66  F0                 ROR  $F0      ; GET NEXT BIT
3E  D2  D0             ROL  $D0D2,X  ; SHIFT BIT IN, PRESERVE CARRY
CA                     DEX
10  F5                 BPL  OUT
30  EA                 BMI  LOOP
```

Figure 16.2 Binary Display Program

A logical shift right in the accumulator is central to the subroutine of Figure 16.3. This subroutine could prove useful in many places. For one, use it to improve "execute-in-a-box" so that it displays the flags, X, Y, A and S automatically. You can put the subroutine anywhere without changing any code. Subroutines with this property are called relocatable. The key to writing relocatable subroutines is to avoid absolute or zero page addressing modes referencing bytes within the subroutine.

; HEX DISPLAY OF ZERO PAGE BINARY CODE STARTING AT X

; DISPLAY STARTS AT $D0C6,Y

; NUMBER OF BINARY BYTES IN A

; USES $FF FOR COUNTER

;

```
85  FF      HEXTV    STA  $FF      ; BINARY BYTE COUNTER
B5  00      LOOPH    LDA  $0,X     ; GET BINARY BYTE
4A                   LSR  A        ; SHIFT LEFT HEX DIGIT
4A                   LSR  A        ; TO RIGHT DIGIT
4A                   LSR  A
4A                   LSR  A
20  CA  FE           JSR  $FECA    ; ASCII TO $D0C6,Y; INCREMENT Y
B5  00               LDA  $0,X     ; GET A FRESH COPY
20  CA  FE           JSR  $FECA    ; RIGHT DIGIT ASCII T0 $D0C6,Y AND INC Y
EA                   NOP           ; FOR GAP BETWEEN BYTES, INSERT INY HERE
E8                   INX           ; ADVANCE IN BINARY INPUT
C6  FF               DEC  $FF      ; COUNTING INPUT BYTES
D0  EC               BNE  LOOPH
60                   RTS
```

Figure 16.3 Binary to Hexadecimal Display Subroutine

Normally you would put a subroutine somewhere beyond the first two pages of memory. Zero page locations are too valuable because any data in the zero page can be addressed with a single byte in the instruction. The stack

41

occupies page one of memory.

What happens when register Y is incremented beyond 255_{10} or FF_{16}? It just starts over, right? You can take advantage of this to display several lines in "execute-in-a-box," letting the active line do a 'round robin' progression over the display.

SECTION 17

IT'S VERY LOGICAL

In most computers the individual bits within addressable units are operated on by a technique known as masking. A binary code called a mask selects the bits to be operated upon. The operations are called bit logical or simply, logical operations. Like most processors, the 6502 provides bit logical operations for selectively clearing, setting and complementing bits.

In the logical operations, each bit of the mask interacts with the corresponding bit of the operand in the same position. The effects on operand bits are shown in Figure 17.1. The names AND and OR are rather universal names for these operations. EOR is short for exclusive or. The AND operation clears all bits in the operand selected by 0's of the mask. The OR sets all bits selected by mask 1's, while the EOR complements them. Figure 17.2 covers the 6502 logical instructions and a special compare instruction (BIT) which uses the accumulator as a mask to select individual bits for setting the zero flag Z.

As an example, if the accumulator contains

     10101100,

the instruction AND with 11110000 produces 10100000 in the accumulator,

the instruction AND with 00001111 produces 00001100 in the accumulator,

the instruction ORA with 11110000 produces 11111100 in the accumulator,

the instruction ORA with 00001111 produces 10101111 in the accumulator,

the instruction EOR with 11110000 produces 01011100 in the accumulator,

the instruction EOR with 00001111 produces 10100011 in the accumulator.
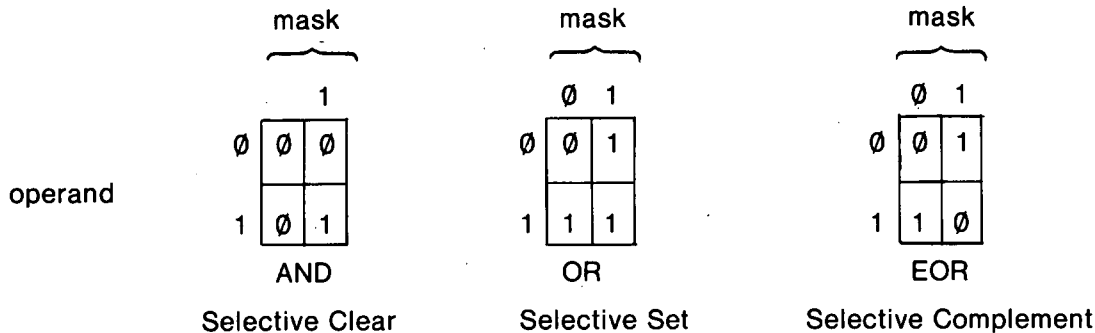
Figure 17.1 The Effect of Logical Operations

MNEMONIC	EXPLANATION	immediate	absolute	zero page	(ind,X)	(ind),Y	zero page,X	abs,X	abs,Y	FLAGS
AND	AND accumulator and memory mask	29	2D	25	21	31	35	3D	39	N,Z
ORA	OR accumulator and memory mask	Ø9	ØD	Ø5	Ø1	11	15	1D	19	N,Z
EOR	EXCLUSIVE-OR accumulator and memory mask	49	4D	45	41	51	55	5D	59	N,Z

BIT   Test memory by accumulator mask

2C - Absolute
24 - Zero page

Z = Ø if A AND memory = Ø
  = 1 otherwise
N = memory sign bit
V = next bit of memory

Figure 17.2 65Ø2 Logical Instructions

Logical operations are important in many algorithms because they represent, very neatly, the combining of sets of objects to form new sets. Imagine that we have devoted one bit to each object. Then a set of objects is represented by a binary code in which '1' means 'belongs to the set.' Now the AND of two codes is the intersection of the sets, the objects belonging to both. OR the codes to get the union of the sets, the set of objects belonging to one or the other. EOR produces the union but eliminates objects in both sets.

The Monitor subroutine starting at $FECA illustrates masking with logical instructions. It is called by the hexadecimal display subroutine of the last section. As Figure 17.3 shows, the subroutine constructs and stores the ASCII code for a hexadecimal digit contained in the right four bits of the accumulator. With your improved "execute-in-a-box," trace through the routine with some typical input data and confirm its behavior.

```
              ; DISPLAY AT $DØC6,Y THE ASCII CODE
              ; FOR THE RIGHT NIBBLE OF A
              ; INCREMENTS Y
              ;
FECA 29 ØF      ASCII AND #$ØF   ; REPLACE LEFT NIBBLE
     Ø9 3Ø            ORA #$3Ø   ; BY '3'
     C9 3A            CMP #$3A   ; WAS IT Ø-9?
     3Ø Ø3            BMI STO    ; YES, NO ADJUST NECESSARY
     18               CLC        ;
     69 Ø7            ADC #7     ; NO, ADD 7 FOR A-F ASCII
     99 C6 DØ STO     STA $DØC6,Y ; STORE IN DISPLAY LOCATION
     C8               INY        ; ADVANCE TO NEXT DISPLAY LOCATION
     6Ø               RTS
```

Figure 17.3 Monitor Subroutine Featuring Logical Operations

SECTION 18

HARDWORKING SUBROUTINES

Machine language programs take considerable effort to code and get working because they involve large numbers of instructions. On the other hand, they are the most machine efficient form of program. A good way to increase the power of a computer is to build a collection of machine language subroutines that can be combined in different ways to form executing programs. Some subroutines do commonly useful tasks which can be used in many programs. In writing such subroutines, it is important to keep them as general as possible. Avoid combining several processing tasks together into one subroutine, restricting it to fewer situations.

There are many ways to provide input data and locations for output data for subroutines. Input data can be left in processor registers. One example would be the Monitor subroutine at $FE93, described in Figure 14.2. The input and output result are transmitted in the accumulator. Another useful example is the 'time delay' subroutine of Figure 18.1, with input data left in register X. It passes time by executing an inner loop 256 times and repeating this the number of times specified by X. Place calls to this subroutine in a program to slow down the action, so you can actually see something happening. If X is in use at the point of the desired delay, then you must save its contents somewhere before calling 'time delay' and restore its contents afterwards. The location $00FD, which is wiped out by the call to 'time delay,' does not have to be initially set to any particular value before the call unless an exact time is required.

