A BYTE BOOK

>$16.95

TOP-DOWN ASSEMBLY LANGUAGE PROGRAMMING for your VIC-20™ AND COMMODORE 64™

KEN SKIER

8 K Expansion RAM required for VIC-20

Top-Down Assembly Language Programming for Your

VIC-20™

and

Commodore 64™

Ken Skier

McGraw-Hill Book Company

New York St. Louis San Francisco Auckland
Bogotá Hamburg Johannesburg London Madrid
Mexico Montreal New Delhi Panama Paris
São Paulo Singapore Sydney Tokyo Toronto

Library of Congress Cataloging in Publication Data

Skier, Ken.
Top-down assembly language programming for your VIC-20 and Commodore 64.

Includes index.
1. VIC-20 (Computer)—Programming. 2. Commodore 64 (Computer)—Programming. 3. Assembler language (Computer program language) 4. Structured programming. I. Title.
QA76.8.V5S55 1984 001.64'2 84-758
ISBN 0-07-057864-8

The author of the programs provided with this book has carefully reviewed them to ensure their performance in accordance with the specifications described in the book. Neither the author nor McGraw-Hill, however, makes any warranties concerning the programs and neither assumes any responsibility or liability of any kind for errors in the programs, or for the consequences of any such errors. The programs are the sole property of the author and have been registered with the United States Copyright Office.

1234567890 HAL/HAL 8987654

ISBN 0-07-057864-8 PBK.

The editors for this book were Stephen G. Guty and Esther Gelatt and the production supervisor was Sally Fliess.

Printed and bound by Halliday Lithograph.

Table of Contents

Introduction

Objectives

Sometimes I hear people talk about how smart computers have become. But computers aren't smart: programmers are. Programmers make microprocessors act like calculators, moon landers, or income tax preparers. Programmers must be smart, because by themselves microprocessors can't do much of anything.

Sound programming, then, is fundamental to successful computer use. With this principle in mind, this book has two objectives: first, to introduce newcomers to some of the techniques, terminology, and power of assembly-language programming in general, and of the 6502 in particular; and second, to present a set of software tools to use in developing assembly-language programs for the 6502.

Chapter 1 takes you on a quick tour of your computer's hardware and software; Chapters 2 thru 4 comprise a short course in assembly-language programming for those readers new to the subject. The rest of the book presents source listings, object code, and assembler listings for programs that you may enter into your computer and run.

Programmers have long sought to develop small and fast programs with the unfortunate result that occasionally code has been written that is unreadable (and even unworkable) simply because a programmer wanted to save a few bytes or a few cycles. In certain instances when memory space is particularly tight or execution time is critical, readability is sacrificed for performance. But today the average programmer is not forced to make this choice. Of course, all other things being equal, I, too, value programs that are quick and compact.

But how often are all other things equal?

While developing the programs that appear in this book, I had a number of objectives, most of them more important than the speed or size of a block of code. I designed these programs to be:

Useful: No program is presented simply to demonstrate a particular programming technique. All of the programs in this book were written because I needed certain things done — usually something I didn't want to be bothered with doing myself. The monitor monitors, the disassembler disassembles, and the text editor lets me enter and edit text strings. These programs earn their keep.

Easy to Use: Simply by glancing at the screen you can tell which program is running and what mode it is in. When a program needs information, it asks you for it and allows you to correct mistakes you might make while answering. This software doesn't require you to remember the addresses of programs or of variables. Functions are mapped to individual keys, and you can assign functions to keys in any way that makes sense to you.

Readable: A beginning 6502 programmer should be able to understand the workings of every program in this book. The labels and comments in the listings were carefully chosen to reveal the purpose of each variable, subroutine, and line of code. I am writing first and foremost for you, the reader, not for the 6502.

Portable: The book's software runs on a Commodore 64 or VIC-20 computer. With proper initialization of the System Data Block, it should run on *any* 6502-based computer equipped with a keyboard and a memory-mapped, character-graphics video display.

Compatible: These routines are very good neighbors. As long as the other software in your system does not use the fourth 4 K block of memory (hexadecimal memory locations 3000 thru 3FFF), there should be no conflict between your software and the software in this book. In particular, most of the software in this book preserves the zero page, so your software may use the zero page as much as you like, and you won't be bothered with having to save and restore it before and after calls to the software presented herein.

Expandable: The programs in this book are highly modular, and you may extend or restructure them to meet your individual needs. System-specific subroutines are called indirectly, so that other subroutines may be substituted for them, and most values are treated as variables, rather than as constants hard-wired into the code. There are no monolithic programs in this book; they're all subroutines and may be combined in many ways to build powerful new structures.

Compact: I know that every personal computer has exactly the same available memory: too little. I also know ways to write a program in ten or twenty percent less space. But if doing so required sacrificing readability, portability, or expandability, I did not do so. In many cases I feared that to save a byte, I might lose a reader's clear understanding of how a program works. I considered that too great a price to pay for a somewhat smaller program.

Note: If you have a VIC-20, you must have 8 K of expansion RAM to run the software in this book.

Fast: Assuming that the above objectives have been met, the software in this book has been developed to operate as quickly as possible. But in any trade-off between speed and the other objectives, speed loses. A fast program that you can't understand holds little value. None of the programs in this book are likely to make you complain about how long you have to wait. I can't tell if I'm waiting an extra millisecond. Can you?

So go ahead. Read. Program. Enjoy!

Chapter 1:

Your Computer

Your Commodore 64 or VIC-20 is a very powerful computer. But to take advantage of that power, we must understand how it works. So before we begin programming it, let's take a quick tour of your computer.

The 6502 Microprocessor

We'll start with the 6502 microprocessor, the component in your system that actually computes. By itself, the 6502 can't do much. It has three *registers* (special memory areas for storing the data upon which the program is operating), called A, X, and Y, which can each hold a number in the range of 0 to 255. Different registers have different capabilities. For example, if a number is in A (the accumulator), the 6502 can add to it, or subtract from it, any value up to 255. But if a number is in the X register or the Y register, the 6502 can only increment or decrement that number (ie: add or subtract one from it).

The 6502 can also set one register equal to the value of another register, and it can store the contents of any register anywhere in memory, or load any register from any location in memory. Thus, although the 6502 can only operate on one number at a time, it can operate on many numbers, just by loading registers from various locations in memory, operating on the registers, and then storing the results of those operations back into memory.

Types of Memory

You may have heard that a computer stores information as a series of ones and zeros. This is because the computer's memory is simply an elaborate array of switches, and an individual switch can have only two states: closed or open. These two states may also be expressed as on and off, or as one and zero.