```
              ; TIME DELAY OF ABOUT 2265.X CYCLES

              ; CLEARS $FD

EA      TIME    NOP         ; EACH NOP ADDS 512 CYCLES TO THE LOOP

C6  FD          DEC  $FD    ; 256 TIMES, AFTER THE FIRST

DØ  FB          BNE  TIME

88              DEX         ; COUNTS EXECUTED LOOPS

DØ  F8          BNE  TIME

6Ø              RTS
```

Figure 18.1 Time Display Subroutine

Another way of passing data to subroutines is illustrated in Figure 18.2. The address and length of the 'scroll' field are placed in zero page locations. The LDA and STA instructions use an addressing mode called indirect indexed, in which the instruction identifies the zero page location of the address of the scroll field, and the contents of Y are added to form the effective address of a byte in the field. the scroll field address is a two-byte address which can be changed to relocate the scroll field anywhere in the display. The scroll subroutine does a big part of good listener's task. See if you can rewrite "good listener" to use the scroll subroutine. Don't forget to load appropriate values into locations ØØF8, ØØF9 and ØØFA.

You can experiment with the time delay subroutine by loading it and inserting the instructions

```
    LDX   #_____
```

```
    JSR   TIME
```

within the scroll subroutine's loop. Remember to subtract five from the relative addressing displacement in

```
    BNE   SHIFT
```

to adjust for the insertion. Adjust the immediate value loaded into X for a pleasing 'ripple' effect as you enter data.

```
                   ; SHIFT DISPLAY FIELD LEFT, STORE A INTO RIGHTMOST CHARACTER
                   ; FIELD ADDRESS AT $F9,$F8,LENGTH-1 AT $FA
                   ; AFFECTS Y
A0  00   SCROLL   LDY   #0          ; STRING POINTER
48                PHA               ; SAVE INSERT
C8       SHIFT    INY               ; I+1
B1  F8            LDA   ($F8),Y     ; MOVE (I+1) ST POSITION
88                DEY               ; I
91  F8            STA   ($F8),Y     ; TO (I)TH POSITION
C8                INY               ; ADVANCE I
C4  FA            CPY   $FA         ; LOOP TEST VALUE
D0  F5            BNE   SHIFT
68                PLA               ; RESTORE AND
91  F8            STA   ($F8),Y     ; INSERT INTO RIGHTMOST
60                RTS
```

Figure 18.2 Character Scroll

SECTION 19

TWO-WAY ARITHMETIC

Many computers can do decimal arithmetic directly, in addition to binary arithmetic. As you know, converting numbers between the binary and decimal number systems is not so easy. A program that receives decimal data and outputs decimal results, while doing all internal computations in binary, can have a lot of converting to do. Decimal arithmetic, though not as efficient as binary arithmetic, can be better in many situations as an alternative to conversions in both directions.

The 6502 processor has a switch that can be thrown to make add and subtract operations produce decimal, rather than binary results. In this mode of operation, byte operands which are valid BCD (Binary Coded Decimal) inputs are combined into valid BCD sums and differences, with the carry flag representing carries or borrows of ten. See Section 5 to review BCD. Although a ten's complement corresponding to the two's complement does exist for the representation of negative numbers, decimal arithmetic is usually done with positive numbers.

A decimal mode flag is the means of controlling the arithmetic mode of the processor and indicating which mode the processor is in. One-byte instructions set this flag to enter decimal mode and clear it to return to binary arithmetic. These instructions and the position of the decimal mode flag (D) in the P register which is pushed onto the stack by the PHP instruction, are shown in Figure 19.1

MNEMONIC	EXPLANATION	OPCODE	FLAGS
CLD	Clear Decimal Mode Flag	D8	D=0
SED	Set Decimal Mode Flag	F8	D=1

Figure 19.1 The Decimal Mode Flag

Decimal mode arithmetic can be explored by changing the first instruction in "total mystery" (original or your extended version) to an SED instruction. This makes the program execute in decimal mode. The Monitor clears the decimal mode flag as it begins execution, but care must be taken when entering the Monitor at other points that the decimal mode flag is cleared.

You might enjoy the program of Figure 19.2, which requires several of the subroutines you have seen in previous sections. The program converts decimal numbers to hexadecimal by using decimal arithmetic. Minor changes allow it to convert decimal numbers to hexadecimal by doing the same operations in binary arithmetic. A necessary multiplication is carried out by adding repeatedly; not the most efficient method, but simple enough to make the two-way arithmetic idea work. More efficient multiplication methods involving adding and shifting are spelled out in many references. The bibliography at the end of this manual will lead you to specific methods and coded subroutines.

```
; DECIMAL-HEX CONVERTER:
;
4000   D8          START    CLD              ; CLD OR SED
4001   A9 20       REPEAT   LDA  #$20        ; ASCII BLANK
4003   A0 00                LDY  #0          ; ZERO

4005   A2 07                LDX  #07
4007   9D C6 D0 CLEAR       STA  $D0C6,X     ; CLEAR ENTRY FIELD
400A   94 E0                STY  $E0,X       ; CLEAR ACCUMULATOR
400C   CA                   DEX
```

47

400D	10	F8			BPL	CLEAR		
400F	A2	E0		AOUT	LDX	#$E0	; INPUT ADDRESS	
4011	A0	40			LDY	#$40	; DISPLAY POSITION	
4013	A9	08			LDA	#$08		
4015	20	—	—		JSR	HEXTV	; ACCUMULATOR DISPLAY, FIG. 16.3	
4018	20	ED	FE		JSR	$FEED	; GET DIGIT	
401B	48				PHA		; SAVE ASCII	
401C	20	93	FE		JSR	$FE93	; STRIP TO DIGIT, FIG 14.2	
401F	10	03			BPL	DIGIT	; BRANCH IF DIGIT	
4021	68				PLA		; IF NOT, DISCARD IT	
4022	50	DD			BVC	REPEAT	;	
4024	48				PHA		; SAVE STRIPPED FOR ADD	
4025	A2	07			LDX	#7		
4027	EA				NOP		; F8	SED	- FOR HEX-DEC	*
4028	B5	E0		COPY	LDA	$E0,X	; COPY FOR MULTIPLY	
402A	95	E8			STA	$E8,X	;	
402C	CA				DEX			
402D	10	F9			BPL	COPY		
402F	A0	09		CNT	LDY	#9	; CHANGE #9 to #$F	- FOR HEX-DEC	*
4031	A2	07		MULT	LDX	#7	; MULTIPLY BY ADDING	
4033	20	—	—		JSR	ADDN	; FIGURE 15.4	
4036	88				DEY			
4037	D0	F8			BNE	MULT		
4039	68				PLA		; PULL STRIPPED DIGIT	
403A	A2	07			LDX	#7		
403C	EA				NOP		; D8	CLD	- FOR HEX-DEC	*
403D	EA	EA			NOPs		; C9 0A	CMP #$A	- FOR HEX-DEX	*
403F	EA	EA			NOPs		; 90 02	BCC	- FOR HEX-DEC	*
4041	EA	EA			NOPs		; 69 05	ADC #$5	- FOR HEX-DEC	*
4043	EA				NOP		; F8	SED	- FOR HEX-DEC	*
4044	20	—	—		JSR	ADD1	; ADD TO ACCUMULATOR, FIGURE 15.4	
4047	68				PLA		; PULL DIGIT ASCII	
4048	20	—	—		JSR	SCROLL	; ROLL INTO INPUT DISPLAY, FIG. 18.2	
404B	EA				NOP		; D8	CLD	- FOR HEX-DEC	*
404C	4C	0F	40		JMP	AOUT	; DISPLAY NEW ACCUMULATOR	

* Make these changes to convert this program to a Hexadecimal-Decimal Converter.

Figure 19.2 Decimal-Hexadecimal Converter

The machine code addresses in Figure 19.2 assume the program is loaded into memory beginning at location $4000. Actually it can be loaded anywhere in RAM, provided the absolute address in the last instruction is adjusted to the location of the instruction labeled AOUT. This one dependence on the load address of the program can be removed by replacing the last instruction with a conditional branch that is known to branch from the known value of its flag. This strategy was followed in the instruction shown at $401F.

Blanks occur in some absolute address positions in the program, corresponding to subroutine calls. The subroutines are those defined in this manual. You can load them anywhere in RAM and fill in their absolute addresses (with reversed bytes) to make a complete program.

Adjusting a program to its loading location is called relocation. Filling in addresses to reflect the location of other routines and data is called linking or resolving references. Many computer systems have a system program, called a loader, which loads, relocates and links the user's programs.

If you have trouble getting the routines to run please refer to Appendix K which contains completed listings.

SECTION 20

A SPARKLING FINISH

You have almost reached the end of this guide, but that is no reason to slow down in enabling your computer to do new useful and amusing things, through machine language programming.

We have explored almost every 6502 instruction, but there is an addressing mode we have not encountered. It is called indexed indirect. In this mode, the index register X selects an address from a block of reversed-byte addresses in the zero page. The starting byte of the block is designated by the second byte of the indexed instruction. In the symbolic assembler language form the indexed indirect mode is indicated by an operand of the form.

(zero-page-address,X)

The indexed indirect addressing mode is useful for accessing bytes in random order rather than in increasing or decreasing order. The program of Figure 20.1 illustrates the situation. Select a set of displayed cells scattered 'randomly' around on the screen. List them in a block of reversed-byte addresses, not in any particular order. Load the program somewhere beyond the block of addresses and load the 'time delay' subroutine (pg.45) along with it. Link the 'time delay' into your program. Place the number of addresses in location $0001 and adjust the timing constant for a pleasing effect.

In the sparkle program, the 'stars' stay off most of the time. To make a 'twinkle, twinkle, little star' version, reverse the star and blank characters $E8 and $20.