Not all memory switches are the same. Some, in what is called ROM (read-only memory), are hard-wired into your computer's circuitry and cannot be changed except by physically replacing the ROM circuits containing those switches. Others, in what is called RAM (random-access memory) or programmable memory, can be changed by the processor. The 6502 can open or close any of the switches, called bits (binary digits), in its programmable memory, and later on read what it "wrote" into that memory. Figure 1.1 shows how the processor has access to read-only memory and programmable memory.

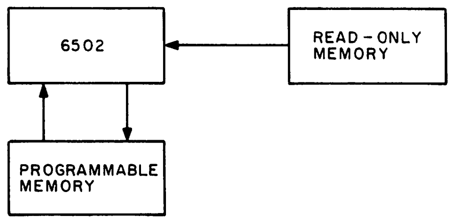

Figure 1.1: *How the 6502 interacts with memory. The arrows indicate the flow of data.*

A third kind of memory is set by some external device, not by the 6502. Such memory switches are called *input ports*, and may be connected to keyboards, terminals, burglar alarms — virtually anything that can generate an electrical signal. The 6502 perceives these externally generated signals by reading the appropriate input ports.

Yet another kind of memory switch, called an *output port*, generates a high or a low voltage on some particular wire depending on whether the 6502 sets a given memory switch to a one or a zero. One or more of these output ports can enable the 6502 to "talk" to the outside world.

Now don't jump up and think I'm going to show you how to synthesize speech in this book. "Talk" is just my way of anthropomorphizing the 6502. It will happen elsewhere in this book, when the 6502 "sees," "remembers," and "knows" what to do. Of course the 6502 doesn't see, remember, or know anything, but I often find it helpful to put myself in its place. That way I can better understand how a program will run, or why a program doesn't run, and I *do* see, remember, and know things.

But don't take such verbs too literally. The 6502 doesn't talk. It causes signals to be generated that may be sensed by other devices, such as cassette recorders, printers, disk drives — and yes, even speech synthesizers. But not in this book.

Some peripheral devices are actually connected to both an input and an output port. Examples of these devices are cassette tape machines and floppy-disk drives, which are mass-storage or secondary-storage devices. Figure 1.2 summarizes the processor's access to memory and to peripheral devices.

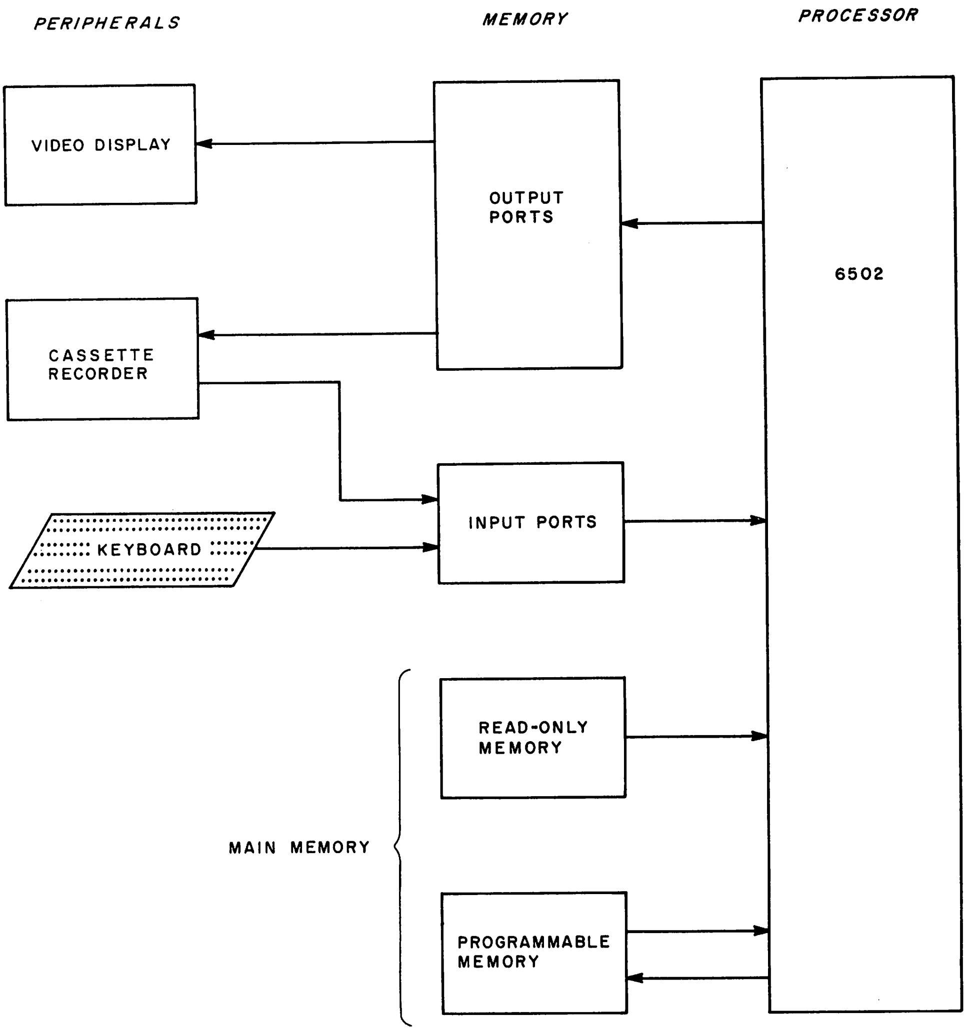

Figure 1.2: *A summary of the 6502 microprocessor's access to data in main memory and through I/O (input and output) ports. The arrows indicate the flow of data.*

A video screen connected to your computer looks like memory to the 6502, so the 6502 can read from and write to the screen. The keyboard is scanned by I/O (input/output) ports that are decoded to look like any other programmable memory address, so the 6502 can look at the keyboard just by looking at a particular place in memory. Thus, the 6502 can interact directly with memory only, but because all I/O devices are mapped to addresses in memory, the 6502 can interact with the user. See figure 1.3.