```
        ;
        ; SPARKLING FINISH
        ;
        ; LOAD A TIMING CONSTANT FOR THE 'ON' TIME IN $00
        ; LOAD N INTO $01
        ; PLACE ADDRESSES OF SPARKLE DISPLAY POINTS IN LOCATIONS
                $04$05 THROUGH 2N, 2N+1 IN REVERSED BYTE FORM
        ;

A6  01      BEGIN   LDX  1        ;
A9  E8      LOOP    LDA  #$E8     ; A 'STAR'
81  02              STA  ($02,X)  ; THE STAR APPEARS
8A                  TXA           ; SAVE FOR TIME
A6  00              LDX  0
20  — —             JSR  TIME     ; FLASH A STAR
AA                  TAX           ; RESTORE STAR POINTER
A9  20              LDA  #$20     ; BLANK
81  02              STA  ($02,X)  ; TURN STAR OFF
CA                  DEX           ;
CA                  DEX           ; INDEX NEXT STAR LOCATION
D0  ED              BNE  LOOP     ; FLASH NEXT STAR
F0  E9              BEQ  BEGIN    ; CYCLE THROUGH STARS AGAIN
```

Figure 20.1 A Sparkling Program

50

APPENDIX A

ASCII CHARACTER CODES

CODE	CHAR	CODE	CHAR	CODE	CHAR
ØØ	NUL	2B	+	56	V
Ø1	SOH	2C	,	57	W
Ø2	STX	2D	—	58	X
Ø3	ETX	2E	.	59	Y
Ø4	EOT	2F	/	5A	Z
Ø5	ENQ	3Ø	Ø	5B	[
Ø6	ACK	31	1	5C	/
Ø7	BEL	32	2	5D	]
Ø8	BS	33	3	5E	∧
Ø9	HT	34	4	5F	—
ØA	LF	35	5	6Ø	
ØB	VT	36	6	61	a
ØC	FF	37	7	62	b
ØD	CR	38	8	63	c
ØE	SO	39	9	64	d
ØF	SI	3A	:	65	e
1Ø	DLE	3B	;	66	f
11	DC1	3C	<	67	g
12	DC2	3D	=	68	h
13	DC3	3E	>	69	i
14	DC4	3F	?	6A	j
15	NAK	40	@	6B	k
16	SYN	41	A	6C	l
17	ETB	42	B	6D	m
18	CAN	43	C	6E	n
19	EM	44	D	6F	o
1A	SUB	45	E	7Ø	p
1B	ESC	46	F	71	q
1C	FS	47	G	72	r
1D	GS	48	H	73	s
1E	RS	49	I	74	t
1F	US	4A	J	75	u
2Ø	SP	4B	K	76	v
21	!	4C	L	77	w
22	"	4D	M	78	x
23	#	4E	N	79	y
24	$	4F	Ø	7A	z
25	%	50	P	7B	{
26	&	51	Q	7C	}
27	'	52	R	7D	¦
28	(	53	S	7E	÷
29	)	54	T	7F	DEL
2A	*	55	U		

APPENDIX B

65Ø2 MICROPROCESSOR ARCHITECTURE

DB7   DBØ

8 – BIT DATA BUS

ALU
8 BITS
BINARY/
DECIMAL

INSTRUCTION
DECODING
AND CYCLE
ENCODING

| A | X |
| Y | S |

PROGRAM COUNTER

8 -BIT ACCUMULATOR

8 -BIT INDEX REGISTERS

9 -BIT STACK POINTER

16 -BITS

7   Ø

| N | V | | B | D | I | Z | C |

TIMING
AND CONTROL
LOGIC

TIMING, PROCESSOR
CONTROL, AND
BUS SIGNALS

AB15   ABØ
ADDRESS BUS
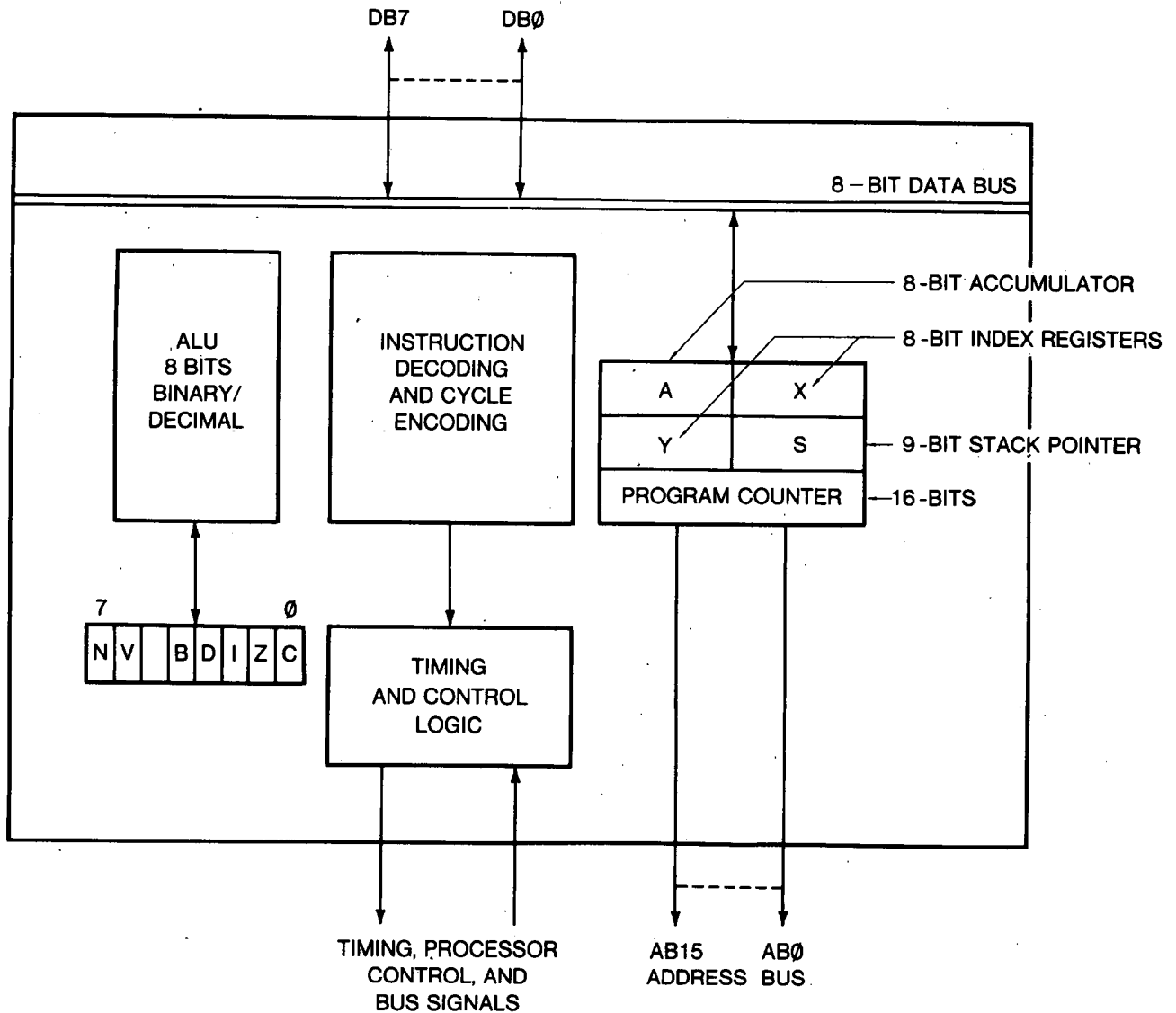
APPENDIX C

65Ø2 INSTRUCTION SET - MNEMONIC LIST

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive Or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift Right One Bit (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack

ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

APPENDIX D

65Ø2 INSTRUCTION SET - HEX LISTING

Dec	Hex		Instruction		Dec	Hex		Instruction
	ØØ	-	BRK		47	2F	-	Future Expansion
	Ø1	-	ORA-(Indirect,X)		48	30	-	BMI
	Ø2	-	Future Expansion		49	31	-	AND-(Indirect),Y
	Ø3	-	Future Expansion		50	32	-	Future Expansion
	Ø4	-	Future Expansion		51	33	-	Future Expansion
	Ø5	-	ORA-Zero Page		52	34	-	Future Expansion
	Ø6	-	ASL-Zero Page		53	35	-	AND-Zero Page,X
	Ø7	-	Future Expansion		54	36	-	ROL-Zero Page,X
	Ø8	-	PHP		55	37	-	Future Expansion
	Ø9	-	ORA-Immediate		56	38	-	SEC
10	ØA	-	ASL-Accumulator		57	39	-	AND-Absolute,Y
11	ØB	-	Future Expansion		58	3A	-	Future Expansion
12	ØC	-	Future Expansion		59	3B	-	Future Expansion
13	ØD	-	ORA-Absolute		60	3C	-	Future Expansion
14	ØE	-	ASL-Absolute		61	3D	-	AND-Aboslute,X
15	ØF	-	Future Expansion		62	3E	-	ROL-Absolute,X
16	10	-	BPL		63	3F	-	Future Expansion
17	11	-	ORA-(Indirect),Y		64	40	-	RTI
18	12	-	Future Expansion		65	41	-	EOR-(Indirect,X)
19	13	-	Future Expansion		66	42	-	Future Expansion
20	14	-	Future Expansion		67	43	-	Future Expansion
21	15	-	ORA-Zero Page,X		68	44	-	Future Expansion
22	16	-	ASL-Zero Page,X		69	45	-	EOR-Zero Page
23	17	-	Future Expansion		70	46	-	LSR-Zero Page
24	18	-	CLC		71	47	-	Future Expansion
25	19	-	ORA-Absolute,Y		72	48	-	PHA
26	1A	-	Future Expansion		73	49	-	EOR-Immediate
27	1B	-	Future Expansion		74	4A	-	LSR-Accumulator
28	1C	-	Future Expansion		75	4B	-	Future Expansion
29	1D	-	ORA-Absolute,X		76	4C	-	JMP-Absolute
30	1E	-	ASL-Absolute,X		77	4D	-	EOR-Absolute
31	1F	-	Future Expansion		78	4E	-	LSR-Absolute
32	20	-	JSR		79	4F	-	Future Expansion
33	21	-	AND-(Indirect,X)		80	50	-	BVC
34	22	-	Future Expansion		81	51	-	EOR-(Indirect),Y
35	23	-	Future Expansion		82	52	-	Future Expansion
36	24	-	BIT-Zero Page		83	53	-	Future Expansion
37	25	-	AND-Zero Page		84	54	-	Future Expansion
38	26	-	ROL-Zero Page		85	55	-	EOR-Zero Page,X
39	27	-	Future Expansion		86	56	-	LSR-Zero Page,X
40	28	-	PLP		87	57	-	Future Expansion
41	29	-	AND-Immediate		88	58	-	CLI
42	2A	-	ROL-Accumulator		89	59	-	EOR-Absolute,Y
43	2B	-	Future Expansion		90	5A	-	Future Expansion
44	2C	-	BIT-Absolute		91	5B	-	Future Expansion
45	2D	-	AND-Absolute		92	5C	-	Future Expansion
46	2E	-	ROL-Absolute		93	5D	-	EOR-Absolute,X

	Hex		Instruction		Hex		Instruction
94	5E	-	LSR-Absolute,X	149	95	-	STA-Zero Page,X
95	5F	-	Future Expansion	150	96	-	STX-Zero Page,Y
96	60	-	RTS	151	97	-	Future Expansion
97	61	-	ADC-(Indirect,X)	152	98	-	TYA
98	62	-	Future Expansion	153	99	-	STA-Absolute,Y
99	63	-	Future Expansion	154	9A	-	TXS
100	64	-	Future Expansion	155	9B	-	Future Expansion
101	65	-	ADC-Zero Page	156	9C	-	Future Expansion
102	66	-	ROR-Zero Page	157	9D	-	STA-Absolute,X
103	67	-	Future Expansion	158	9E	-	Future Expansion
104	68	-	PLA	159	9F	-	Future Expansion
105	69	-	ADC-Immediate	160	A0	-	LDY-Immediate
106	6A	-	ROR-Accumulator	161	A1	-	LDA-(Indirect,X)
107	6B	-	Future Expansion	162	A2	-	LDX-Immediate
108	6C	-	JMP-Indirect	163	A3	-	Future Expansion
109	6D	-	ADC-Absolute	164	A4	-	LDY-Zero Page
110	6E	-	ROR-Absolute	165	A5	-	LDA-Zero Page
111	6F	-	Future Expansion	166	A6	-	LDX-Zero Page
112	70	-	BVS	167	A7	-	Future Expansion
113	71	-	ADC-(Indirect),Y	168	A8	-	TAY
114	72	-	Future Expansion	169	A9	-	LDA-Immediate
115	73	-	Future Expansion	170	AA	-	TAX
116	74	-	Future Expansion	171	AB	-	Future Expansion
117	75	-	ADC-Zero Page,X	172	AC	-	LDY-Absolute
118	76	-	ROR-Zero Page,X	173	AD	-	LDA-Absolute
119	77	-	Future Expansion	174	AE	-	LDX-Absolute
120	78	-	SEI	175	AF	-	Future Expansion
121	79	-	ADC-Absolute,Y	176	B0	-	BCS
122	7A	-	Future Expansion	177	B1	-	LDA-(Indirect),Y
123	7B	-	Future Expansion	178	B2	-	Future Expansion
124	7C	-	Future Expansion	179	B3	-	Future Expansion
125	7D	-	ADC-Absolute,X	180	B4	-	LDY-Zero Page,X
126	7E	-	ROR-Absolute,X	181	B5	-	LDA-Zero Page,X
127	7F	-	Future Expansion	182	B6	-	LDX-Zero Page,Y
128	80	-	Future Expansion	183	B7	-	Future Expansion
129	81	-	STA-(Indirect,X)	184	B8	-	CLV
130	82	-	Future Expansion	185	B9	-	LDA-Absolute,Y
131	83	-	Future Expansion	186	BA	-	TSX
132	84	-	STY-Zero Page	187	BB	-	Future Expansion
133	85	-	STA-Zero Page	188	BC	-	LDY-Absolute,X
134	86	-	STX-Zero Page	189	BD	-	LDA-Absolute,X
135	87	-	Future Expansion	190	BE	-	LDX-Absolute,Y
136	88	-	DEY	191	BF	-	Future Expansion
137	89	-	Future Expansion	192	C0	-	CPY-Immediate
138	8A	-	TXA	193	C1	-	CMP-(Indirect,X)
139	8B	-	Future Expansion	194	C2	-	Future Expansion
140	8C	-	STY-Absolute	195	C3	-	Future Expansion
141	8D	-	STA-Absolute	196	C4	-	CPY-Zero Page
142	8E	-	STX-Absolute	197	C5	-	CMP-Zero Page
143	8F	-	Future Expansion	198	C6	-	DEC-Zero Page
144	90	-	BCC	199	C7	-	Future Expansion
145	91	-	STA-(Indirect),Y	200	C8	-	INY