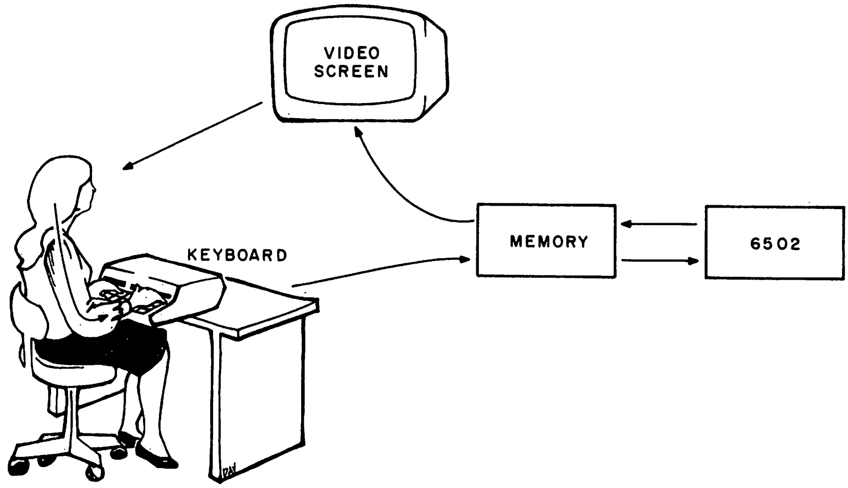

Figure 1.3: *How the 6502 interacts with the user. Arrows indicate the flow of data.*

The Operating System

Thus far we have discussed your machine's hardware. But the Commodore 64 and VIC-20 computers feature more than hardware. For example, these computers have an operating system (stored in ROM) which includes the I/O software routines that are needed to use the screen and the keyboard. We are not particularly concerned with how these subroutines work, but we depend on them to be there when we need them.

There are many other subroutines in your computer's operating system. The *Programmer's Reference Guide* for your system describes these subroutines in detail. All of this means power for you, the programmer. The more you know about your computer, the more you can make it do. Because the software in this book was developed to run on a number of systems, I chose not to use routines available in your machine's ROM, no matter how powerful they might be, unless I could be sure that they would be available in the operating systems of many popular personal computers. In other words, the software in this book does not take full advantage of the power in your operating system. But the software you write, which need only run on your system, should exploit to the fullest the power of your computer's ROM routines.

BASIC

One of the most important features of your computer is the BASIC interpreter in ROM. This interpreter is a program that enables your computer to understand commands given in BASIC. Your system's documentation tells you what commands are legal in the particular dialect of BASIC implemented on your machine. BASIC is an easy language to learn and you can do a lot with it.

However, each BASIC statement must be analyzed (parsed) by the BASIC interpreter before the computer can take any action. So a BASIC program is not very fast—certainly not as fast as a comparable program written in *6502 code*.

6502 Code

The central processor is the computer's heart. The Commodore 64 and VIC-20 computers use the 6502 microprocessor. Every microprocessor has a certain *instruction set*, or group of instructions, which the microprocessor can execute. These instructions are at a much lower level than the BASIC commands with which you may be familiar. For example, in BASIC you can have a single line in a program to PRINT "HELLO." It would take a sequence of many 6502 instructions to perform the same function.

A given sequence of microprocessor instructions will run on any computer featuring that microprocessor. Thus, if you write a program consisting of 6502 instructions to perform some function, that program should run on any 6502-based computer. It won't run on an 8080-based computer, a Z80-based computer, or a 6800-based computer, but it should run on an Apple, a PET, an Atari, an OSI, or any other system built around a 6502. 6502 programs can also run much faster than equivalent programs written in BASIC and can be smaller than BASIC programs. The programs presented in this book are all written in 6502 code, and require only half of the memory available on a computer containing 8,000 bytes of programmable memory, thus leaving more than enough room for your own programs.

Chapter 2:

Introduction to Assembler

Ever watch a juggler or a good juggling team? The balls, pins, or whatever are in the air in such intricate patterns that you can hardly follow them, let alone duplicate the performance yourself. It's beautiful, but not magic; just an application of some simple rules. I've learned to juggle recently, and although I'm still a rank beginner, I've taught my two hands to keep three balls moving through the air. Yet neither hand knows very much. A hand will toss a ball into the air, and then it will catch a ball. The other hand will toss a ball into the air, and then it will catch a ball. That's all. My hands perform only two operations: toss and catch. Yet with those two primitive operations I can put on a pleasant little performance.

Assembly-language programming is not so different from juggling. Like juggling, programming enables you to put on an impressive or baffling performance. In its simplest terms, juggling is nothing more than taking something from one place and putting it someplace else. The same thing is true of the central processor: the 6502 takes something from one place and puts it someplace else.

In fact, programming the 6502 is easier than juggling in several ways. First, the 6502 is obviously much faster than even the most skillful juggler. In the time it takes me to pick up a ball with one hand and place that ball somewhere else, the 6502 can get something from one place and put it someplace else hundreds of thousands of times. Sleight of hand requires quickness, and the 6502 is quick.

The 6502 even gives me a helping hand. When I try to juggle, I must keep the balls moving with nothing but my two hands. But my home computer has three hands (registers A, X, and Y in the 6502) and thousands of pockets (8,000 bytes or more of programmable memory).

A byte is 8 bits of data that may be loaded together into a register. A register holds 1 byte. Each location in memory holds 1 byte. The 6502 can affect only 1 byte in one operation. But because the 6502 can perform hundreds of thousands of operations each second, it can affect hundreds of thousands of bytes each second.

Binary

In the final analysis, any value is stored within the computer as a series of bits. If we wish, we may specify a byte by its bit pattern: such a representation uses only ones and zeroes, and is called binary. For example, the number 25 in binary is 00011001.

In binary, each bit indicates the presence or absence of some value. Each bit represents twice as much value, or significance, as the bit to its right, so the right-most bit is the least significant, and the left-most bit is the most significant. Table 2.1 gives the significance of each bit in an 8-bit byte:

Table 2.1: *Bit significance in an 8-bit byte.*

Bit Number:	b7	b6	b5	b4	b3	b2	b1	b0
Bit Significance:	128	64	32	16	8	4	2	1

The right-most bit (called bit 0) tells us whether we have a one in our byte. The bit to its left (bit 1) tells us whether we have a two; the bit to *its* left tells us whether we have a four...and the leftmost bit (bit 7) tells us whether we have a 128 in our byte.

To determine the bit pattern for a given value — say, 25 — determine first what powers of two must be added to equal your value. For instance, $25 = 16 + 8 + 1$, so 25 in binary is 00011001.

Twenty-five can be expressed in other ways as well. Rather than specify every number as a pattern of eight ones and zeros, we often express numbers in hexadecimal representation.

Hexadecimal

Unlike binary, which requires a group of eight characters to represent an 8-bit value, hexadecimal notation allows us to represent an 8-bit value with a group of only two characters. These characters are not limited to 0 and 1, but may include any digit from 0 to 9, and any letter from "A" to "F." That gives us a set of sixteen characters, which is just right because we want to represent numbers in base 16.

(Hexadecimal stands for 16: hex for six, and decimal for ten. Six plus ten equals sixteen.)

To represent a byte in hexadecimal notation, divide the 8-bit byte into two 4-bit units (sometimes called *nybbles*). Each of these 4-bit units has a value of from 0 to 15 (decimal), which we express with a single hexadecimal digit. A decimal 10 is a hexadecimal $A. (The dollar sign indicates that a number is in hexadecimal representation.) Table 2.2 gives the conversions of decimal to hexadecimal for decimal numbers 0 thru 15.