146	92	-	Future Expansion	201	C9	-	CMP-Immediate
147	93	-	Future Expansion	202	CA	-	DEX
148	94	-	STY-Zero Page,X	203	CB	-	Future Expansion

6502 INSTRUCTION SET - HEX LISTING (CONTINUED)

CC	-	CPY-Absolute
CD	-	CMP-Absolute
CE	-	DEC-Absolute
CF	-	Future Expansion
D0	-	BNE
D1	-	CMP-(Indirect),Y
D2	-	Future Expansion
D3	-	Future Expansion
D4	-	Future Expansion
D5	-	CMP-Zero Page,X
D6	-	DEC-Zero Page,X
D7	-	Future Expansion
D8	-	CLD
D9	-	CMP-Absolute,Y
DA	-	Future Expansion
DB	-	Future Expansion
DC	-	Future Expansion
DD	-	CMP-Absolute,X
DE	-	DEC-Absolute,X
DF	-	Future Expansion
E0	-	CPX-Immediate
E1	-	SBC-(Indirect,X)
E2	-	Future Expansion
E3	-	Future Expansion
E4	-	CPX-Zero Page
E5	-	SBC-Zero Page

E6	-	INC-Zero Page
E7	-	Future Expansion
E8	-	INX
E9	-	SBC-Immediate
EA	-	NOP
EB	-	Future Expansion
EC	-	CPX-Absolute
ED	-	SBC-Absolute
EE	-	INC-Absolute
EF	-	Future Expansion
F0	-	BEQ
F1	-	SBC-(Indirect),Y
F2	-	Future Expansion
F3	-	Future Expansion
F4	-	Future Expansion
F5	-	SBC-Zero Page,X
F6	-	INC-Zero Page,X
F7	-	Future Expansion
F8	-	SED
F9	-	SBC-Absolute,Y
FA	-	Future Expansion
FB	-	Future Expansion
FC	-	Future Expansion
FD	-	SBC-Absolute,X
FE	-	INC-Absolute,X
FF	-	Future Expansion

APPENDIX E

6502 DISASSEMBLY TABLE

OPCODE TABLE

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

LSD—Least Significant Digit
MSD—Most Significant Digit

APPENDIX F

SPECIAL SYMBOLS

A	Accumulator
X,Y	Index Registers
M	Memory
P	Processor Status Register
S	Stack Pointer
✔	Change
-	No Change
+	Add
∧	Logic AND
−	Subtract
⊻	Logic Exclusive Or
↑	Transfer from Stack
↓	Transfer to Stack
→	Transfer to
←	Transfer to
V	Logical OR
PC	Program Counter
PCH	Program Counter High
PCL	Program Counter Low
OPER	OPERAND
#	IMMEDIATE ADDRESSING MODE
$	Indicates a Hex Value

APPENDIX G

COMPLETE INSTRUCTION LIST WITH OPCODES

ADC

ADD MEMORY TO ACCUMULATOR WITH CARRY

Operation: A + M + C → A, C

N	Z	C	I	D	V
✔	✔	✔	—	—	✔

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Immediate	ADC	# Oper	69 *105*
Zero Page	ADC	Oper	65 *101*
Zero Page, X	ADC	Oper, X	75 *117*
Absolute	ADC	Oper	6D *109*
Absolute, X	ADC	Oper, X	7D *125*
Absolute, Y	ADC	Oper, Y	79 *121*
(Indirect, X)	ADC	(Oper, X)	61 *97*
(Indirect), Y	ADC	(Oper), Y	71 *113*

AND

"AND" MEMORY WITH ACCUMULATOR

Logical AND to the Accumulator

Operation: A ∧ M → A

N	Z	C	I	D	V
✔	✔	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Immediate	AND	# Oper	29 *41*
Zero Page	AND	Oper	25 *37*
Zero Page, X	AND	Oper, X	35 *53*
Absolute	AND	Oper	2D *45*
Absolute, X	AND	Oper, X	3D *61*
Absolute, Y	AND	Oper, Y	39 *57*
(Indirect, X)	AND	(Oper, X)	21 *33*
(Indirect), Y	AND	(Oper), Y	31 *49*

ASL

ASL SHIFT LEFT ONE BIT (MEMORY OR ACCUMULATOR)

Operation: C ← [7|6|5|4|3|2|1|0] ← 0

	N	Z	C	I	D	V
	✓	✓	✓	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Accumulator	ASL	A	ØA *10*
Zero Page	ASL	Oper	Ø6 *6*
Zero Page, X	ASL	Oper, X	16 *22*
Absolute	ASL	Oper	ØE *14*
Absolute, X	ASL	Oper, X	1E *30*

BCC

BCC BRANCH ON CARRY CLEAR

Operation: Branch on C = Ø

	N	Z	C	I	D	V
	—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Relative	BCC	Oper	9Ø *144*

BCS

BCS BRANCH ON CARRY SET

Operation: Branch on C = 1

	N	Z	C	I	D	V
	—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Relative	BCS	Oper	BØ *176*

BEQ

BEQ BRANCH ON RESULT ZERO

Operation: Branch on Z = 1

	N	Z	C	I	D	V
	—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Relative	BEQ	Oper	FØ *246*

BIT

BIT TEST BITS IN MEMORY WITH ACCUMULATOR

Operation: $A \wedge M$, $M_7 \rightarrow N$, $M_6 \rightarrow V$

Bit 6 and 7 are transferred to the status Register.	N	Z	C	I	D	V
If the result of $A \wedge M$ is zero then $Z = 1$, otherwise $\emptyset$.	M_7	✔	—	—	—	M_6

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Zero Page	BIT	Oper	24 *36*
Absolute	BIT	Oper	2C *45*

BMI

BMI BRANCH ON RESULT MINUS

Operation: Branch on $N = 1$

N	Z	C	I	D	V
—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Relative	BMI	Oper	3∅ *48*

BNE

BNE BRANCH ON RESULT NOT ZERO

Operation: Branch on $Z = \emptyset$

N	Z	C	I	D	V
—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Relative	BNE	Oper	D∅ *208*

BPL

BPL BRANCH ON RESULT PLUS

Operation: Branch on $N = \emptyset$

N	Z	C	I	D	V
—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Relative	BPL	Oper	1∅ *16*

BRK

BRK FORCE BREAK

Operation: Forced Interrupt PC + 2 ↓ P ↓

N	Z	C	I	D	V
—	—	—	·1·	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	BRK	00 ○

A BRK command cannot be masked by setting I.

BVC

BVC BRANCH ON OVERFLOW CLEAR

Operation: Branch on V = O

N	Z	C	I	D	V
—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Relative	BVC Oper	50 *80*

BVS

BVS BRANCH ON OVERFLOW SET

Operation: Branch on V = 1

N	Z	C	I	D	V
—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Relative	BVS Oper	70 *112*

CLC

CLC CLEAR CARRY FLAG

Operation: $0 \rightarrow C$

N	Z	C	I	D	V
—	—	0	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	CLC	18 *24*

CLD

CLD CLEAR DECIMAL MODE

Operation: $0 \rightarrow D$

N	Z	C	I	D	V
—	—	—	—	0	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	CLD	D8 *216*

63

CLI

CLI CLEAR INTERRUPT DISABLE BIT

Operation: $0 \rightarrow I$

	N	Z	C	I	D	V
	—	—	—	0	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	CLI	58 *88*

CLV

CLV CLEAR OVERFLOW FLAG

Operation: $0 \rightarrow V$

	N	Z	C	I	D	V
	—	—	—	—	—	0

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	CLV	B8 *184*

CMP

CMP COMPARE MEMORY AND ACCUMULATOR

Operation: $A - M$

	N	Z	C	I	D	V
	✔	✔	✔	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Immediate	CMP	# Oper	C9 *201*
Zero Page	CMP	Oper	C5 *197*
Zero Page, X	CMP	Oper, X	D5 *213*
Absolute	CMP	Oper	CD *205*
Absolute, X	CMP	Oper, X	DD *221*
Absolute, Y	CMP	Oper, Y	D9 *217*
(Indirect, X)	CMP	(Oper, X)	C1 *193*
(Indirect), Y	CMP	(Oper), Y	D1 *209*

CPX

CPX COMPARE MEMORY AND INDEX X

Operation: $X - M$

	N	Z	C	I	D	V
	✔	✔	✔	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Immediate	CPX	# Oper	E0 *224*
Zero Page	CPX	Oper	E4 *228*
Absolute	CPX	Oper	EC *236*

CPY

CPY COMPARE MEMORY AND INDEX Y

Operation: Y − M

	N	Z	C	I	D	V
	✔	✔	✔	−	−	−

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Immediate	CPY	# Oper	C0 *192*
Zero Page	CPY	Oper	C4 *196*
Absolute	CPY	Oper	CC *204*

DEC

DEC DECREMENT MEMORY BY ONE

Operation: M − 1 → M

	N	Z	C	I	D	V
	✔	✔	−	−	−	−

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Zero Page	DEC	Oper	C6 *198*
Zero Page, X	DEC	Oper, X	D6 *214*
Absolute	DEC	Oper	CE *206*
Absolute, X	DEC	Oper, X	DE *222*

DEX

DEX DECREMENT INDEX X BY ONE

Operation: X − 1 → X

	N	Z	C	I	D	V
	✔	✔	−	−	−	−

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	DEX	CA *202*

DEY

DEY DECREMENT INDEX Y BY ONE

Operation: Y − 1 → Y

	N	Z	C	I	D	V
	✔	✔	−	−	−	−

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	DEY	88 *136*

EOR

EOR "EXCLUSIVE-OR" MEMORY WITH ACCUMULATOR

Operation: A $\veebar$ M → A

N	Z	C	I	D	V
✔	✔	—	—	—	—

ADDRESSING MODE		ASSEMBLY LANGUAGE FORM	OP CODE	
Immediate	EOR	# Oper	49	73
Zero Page	EOR	Oper	45	69
Zero Page, X	EOR	Oper, X	55	85
Absolute	EOR	Oper	4D	77
Absolute, X	EOR	Oper, X	5D	93
Absolute, Y	EOR	Oper, Y	59	89
(Indirect, X)	EOR	(Oper, X)	41	65
(Indirect), Y	EOR	(Oper), Y	51	81

INC

INC INCREMENT MEMORY BY ONE

Operation: M + 1 → M

N	Z	C	I	D	V
✔	✔	—	—	—	—

ADDRESSING MODE		ASSEMBLY LANGUAGE FORM	OP CODE	
Zero Page	INC	Oper	E6	230
Zero Page, X	INC	Oper, X	F6	246
Absolute	INC	Oper	EE	238
Absolute, X	INC	Oper, X	FE	254

INX

INX INCREMENT INDEX X BY ONE

Operation: X + 1 → X

N	Z	C	I	D	V
✔	✔	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE	
Implied	INX	E8	232

INY

INY INCREMENT INDEX Y BY ONE

Operation: Y + 1 → Y

N	Z	C	I	D	V
✔	✔	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE	
Implied	INY	C8	200

JMP

JMP JUMP TO NEW LOCATION

Operation: (PC + 1) → PCL

(PC + 2) → PCH

N	Z	C	I	D	V
—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Absolute	JMP	Oper	4C *76*
Indirect	JMP	(Oper)	6C *108*

JSR

JSR JUMP TO NEW LOCATION SAVING RETURN ADDRESS

Operation: PC + 2 ↓ , (PC + 1) PCL

(PC + 2) PCH

N	Z	C	I	D	V
—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Absolute	JSR	Oper	2Ø *32*

LDA

LDA LOAD ACCUMULATOR WITH MEMORY

Operation: M → A

N	Z	C	I	D	V
✔	✔	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Immediate	LDA	# Oper	A9 *169*
Zero Page	LDA	Oper	A5 *165*
Zero Page, X	LDA	Oper, X	B5 *181*
Absolute	LDA	Oper	AD *173*
Absolute, X	LDA	Oper, X	BD *189*
Absolute, Y	LDA	Oper, Y	B9 *185*
(Indirect, X)	LDA	(Oper, X)	A1 *161*
(Indirect), Y	LDA	(Oper), Y	B1 *177*

LDX

LDX LOAD INDEX X WITH MEMORY

Operation: M → X

N	Z	C	I	D	V
✔	✔	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Immediate	LDX	# Oper	A2 *162*
Zero Page	LDX	Oper	A6 *166*
Zero Page, Y	LDX	Oper, Y	B6 *182*
Absolute	LDX	Oper	AE *174*
Absolute, Y	LDX	Oper, Y	BE *190*

LDY

LDY LOAD INDEX Y WITH MEMORY

Operation: M → Y

	N	Z	C	I	D	V
	✓	✓	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Immediate	LDY	# Oper	A0 *160*
Zero Page	LDY	Oper	A4 *164*
Zero Page, X	LDY	Oper, X	B4 *180*
Absolute	LDY	Oper	AC *172*
Absolute, X	LDY	Oper, X	BC *188*

LSR

LSR SHIFT RIGHT ONE BIT (MEMORY OR ACCUMULATOR)

Operation: 0 → $\boxed{7\,6\,5\,4\,3\,2\,1\,0}$ → C

	N	Z	C	I	D	V
	0	✓	✓	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Accumulator	LSR	A	4A *74*
Zero Page	LSR	Oper	46 *70*
Zero Page, X	LSR	Oper, X	56 *86*
Absolute	LSR	Oper	4E *78*
Absolute, X	LSR	Oper, X	5E *94*

NOP

NOP NO OPERATION

Operation: No Operation (2 cycles)

	N	Z	C	I	D	V
	—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	NOP	EA *234*

ORA

ORA "OR" MEMORY WITH ACCUMULATOR

Operation: A V M → A

	N	Z	C	I	D	V
	✓	✓	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE	
Immediate	ORA	# Oper	09	9
Zero Page	ORA	Oper	05	5
Zero Page, X	ORA	Oper, X	15	21
Absolute	ORA	Oper	0D	13
Absolute, X	ORA	Oper, X	1D	29
Absolute, Y	ORA	Oper, Y	19	25
(Indirect, X)	ORA	(Oper, X)	01	1
(Indirect), Y	ORA	(Oper), Y	11	17

PHA

PHA PUSH ACCUMULATOR ON STACK

Operation: A ↓

	N	Z	C	I	D	V
	—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE	
Implied	PHA	48	72

PHP

PHP PUSH PROCESSOR STATUS ON STACK

Operation: P ↓

	N	Z	C	I	D	V
	—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE	
Implied	PHP	08	8

PLA

PLA PULL ACCUMULATOR FROM STACK

Operation: A ↑

	N	Z	C	I	D	V
	✓	✓	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE	
Implied	PLA	68	104

PLP

PLP PULL PROCESSOR STATUS FROM STACK

Operation: P ↑

	N	Z	C	I	D	V
				From Stack		

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	PLP	28 *40*

ROL

ROL ROTATE ONE BIT LEFT (MEMORY OR ACCUMULATOR)

Operation: ⌐7|6|5|4|3|2|1|0 ← C⌐

N	Z	C	I	D	V
✓	✓	✓	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Accumulator	ROL	A	2A *42*
Zero Page	ROL	Oper	26 *38*
Zero Page, X	ROL	Oper, X	36 *54*
Absolute	ROL	Oper	2E *46*
Absolute, X	ROL	Oper, X	3E *62*

ROR

ROR ROTATE ONE BIT RIGHT (MEMORY OR ACCUMULATOR)

Operation: ⌐C→7|6|5|4|3|2|1|0⌐

N	Z	C	I	D	V
✓	✓	✓	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Accumulator	ROR	A	6A *106*
Zero Page	ROR	Oper	66 *102*
Zero Page, X	ROR	Oper, X	76 *118*
Absolute	ROR	Oper	6E *110*
Absolute, X	ROR	Oper, X	7E *126*

RTI

RTI RETURN FROM INTERRUPT

Operation; P↑ PC↑

	N	Z	C	I	D	V
				From Stack		

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	RTI	40 *64*

RTS

RTS RETURN FROM SUBROUTINE

Operation: PC↑ , PC + 1 → PC

N	Z	C	I	D	V
—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE	
Implied	RTS	60	*96*

SBC

SBC SUBTRACT MEMORY FROM ACCUMULATOR WITH BORROW

Operation: A − M−$\overline{C}$ → A

N	Z	C	I	D	V
✔	✔	✔	—	—	✔

Note: $\overline{C}$ = Borrow

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE	
Immediate	SBC	# Oper	E9	*233*
Zero Page	SBC	Oper	E5	*229*
Zero Page, X	SBC	Oper, X	F5	*245*
Absolute	SBC	Oper	ED	*237*
Absolute, X	SBC	Oper, X	FD	*253*
Absolute, Y	SBC	Oper, Y	F9	*249*
(Indirect, X)	SBC	(Oper, X)	E1	*225*
(Indirect), Y	SBC	(Oper), Y	F1	*241*

SEC

SEC SET CARRY FLAG

Operation: 1 → C

N	Z	C	I	D	V
—	—	1	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE	
Implied	SEC	38	*56*

SED

SED SET DECIMAL MODE

Operation: 1 → D

N	Z	C	I	D	V
—	—	—	—	1	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE	
Implied	SED	F8	*248*

SEI

SEI SET INTERRUPT DISABLE STATUS

Operation: 1 → I

	N	Z	C	I	D	V
	—	—	—	1	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Implied	SEI		78 *120*

STA

STA STORE ACCUMULATOR IN MEMORY

Operation: A → M

	N	Z	C	I	D	V
	—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Zero Page	STA	Oper	85 *133*
Zero Page, X	STA	Oper, X	95 *149*
Absolute	STA	Oper	8D *141*
Absolute, X	STA	Oper, X	9D *157*
Absolute, Y	STA	Oper, Y	99 *153*
(Indirect, X)	STA	(Oper, X)	81 *129*
(Indirect), Y	STA	(Oper), Y	91 *145*

STX

STX STORE INDEX X IN MEMORY

Operation: X → M

	N	Z	C	I	D	V
	—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Zero Page	STX	Oper	86 *134*
Zero Page, Y	STX	Oper, Y	96 *150*
Absolute	STX	Oper	8E *142*

STY

STY STORE INDEX Y IN MEMORY

Operation: Y → M

	N	Z	C	I	D	V
	—	—	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM		OP CODE
Zero Page	STY	Oper	84 *132*
Zero Page, X	STY	Oper, X	94 *148*
Absolute	STY	Oper	8C *140*

TAX

TAX TRANSFER ACCUMULATOR TO INDEX X

Operation: A → X

N	Z	C	I	D	V
✓	✓	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	TAX	AA *170*

TAY

TAY TRANSFER ACCUMULATOR TO INDEX Y

Operation: A → Y

N	Z	C	I	D	V
✓	✓	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	TAY	A8 *168*

TYA

TYA TRANSFER INDEX Y TO ACCUMULATOR

Operation: Y → A

N	Z	C	I	D	V
✓	✓	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	TYA	98 *152*

TSX

TSX TRANSFER STACK POINTER TO INDEX X

Operation: S → X

N	Z	C	I	D	V
✓	✓	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	TSX	BA *186*

TXA

TXA TRANSFER INDEX X TO ACCUMULATOR

Operation: X → A

N	Z	C	I	D	V
✓	✓	—	—	—	—

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	TXA	8A *138*

0422

TXS

TSX TRANSFER INDEX X TO STACK POINTER

Operation: $X \rightarrow S$ N   Z   C   I   D   V

                                                                     —   —   —   —   —   —

ADDRESSING MODE	ASSEMBLY LANGUAGE FORM	OP CODE
Implied	TXS	9A *154*

APPENDIX H

OSI 65V MONITOR MOD 2 LISTING

ASSEM