Table 2.2: *Hexadecimal character set.*

Hexadecimal Character		Decimal Equivalent
$0	=	0
$1	=	1
$2	=	2
$3	=	3
$4	=	4
$5	=	5
$6	=	6
$7	=	7
$8	=	8
$9	=	9
$A	=	10
$B	=	11
$C	=	12
$D	=	13
$E	=	14
$F	=	15

Appendix A1, *Hexadecimal Conversion Table*, shows the hexadecimal representation of every number from 0 to 255 decimal.

In this book, object code, the only code that the machine can execute directly, will generally be presented in hexadecimal, and a thorough understanding of hexadecimal will help you to interpret instructions and follow some of the 6502's actions. Even the sketchiest understanding of hexadecimal math, however, should be sufficient for you to follow and use the programs in this book.

ASCII Characters

Instead of a number from 0 to 255, an 8-bit byte can be used to represent an upper or lower case letter of the alphabet, a punctuation mark, or a printer-control character such as a carriage return. A string of such bytes may represent a word, a message, or even a complete document. Appendix A2, *ASCII Character Codes*, gives the hexadecimal value for any ASCII character. ASCII stands for *A*merican *S*tandard *C*ode for *I*nformation *I*nterchange, and is the closest thing the industry has to a standard set of character codes. If you want to store the letter "A" in some location in memory, you can see from Appendix A2 that you must store a $41 in that location.

Whether a given byte is interpreted as a number, an ASCII character, or something else depends entirely on the program using that byte. Just as beauty is in the eye and mind of the beholder, so is the meaning of a given byte determined by the program that sees and uses it.

The Instruction Cycle

A microprocessor such as the 6502 can't do anything without being told. It only knows 151 instructions, called opcodes (operation codes). Each opcode is 1 byte long. An opcode may command the 6502 to take something from one register and to put it someplace in memory, to load some register with the contents of some location in memory, or to perform some other equally simple operation. See Appendix A4 for a list of opcodes for the 6502 microprocessor.

What do 6502s do all day? They work while programmers play. The 6502 gets an opcode, performs the specified operation, gets the next opcode, performs the specified operation, gets the next opcode, performs the...

You get the picture.

How does the 6502 know where to find the next opcode? The 6502 has a 16-bit register called the PC (program counter). The PC holds the address of some location in memory. When the 6502 starts its instruction cycle, it gets the opcode stored at the memory location specified by the PC. Then it performs the operation specified by that opcode. When it has executed that instruction, it makes the PC point to the next opcode and starts on a new instruction cycle by getting the opcode whose address is now in the PC.

Figure 2.1 shows a flowchart for the instruction cycle of the 6502 microprocessor.

"That's it? That's all the 6502 does?" you ask.

That's it. But with the right program in memory, we can make the 6502 dance.

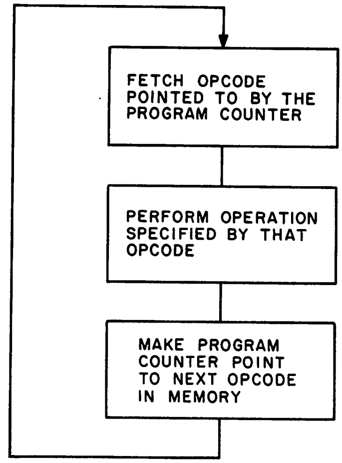

Figure 2.1: *The 6502 instruction cycle.*

Machine Language

A machine-language program is nothing more than a series of machine-language instructions stored in memory. If the PC in the 6502 can be made to hold the address of the start of your program, then we say that the PC is *pointing* to your program. When the 6502 starts its instruction cycle, it will *fetch* the first opcode in your program, and then perform the operation specified by that opcode. At this point, we say that your program is *running*.

Each machine-language instruction is stored in memory as a 1-byte opcode, which may be followed by 1 or 2 bytes of operand. Thus, a 6502 machine-language program might be "A9 05 20 02 04 A2 F5 60."

Just a bunch of numbers! (Hexadecimal numbers, in this case.) But it is exactly these numbers that the machine understands; hence the term, machine language.

Assemblers

Machine language is easy to read — if you're a machine. But programmers are people. So programming tools called assemblers have been developed, which take more readable assembly-language *source code* as input and produce *listings* and *object code* as output. The listing is the assembler's output intended for a human reader. The object code is a series of 6502 machine-language instructions intended to be stored in memory and executed by the 6502.

For each chapter in this book that presents a program, there is an appendix at the back of the book containing an assembler listing and a hexdump of the same program. The assembler listing includes both source and object code, making it easy for you to read the program; the hexdump shows you what the object code for that program actually looks like in your computer's memory. Figure 2.2 shows how an assembler is used to produce an assembler listing for the programmer and object code for the processor.

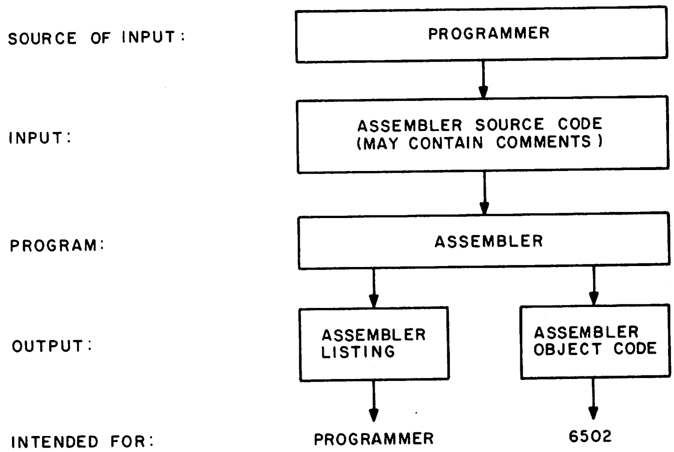

Figure 2.2: *From programmer to object code. The assembler takes source code as input and produces an assembler listing and object code as output.*

The programs in this book have all been produced on the OSI 6500 Assembler/Editor, running under the OSI 65-D Disk Operating System, on an OSI C-IP machine with 24 K bytes of programmable memory and one 5-inch floppy disk. The object code, however, runs on any Commodore 64, or any VIC-20 with at least 8 K of expansion RAM. (Incidentally, the source code in each chapter of this book should fit into the workspace of a computer with much less than 24 K bytes of user memory, if you delete many of the comments. But then, of course, your listings will be a lot less readable.)

But you don't write a listing; an assembler produces a listing. What you write is assembly-language source code.