```
 10 0000                    ; OSI 65U PROM MONITOR MOD 2
 20 0000                    ;
 30 0000                    FLAG=$FB
 40 0000                    DAT=$FC
 50 0000                    PNTL=$FE
 60 0000                    PNTH=$FF
 70 0000                    ;
 80 FE00                              *=$FE00
 90 FE00  A228     VM       LDX   *$28          INITIALIZATION
100 FE02  9A                TXS
110 FE03  D8                CLD
120 FE04  AD06FB            LDA   $FB06
130 FE07  A9FF              LDA   #$FF
140 FE09  8D05FB            STA   $FB05
150 FE0C  A2D8              LDX   #$D8
160 FE0E  A9D0              LDA   #$D0
170 FE10  85FF              STA   PNTH
180 FE12  A900              LDA   #0
190 FE14  85FE              STA   PNTL
200 FE16  85FB              STA   FLAG
210 FE18  A8                TAY
220 FE19  A920              LDA   #'
230 FE1B  91FE     VM1      STA   (PNTL),Y
240 FE1D  C8                INY
250 FE1E  D0FB              BNE   VM1
260 FE20  E6FF              INC   PNTH
270 FE22  E4FF              CPX   PNTH
280 FE24  D0F5              BNE   VM1
290 FE26  84FF              STY   PNTH
300 FE28  F019              BEQ   IN
310 FE2A             ;
320 FE2A  20E9FE   ADDR     JSR   INPUT         ADDRESS MODE
330 FE2D  C92F              CMP   #'/
340 FE2F  F01E              BEQ   DATA
350 FE31  C947              CMP   #'G
360 FE33  F017              BEQ   GO
370 FE35  C94C              CMP   #'L
380 FE37  F043              BEQ   LOAD
390 FE39  2093FE            JSR   LEGAL
400 FE3C  30EC              BMI   ADDR
410 FE3E  A202              LDX   #2
420 FE40  20DAFE            JSR   ROLL
430 FE43  B1FE     IN       LDA   (PNTL),Y
440 FE45  85FC              STA   DAT
450 FE47  20ACFE            JSR   OUTPUT
460 FE4A  D0DE              BNE   ADDR
```

```
470 FE4C
480 FE4C  6CFE00  GO      JMP     (PNTL)
490 FE4F                   ;
500 FE47  20E9FE  DATA    JSR     INPUT     DATA MODE
510 FE52  C92E            CMP     #'.
520 FE54  F0D4            BEQ     ADDR
530 FE56  C90D            CMP     #$D
540 FE58  D00F            BNE     DAT4
550 FE5A  E6FE            INC     PNTL
560 FE5C  D002            BNE     DAT3
570 FE5E  E6FF            INC     PNTH
580 FE60  A000    DAT3    LDY     #0
590 FE62  B1FE            LDA     (PNTL),Y
600 FE64  85FC            STA     DAT
610 FE66  4C77FE          JMP     INNER
620 FE69  2093FE  DAT4    JSR     LEGAL
630 FE6C  30E1            BMI     DATA
640 FE6E  A200            LDX     #0
650 FE70  20DAFE          JSR     ROLL
660 FE73  A5FC            LDA     DAT
670 FE75  91FE            STA     (PNTL),Y
680 FE77  20ACFE  INNER   JSR     OUTPUT
690 FE7A  D0D3            BNE     DATA
700 FE7C                  ;
710 FE7C  85FB    LOAD    STA     FLAG      KICK INPUT DEVICE FLAG
720 FE7E  F0CF            BEQ     DATA
730 FE80                  ;
740 FE80  AD00FC  OTHER   LDA     $FC00     SERIAL INPUT SUB.
750 FE83  4A              LSR     A         (FOR AUDIO CASSETTE)
760 FE84  90FA            BCC     OTHER
770 FE86  AD01FC          LDA     $FC01
780 FE89  EAEAEA          NOP     NOP       NOP
790 FE8C  297F            AND     #$7F
800 FE8E  60              RTS
810 FE8F                  ;
820 FE8F  00              .BYTE   0,0,0,0   EXCESS ROOM
820 FE90  00
820 FE91  00
820 FE92  00
830 FE93                  ;
840 FE93  C930    LEGAL   CMP     #'0       IGNORE NON HEX CHAR.
850 FE95  3012            BMI     FAIL
860 FE97  C93A            CMP     #':
870 FE99  300B            BMI     OK
880 FE9B  C941            CMP     #'A
890 FE9D  300A            BMI     FAIL
900 FE9F  C947            CMP     #'G
910 FEA1  1006            BPL     FAIL
920 FEA3  38              SEC
930 FEA4  E907            SBC     #7
940 FEA6  290F    OK      AND     #$F
950 FEA8  60              RTS
960 FEA9  A980    FAIL    LDA     #$80
970 FEAB  60              RTS
980 FEAC                  ;
```

76