Source Code

An assembly-language source program consists of one or more lines of assembly-language source code. A line of assembly-language source code consists of up to four fields:

LABEL MNEMONIC OPERAND COMMENT

The mnemonic, required in all cases, is a group of three letters chosen to suggest the function of a given machine-language instruction. For example, the mnemonic *LDA* stands for *LoaD Accumulator*. *LDX* stands for *LoaD X* register. *TXA* means *Transfer the X* register to the *Accumulator*. 6502 mnemonics are not nearly as meaningful as BASIC commands, but they're a big improvement over the machine-language opcodes. See Appendix A3 for a list of 6502 mnemonics.

Some operations require an operand field. For example, the operation *load accumulator* requires an operand, because the line of source code must specify what you wish to load into the accumulator.

The label and comment fields are optional. A label lets you operate on some location in memory by a name that you have assigned to it. Comments are not included in the object code that will be assembled from your program, but they make your source code and your listings much more meaningful to a human reader. When you write a program, even if no one but yourself will ever read it, try to choose your labels and comments so that someone else can understand the purpose of each part of the program. Such careful documentation will save you a lot of time weeks or months down the road, when you might otherwise reread your program and have no idea why you included some unlabeled, uncommented line of source code.

Loading a Register

Let's write a simple program to load a register with a number — say, to load the accumulator with the number "10." Since we want to load the accumulator, we'll use the LDA instruction. (If we wanted to load the X register, we would use the LDX instruction, and if we wanted to load the Y register, we'd use LDY.) We know what mnemonic to write into our first line of source code. But a glance at Appendix A6, *6502 Opcodes by Mnemonic and Addressing Mode*, shows that LDA has many addressing modes. What operand shall we write into this line of source code?

We know that we want to load the accumulator with a "10," and not with any other number, so we can use the immediate addressing mode to load a "10" directly into the accumulator. We'll use a "#" sign to indicate the immediate mode:

Example I

```
LDA #10
```

Example 1 is a legitimate line of source code containing only two fields: a mnemonic and an operand. The mnemonic, LDA, means "load the accumulator." But load it with what? The operand tells us what to load into the accumulator. The "#" sign specifies that this operation is to take place in the immediate mode, which means we want to load the accumulator with a constant to be found in this line of source code, rather than with data or a variable to be found in some location in memory. Then the operand specifies the constant to be loaded into the accumulator, in this case "10."

Constants

A constant is any value that is known by the programmer and "hard-wired" into the code. A constant does not change during the execution of a program. If a value changes during the execution of a program, then it is a variable, and one or more memory locations must be allocated to hold the current value of each variable.

There are several kinds of constants. Any number is a constant. The number "7," for example, is a constant: a seven now will still be a seven this afternoon. A character is another kind of constant: the letter "A" will still be the letter "A" tomorrow. But a variable, such as one called FUEL, will change during the course of a program (such as a lunar lander simulation), so it is not a constant.

In Example 1, note that the "#" sign is the only punctuation in the operand field. In the absence of special punctuation marks (such as the dollar sign indicating a hexadecimal number and the apostrophe indicating an ASCII character representation), any numbers given in this book are in decimal.

What object code will be assembled from this line of source code? Let's hand-assemble it and see. Appendix A6 shows us that the opcode for load accumulator, immediate mode, is $A9. So the first byte of object code for this instruction will be $A9. The second byte must specify what the 6502 should load into the accumulator. We want to load register A with a decimal 10, which is $0A. So the object code assembled from Example 1 is: A9 0A.

When these 2 bytes of object code are executed by the 6502, it will result in the accumulator holding a value of $0A, or decimal 10. In effect, we've just told a juggler: put a "10" in your right hand.

What if we wanted to load the accumulator with the letter "M," rather than with a number? We'd still use LDA to load the accumulator, and we'd still use the immediate mode of addressing, specifying in the operand the constant to be loaded into the accumulator. Either of the following two lines of source code will work:

Example 2

LDA #' M

or

LDA #$4D

In each line of source code above, the mnemonic and the "#" sign tell us we're loading the accumulator in the immediate mode — ie: with a constant. The operand following the "#" sign specifies the constant. An apostrophe indicates that an ASCII character follows, whereas a "$" sign indicates that a hexadecimal number follows. Appendix A2 shows that an ASCII "M" = $4D; they are simply two representations of the same bit pattern. So the two lines of source code above are equivalent; they will both assemble into the same object code: A9 4D.

Which of the two lines of source code is more readable? If a constant will be used in a program as an ASCII character, then represent it in your source code as an ASCII character.

Storing the Register

Now let's say we want to store the contents of the accumulator someplace in memory. Every location in memory has a unique address (just like houses do), ranging from $0000 to $FFFF. Suppose we decide to store the contents of the accumulator at memory location $020C. We could do it with the following line of source code:

Example 3

STA $020C

Example 3 will assemble into these 3 bytes of machine language: 8D 0C 02.

According to the Appendix A6, the 6502 opcode for "store accumulator, absolute mode" (STA) is $8D.

When the 6502 fetches the opcode "8D," it knows that it must store the contents of the accumulator at the address specified by the next 2 bytes. This is why it is called absolute mode. Absolute mode is used when specifying an exact memory location in an instruction.

In the example above, that address seems wrong. It looks like the machine-language operand is specifying address $0C02, because the bytes are in that order: "0C" followed by "02." But we want to operate an address $020C. Is something wrong here?

Low Byte First

You and I might think something is wrong when the address $020C is written as an "0C" followed by an "02" but you and I are people. We don't think like the 6502. When you and I write a number, we tend to write the most significant digit first and the least significant digit last. But the 6502 doesn't work that way. When the 6502 interprets two sequential bytes as an address, the first byte must contain the less significant part of the address (the "low byte"), and the second byte must contain the more significant part of the address (the "high byte"). All addressing modes that require a 2-byte operand require that the 2 bytes be in this order: less significant byte first, followed by the more significant byte.

However, not all addressing modes require a 2-byte operand.

Zero-Page Addressing

Memory is divided into pages, where a page is a block of 256 contiguous addresses. The page from $0000 to $00FF is called the zero page, because all addresses in this page have a high byte of zero. The zero-page addressing mode takes advantage of this fact. Source code assembled using the zero-page addressing mode requires only 1 byte in the operand, because the opcode specifies the zero page mode of addressing, and the high byte of the operand is unnecessary because it is understood to be zero. Thus, you can specify an address in the zero page by the absolute or by the zero-page addressing mode, but the zero-page mode will let you do it using one less byte.

If you want to use some location in the zero page to hold a number, you might decide to use location $00F4. We could write:

Example 4

```
STA $00F4
```

or

```
STA $F4
```

We could then assemble either line of source code using the absolute addressing mode: 8D F4 00. Or we could assemble either line of source code using the zero-page mode: 85 F4.

The opcode "85" means "store accumulator, zero page." Where in the zero page? At location $F4 in the zero page, the same location whose absolute address is $00F4.

Symbolic Expressions

Let's say you want to copy the 3 bytes at memory locations \$0200, \$0201, and \$0202 to \$0300, \$0301, and \$0302, respectively. We could write these lines of source code:

Example 5

```
LDA $0200
STA $0300
LDA $0201
STA $0301
LDA $0202
STA $0302
```

This alternately loads a byte into the accumulator, then stores the contents of the accumulator into another byte in memory. Note that loading a register from a location in memory changes the register, but leaves the contents of the memory location unchanged.

Or we could write the following code, which refers to addresses as symbolic expressions:

Example 6

```
1   ORIGIN = $0200
2   DEST   = $0300
3   LDA    ORIGIN
4   STA    DEST
5   LDA    ORIGIN + 1
6   STA    DEST + 1
7   LDA    ORIGIN + 2
8   STA    DEST + 2
```

In Example 6, lines 1 and 2 are assembler directives, which equate the labels "ORIGIN" and "DEST" with the addresses \$0200 and \$0300, respectively. Other lines of source code following these *equates* may then refer to these addresses by their labels, or refer to any address as a symbolic expression consisting of labels and, optionally, constants and arithmetic operators. The source code above will cause an assembler to generate exactly the same object code as the source code in Example 5, but Example 6, whose operands consist of symbolic expressions, is much more readable than Example 5, whose operands are given in hexadecimal.

Some Exercises

1) Write the 6502 instructions necessary to load the accumulator with the value 127, to load the X register with the letter "r," and to load the Y register with the contents of address $BO92.

2) Write the 6502 instructions necessary to copy the byte at address $0043 to the address $0092.

Chapter 3:

Loops and Subroutines

Indexed Addressing

Although readable, Example 6 is not very efficient, because it requires two lines of source code to move each byte. If we want to move 50 or 100 bytes must we then write 100 or 200 lines of source code?

Indexed addressing comes in quite handily here. Instead of specifying the absolute or zero-page address on which an operation is to be performed, we can specify a *base address* and an *index* register. The 6502 will add the value of the specified index registers to the base address, thereby determining the address on which the operation is to be performed. Thus, if we want to move 9 bytes from an origin to a destination, we could do it in the following manner, using the indexed addressing mode with X as the index register:

Example 7

```
        ORIGIN = $0200
        DEST   = $0300

INIT    LDX #0           Initialize X register to zero, so we'll start
                         with the first byte in the block.
GET     LDA ORIGIN,X     Get Xth byte in origin block.
PUT     STA DEST,X       Put it into the Xth position in the
                         destination block.
ADJUST  INX              Adjust X for next byte by incrementing
                         (adding 1) to the X register.
```

```
TEST        CPX #9          Done 9 bytes yet?
BRANCH      BNE GET         If not, go back and get next byte...
```

We will use Example 7 in the following sections to introduce several new instructions and addressing modes. Example 7 includes six lines of source code to move 9 contiguous bytes of data. If we tried to move 9 bytes of data with the techniques used in Examples 5 and 6, it would have taken eighteen lines of source code. So with indexed addressing, we've saved ourselves twelve lines of code. But how do these lines work? The lines are labeled so we can look at them one-by-one.

The instruction labeled INIT loads the X register in the immediate mode with the value zero. After executing the line INIT, the 6502 has a value of zero in the X register. We don't know anything about what's in the other registers.

GET loads the accumulator with the Xth byte above the address labeled ORIGIN. The first time the 6502 encounters this line, the X register will hold a value of zero, so the 6502 will load the accumulator with the zeroth byte above the address labeled ORIGIN (ie: it will load the accumulator with the contents of the memory location ORIGIN).

In any line of source code, a comma in the operand indicates that the operation to be performed shall use an indexed addressing mode. A comma followed by an "X" indicates that the X register will be the index register for an instruction, whereas a comma followed by a "Y" indicates that the Y register will be the index for an instruction. There are a number of indexed addressing modes. Two of these are absolute indexed and zero-page indexed. The line GET in Example 7 uses the absolute indexed addressing mode if ORIGIN is above the zero page; if ORIGIN is in the zero page then the line labeled GET can be assembled using the zero-page indexed addressing mode. Zero-page indexed addressing, like zero-page addressing, requires only 1 byte in the operand.

In zero-page indexed and in absolute indexed addressing, the operand field specifies a base address. The 6502 will operate on an address it determines by adding to the base address the value of the specified index register (X or Y). Only if the specified index register has a value of zero will the 6502 operate on the base address itself; in all other cases the 6502 will operate on some address higher in memory.

So we've loaded the accumulator with the byte at ORIGIN. Now the 6502 reaches the line labeled PUT in Example 7. This line tells the 6502 to store the accumulator in the Xth byte above DEST. We haven't done anything to change X since the line INIT set it to zero, so X still holds a value of zero. Therefore, the 6502 will store the contents of the accumulator in the zeroth byte above DEST (ie: in DEST itself).

At this point, we have succeeded in moving 1 byte from ORIGIN to DEST. X is still zero. Now comes the part that makes indexing worthwhile. The line labeled ADJUST is the shortest line of source code we've seen yet, consisting only of the mnemonic INX, which means "increment the X register." Since the X register was zero, when this line is executed the X register will be left holding a value of one.

Compare Register

In Example 7, the line labeled TEST compares the value in the X register with the number "9." There are three compare instructions for the 6502, one for each register. CMP compares a value with the contents of the accumulator; CPX compares a value with the contents of the X register, and CPY compares a value with the contents of the Y register.

We can use these compare instructions to compare any register with any value in memory, or, in the immediate mode, to compare any register with any constant. Such comparisons enable us to test for given conditions. For example, in Example 7, the line labeled TEST tests to see if we've moved 9 bytes yet. If the X register holds the value "9," then we have moved 9 bytes. (Walk through the loop yourself. When you have moved the zeroth through the eighth bytes above ORIGIN to the zeroth through the eighth positions above DEST, then you have moved 9 bytes.)

A compare instruction never changes the contents of a register or of any location in memory. Thus, the X register does not change when the line labeled TEST is executed by the 6502. What may change, however, are some of the 6502's status flags.

Status Flags

In addition to the 6502's general-purpose registers (A, X, and Y), the 6502 contains a special register P, the processor status register. Individual bits in the processor status register are set or cleared each time the 6502 performs certain operations. These bits, or hardware flags, are:

C	bit 0: Carry Flag
Z	bit 1: Zero Flag
I	bit 2: Interrupt Flag
D	bit 3: Decimal Flag
B	bit 4: Break Flag
	bit 5: Undefined
V	bit 6: Overflow Flag
N	bit 7: Negative Flag

In this book, we will not discuss the use of all the flags in the processor status register. In this quick course in assembly-language programming, and in the software subsequently presented in this book, the three flags we will deal with are C, the carry flag; Z, the zero flag; and N, the negative flag.