```
  990 FEAC A203      OUTPUT  LDX   #3           OUTPUT LLLL DD
 1000 FEAE A000              LDY   #0           ONTO SCREEN
 1010 FEB0 B5FC      OUI     LDA   DAT, X
 1020 FEB2 4A                LSR   A
 1030 FEB3 4A                LSR   A
 1040 FEB4 4A                LSR   A
 1050 FEB5 4A                LSR   A
 1060 FEB6 20CAFE            JSR   DIGIT
 1070 FEB9 B5FC              LDA   DAT,X
 1080 FEBB 20CAFE            JSR   DIGIT
 1090 FEBE CA                DEX
 1100 FEBF 10EF              BPL   OUI
 1110 FEC1 A920              LDA   #'
 1120 FEC3 8DCAD0            STA   $D0CA
 1130 FEC6 8DCBD0            STA   $D0CB
 1140 FEC9 60                RTS
 1150 FECA             ;
 1160 FECA 290F       DIGIT   AND   #$F          OUTPUT 1 DIGIT TO SCREEN
 1170 FECC 0930               ORA   #$30
 1180 FECE C93A               CMP   #$3A
 1190 FED0 3003               BMI   HA1
 1200 FED2 18                 CLC
 1210 FED3 6907               ADC   #7
 1220 FED5 99C6D0     HA1     STA   $D0C6,Y
 1230 FED8 C8                 INY
 1240 FED9 60                 RTS
 1250 FEDA              ;
 1260 FEDA A004       ROLL    LDY   #4           MOVE LSD IN AC TO
 1270 FEDC 0A                 ASL   A            LSD IN 2 BYTE NUM.
 1280 FEDD 0A                 ASL   A
 1290 FEDE 0A                 ASL   A
 1300 FEDF 0A                 ASL   A
 1310 FEE0 2A         R01     ROL   A
 1320 FEE1 36FC               ROL   DAT,X
 1330 FEE3 36FD               ROL   DAT+1,X
 1340 FEE5 88                 DEY
 1350 FEE6 D0F8               BNE   R01
 1360 FEE8 60                 RTS
 1370 FEE9 A5FB               LDA   FLAG         CASSETTE IN?
 1380 FEEB D091               BNE   $FE7E        YES-GO DO ACIA INPUT
 1390 FEED 4C00FD             JMP   $FD00        NO-GO POLL KB
 1400 FEF0 A9FF       KBTEST  LDA   #$FF         KB TEST SUBR.
 1410 FEF2 8D00DF             STA   $DF00
 1420 FEF5 AD00DF             LDA   $DF00
 1430 FEF8 60                 RTS
 1440 FEF9 EA                 NOP
 1450 FEFA 3001               .WORD $130         NMI VECTOR
 1460 FEFC 00FE               .WORD $FE00        RESET VECTOR
 1470 FEFE C001               WORD  $1C0         IRQ VECTOR
 1480 FF00                    .END
```

APPENDIX I

BIBLIOGRAPHY

1.* How to Program Microcomputers
   by William Barden
   Howard W. Sams & Company, Inc.
   Indianapolis, IN 46268

2. 6502 Software Gourmet Guide and Cookbook
   by Robert Findley
   SCELBI Publications
   20 Hurlbut Street
   Elmwood, CT 06110

3. The First Book of KIM

4. Programming a Microcomputer: 6502
   by Caxton C. Foster
   Addison Wesley Publishing Company, Inc.
   Reading, MA 01867

5. 6502 Assembly Language Programming
   by Lance Leventhal
   Osborne/McGraw-Hill

6. MCS6500 Microcomputer Family Programming Manual
   MOS Technology, Inc.
   950 Rittenhouse Road
   Norristown, PA 19401

7. MICRO: The 6502 Journal
   P.O. Box 6502
   Chelmsford, MA 01824

8. SY6500/MCS6500 Microcomputer Family Hardware Manual
   Synertek
   3050 Coronado Drive
   Santa Clara, CA 95051

9. Programming the 6502 (Second Edition)
   by Rodney Zaks
   Sybex
   2344 Sixth Street
   Berkeley, CA 94710

10. 6502 Applications Book
   by Rodney Zaks
   Sybex
   2344 Sixth Street
   Berkeley, CA 94710

11. 65Ø2 Games
      by Rodney Zaks
      Sybex
      2344 Sixth Street
      Berkeley, CA 94710

12. Programming & Interfacing The 65Ø2, With Experiments
      by Marvin L. De Jong
      Howard W. Sams & Co., Inc.
      4300 West 62nd Street
      Indianapolis, IN 46268

* Available from OSI

APPENDIX J

65V MONITOR COMMAND SUMMARY

The OS-65V Monitor responds to the following key:

Ø-9,A-F	Hex digits
.	Change to Address Mode
/	Change to Data Mode
G	Go to address shown on screen and execute code there
RETURN	Increment address (only in Data Mode)
L	Transfer control to audio cassette. This command enters the Data Mode, ignores the keyboard and listens only to the cassette port (if present). To transfer control back to the keyboard, press reset or load $ØØFD with a $ØØ via tape.

APPENDIX K

TWO-WAY ARITHMETIC LISTINGS

The following listings show one possible way to implement the Decimal-Hex and Hex-Decimal Converter programs from page 47. The four relocatable subroutines are included. The initial address of $4000 was arbitrarily chosen as a convenient location for disk systems (cassette systems would typically use $1000).

DECIMAL-HEX CONVERTER

4000	D8	CLD	
4001	A920	LDA	#$20
4003	A000	LDY	#$00
4005	A207	LDX	#$07
4007	9DC6D0	STA	$D0C6,X
400A	94E0	STY	$E0,X
400C	CA	DEX	
400D	10F8	BPL	$4007
400F	A2E0	LDX	#$E0
4011	A040	LDY	#$40
4013	A908	LDA	#$08
4015	200041	JSR	$4100
4018	20EDFE	JSR	$FEED
401B	48	PHA	
401C	2093FE	JSR	$FE93
401F	1003	BPL	$4024
4021	68	PLA	
4022	50DD	BVC	$4001
4024	48	PHA	
4025	A207	LDX	#$07
4027	EA	NOP	
4028	B5E0	LDA	$E0,X
402A	95E8	STA	$E8,X
402C	CA	DEX	
402D	10F9	BPL	$4028
402F	A009	LDY	#$09
4031	A207	LDX	#$07
4033	200042	JSR	$4200
4036	88	DEY	
4037	D0F8	BNE	$4031
4039	68	PLA	
403A	A207	LDX	#$07
403C	EA	NOP	
403D	EA	NOP	
403E	EA	NOP	
403F	EA	NOP	
4040	EA	NOP	
4041	EA	NOP	
4042	EA	NOP	
4043	EA	NOP	
4044	200043	JSR	$4300
4047	68	PLA	
4048	200044	JSR	$4400
404B	EA	NOP	
404C	4C0F40	JMP	$400F

The Decimal-Hex Converter program uses four subroutines: HEXTV, ADDN, ADD1 and SCROLL. The listings of these that follow are located to coincide with the calling routines on the previous page.

HEXTV

4100	85FF	STA	$FF
4102	B500	LDA	$00,X
4104	4A	LSR	A
4105	4A	LSR	A
4106	4A	LSR	A
4107	4A	LSR	A
4108	20CAFE	JSR	$FECA
410B	B500	LDA	$00,X
410D	20CAFE	JSR	$FECA
4110	EA	NOP	
4111	E8	INX	
4112	C6FF	DEC	$FF
4114	D0EC	BNE	$4102
4116	60	RTS	

ADDN

4200	18	CLC	
4201	B5E0	LDA	$E0,X
4203	75E8	ADC	$E8,X
4205	95E0	STA	$E0,X
4207	CA	DEX	
4208	10F7	BPL	$4201
420A	60	RTS	

ADD1

4300	18	CLC	
4301	75E0	ADC	$E0,X
4303	95E0	STA	$E0,X
4305	CA	DEX	
4306	A900	LDA	#$00
4308	B0F7	BCS	$4301
430A	60	RTS	

SCROLL

4400	A000	LDY	#$00
4402	48	PHA	
4403	C8	INY	
4404	B1F8	LDA	($F8),Y
4406	88	DEY	
4407	91F8	STA	($F8),Y
4409	C8	INY	
440A	C4FA	CPY	$FA
440C	D0F5	BNE	$4403
440E	68	PLA	
440F	91F8	STA	($F8),Y
4411	60	RTS	

Make these changes to the Decimal-Hex Converter code on page 81 to complete the transition into Hex-Decimal Converter. The subroutines will remain unchanged.

4027	F8	SED	
402F	A00F	LDY	#$0F
403C	D8	CLD	
403D	C90A	CMP	#$0A
403F	9002	BCC	$4043
4041	6905	ADC	#$05
4043	F8	SED	
404B	D8	CLD	

INDEX