A compare operation (CMP, CPX, or CPY) does not change the value of registers A, X, or Y, but it does affect the carry, zero, and negative flags.

For example, if a register is compared with an equal value, the zero flag, Z, will be set; otherwise, Z will be cleared. If an instruction sets bit 7 of a register or an address, the negative flag of the status register will also be set; conversely, if an instruction clears bit 7 of a register or an address, the negative flag will be cleared. Similarly, mathematical and logical operations set or clear the carry flag, which acts as a ninth bit in all arithmetic and logical operations. Table 3.1 summarizes the effects of a compare instruction on the status flags.

Table 3.1: *Status flags affected by compare instructions. Note that if you wish to test the status of the carry flag after a compare, you must set it (using the instruction SEC) before the compare. When testing the N flag, think of the inputs as signed 8-bit values.*

	Carry Flag*	Negative Flag	Zero Flag
Compare a register with an *equal* value and you	set C,	clear N, and	set Z.
Compare a register with a *greater* value and you	clear C,	clear N, and	clear Z.
Compare a register with a *lesser* value and you	set C,	clear N, and	clear Z.

Conditional Branching

We can have a program take one action or another, depending on the state of a given flag. For example, two instructions, BEQ, (*B*ranch on result *EQ*ual) and BNE (*B*ranch on result *N*ot *E*qual) cause the 6502 to *branch*, or jump to a new instruction, based on the state of the zero flag. An instruction which causes the 6502 to branch based on the state of a flag is called a conditional branch instruction. Other conditional branch instructions are based on the state of other status flags and are given in table 3.2.

---

*If you wish to test the status of the carry flag after a compare, you must set it (using the instruction SEC) before the compare.

Table 3.2: *Conditional branch instructions.*

Flag	Instruction	Description	Opcode
C	BCC	Branch if carry clear.	90
C	BCS	Branch if carry set.	B0
N	BPL	Branch if result positive.	10
N	BMI	Branch if result negative.	30
Z	BEQ	Branch if result equal. (Zero Flag set).	F0
Z	BNE	Branch if result not equal. (Zero flag clear.)	D0
V	BVC	Branch if overflow flag clear.	50
V	BVS	Branch if overflow flag set.	70

The line labeled TEST in Example 7 compares the X register to the value "9;" this sets or clears the zero flag. The line labeled BRANCH then takes advantage of the state of the zero flag, by branching back to the line labeled GET if the result of that comparison was not equal. But if Y did equal "9," then the result of the comparison would have been equal, and the 6502 would *not* branch back to GET. Instead, the 6502 would execute the instruction following the line labeled BRANCH.

Loops

Example 7 shows a program loop. We cause the 6502 to perform a certain operation many times, by initializing and then incrementing a counter, and testing the counter each time through the loop to see if the job is done.

There's a lot of power in loops. What would we have to add or change in Example 7 so that it moves not 9, but 90 bytes from one place to another? Happily, we wouldn't have to add anything, and we'd only have to change the operand in the line labeled TEST. Instead of comparing the X register with 9, we'd compare it with 90. See Example 8.

Example 8

Move 90 bytes from origin to destination.

```
ORIGIN = $0200
DEST   = $0300
```

```
INIT      LDX #0           Initialize X register to zero, so we'll start
                           with the first byte in the block.
GET       LDA ORIGIN,X     Get Xth byte in origin block.
PUT       STA DEST,X       Put it into the Xth position in the
                           destination block.
ADJUST    INX              Adjust X for next byte.
TEST      CPX #90          Done 90 bytes yet?
BRANCH    BNE GET          If not, get next byte...
```

Writing loops lets us write code that is not only compact, but easily tailored to meet the demands of a particular application. We couldn't do that, however, without indexing and branching.

Loops can be tricky, though. What's wrong with this loop?

Example 9

```
          ORIGIN = $0200
          DEST   = $0300

INIT      LDX #0           Initialize X register to zero, so we'll start
                           with the first byte in the block.
GET       LDA ORIGIN,X     Get Xth byte in origin block.
PUT       STA DEST,X       Put it into the Xth position in the
                           destination block.
TEST      CPX #9           Done 9 bytes yet?
BRANCH    BNE GET          If not, get next byte...
```

Examine Example 9 very carefully. How does it differ from Example 7? It lacks the line labeled ADJUST, which increments the X register. What will happen when the 6502 executes the code in Example 9? It will initialize X to zero; it will get a byte from ORIGIN and move it to DEST. Then it will compare the contents of register X to 9. Register X won't equal 9, so it will branch back to GET, where it will do *exactly what it did the first time through the loop*, because X will still equal zero. Until the X register equals 9, the 6502 will branch back to GET. But nothing in this loop will ever change the value of X! So the 6502 will sit in this loop forever, getting a byte from ORIGIN and putting it in DEST and determining that the X register does not hold a 9...

Now look at Example 10. Will it cause the 6502 to loop, and if so, will the 6502 ever exit from the loop? Why, or why not?

Example 10

```
        ORIGIN = $0200
        DEST   = $0300

INIT    LDX #0          Initialize X register to zero, so we'll start
                        with the first byte in the block.
GET     LDA ORIGIN,X    Get Xth byte in origin block.
PUT     STA DEST,X      Put it into the Xth position in the
                        destination block.
ADJUST  INX             Adjust X for next byte.
TEST    CPX #9          Done 9 bytes yet?
BRANCH  BNE INIT        If not, get next byte...
```

Relative Addressing

All conditional branch instructions use the relative addressing mode, and they are the only instructions to use this addressing mode. Like the zero page and zero-page indexed addressing mode, the relative addressing mode requires only a 1-byte operand. This operand specifies the relative location of the opcode to which the 6502 will branch if the status register satisfies the condition required by the branch instruction. A relative location of 04 means the 6502 should branch to an opcode 4 bytes beyond the next opcode, if the given condition is satisfied. Otherwise, the 6502 will proceed to the next opcode.

Because the operand in a conditional branch instruction is only 1 byte, it is not possible for a conditional branch instruction to cause a branch more than 127 bytes forward or 128 bytes backward from the current value of the program counter. (A branch backward is indicated if the relative address specified is negative; forward if it's positive. A byte is negative if bit 7 is set. A byte is positive if bit 7 is clear. Thus, a value of 00 is considered positive.) However, an instruction called JMP allows the programmer to specify an unconditional branch to any location in memory. Therefore, if we have a short conditional branch followed by an unconditional jump, we may achieve in two instructions a conditional branch to any location in memory.

Unconditional Branch

Just as BASIC has its GOTO command, which causes an unconditional branch to a specified line in a BASIC program, the 6502 has its JMP instruction, which unconditionally branches to a specified address. A program may loop forever by JMP'ing back to its starting point.

Look at Example 11. Unless a line of code within the loop causes the 6502 to branch to a location outside of the loop, the 6502 will sit in this loop forever.

Example 11

Endless Loop:

```
START   xxxxxxxxxx     some
        xxxxxxxxxx     instructions
        xxxxxxxxxx
        JMP START
```

Indirect Addressing

A JMP instruction may be written in either the absolute addressing mode or the indirect addressing mode. Absolute addressing is used in Example 11. The operand is the address to which the 6502 should jump. But in the indirect mode (which is always signified by parentheses in the operand field) the operand specifies the address of a *pointer*. The 6502 will jump to the address specified by the pointer; it will not jump to the pointer itself.

The line of code "JMP (POINTR)" will cause the 6502 to jump to the address specified by the 2 bytes at POINTR and POINTR+1. Thus, if POINTR = $0600, and the 6502 executes the instruction "JMP (POINTR)" when memory location $0600 holds $00 and $0601 holds $20, then the 6502 will jump to address $2000. (Remember, addresses are always stored in memory with the low byte first.)

How Branching Works

Incidentally, all branches, whether relative, absolute, or indirect, work by operating on the contents of the PC (program counter). Before any branch instruction is executed, the PC holds the address of the current opcode. A branch instruction changes the PC, so that in the next instruction cycle the 6502 will fetch not the opcode following the current opcode, but the opcode at the location specified by the branch instruction. Then execution will continue normally from the new address.

Relocatability

Often I implement short unconditional branches as:

```
CLC
BCC    PLACE
```

rather than as:

```
JMP    PLACE
```

This is because the first method (relying as it does on relative rather than absolute addressing) will still work even if you relocate the code in which it is contained. Making your code relocatable will save you time and trouble when you try to move your programs around in memory and still want them to work.

To relocate code containing the second example, you'd have to change the operand field because the absolute address of PLACE will have changed. To relocate code containing the first example, you wouldn't have to change a thing.

Subroutines

Perhaps the two most powerful instructions available to the assembly-language programmer are the JSR (*J*ump to *S*ub*R*outine) and the RTS (*R*e*T*urn from *S*ubroutine). These instructions (equivalent to GOSUB and RETURN in BASIC) enable us to organize chunks of code as building blocks called subroutines.

Think of the subroutine as a job. Your computer can do more work for you if it knows how to do more jobs. Once you teach the 6502 how to do a given job, you won't have to tell it twice. Let's say you're writing a program in which the same operation must be performed at various times within a program. In every location within your program where the operation is required, you could include code to perform that operation. On the other hand, you could write code in one place to perform that operation, but write that code as a subroutine, and then *call* that subroutine whenever necessary from the main, or calling program. A call to a subroutine causes that routine to execute. When finished, it returns to the instruction following the call in the main program.

It only takes one line of code to call a subroutine. JSR SUB will call the subroutine located at the address labeled SUB. After the 6502 fetches and executes the JSR opcode, the next opcode it fetches will be at the address labeled SUB, in this example. So far it looks like an unconditional JMP. The 6502 will fetch and execute opcodes from the addresses following SUB, until it encounters an RTS instruction.

When the 6502 fetches an RTS instruction, it returns to its caller, jumping to the first opcode following the JSR instruction that called the subroutine. In effect, when a line of code calls a subroutine, the 6502 remembers where it is before it jumps to the new location. Then when it encounters an RTS instruction, it knows the address to which it should return because it remembers where it came from. It then continues to fetch opcodes from the point following the JSR instruction. Figure 3.1 illustrates this procedure. Note that the same subroutine may be called from many different points in the same program, and will always return to the opcode following the JSR instruction that called it.

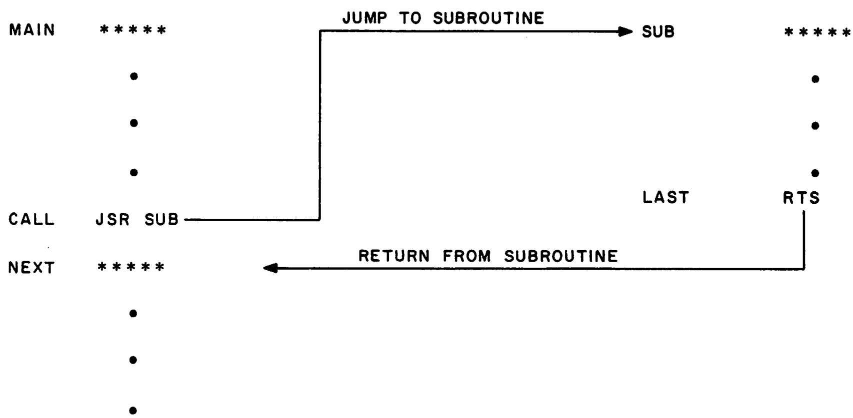

Figure 3.1: *Jump to and return from subroutine. When the processor encounters a JSR (jump to subroutine) instruction, the next instruction executed is the first instruction of the subroutine. Here, the subroutine SUB is called from MAIN. The last instruction executed in a subroutine must be an RTS (return from subroutine) instruction. Here, the instruction at label LAST in subroutine SUB returns control to the next instruction following the call to the subroutine in the main program, the instruction labeled NEXT. The subroutine SUB can be called anywhere in the program MAIN when the particular function of SUB is needed.*

Subroutines allow you to structure your software. With structured software, you can make changes to many programs just by changing one subroutine. If, for example, all programs that print characters do so by calling a single-character-print subroutine, then any time you improve that subroutine you improve the printing behavior of all your programs. Changing something only once is a tremendous advantage over having to change something in many different (usually undocumented) places within a piece of code. For these reasons, all of the software in this book uses subroutines.

Dummies

A *dummy subroutine* is a subroutine consisting of nothing but an RTS instruction. A line of code in a program can call a dummy subroutine and nothing will happen; the 6502 will return immediately, with its registers unchanged.

So why call a dummy subroutine?

A call to a dummy subroutine provides a "hook," which you may use later to call a functional subroutine. While developing a program, I may have many lines of code that call dummy subroutines. Later, when I write the lower-level subroutines, it's easy to change my program so that it calls the functional subroutines rather than the dummy subroutines. Trying to insert a subroutine call to a program lacking such a hook can make you wish for a "memory shoehorn," which might let you squeeze 3 extra bytes of code into the same address space.

The Stack

In addition to the addressing modes that enable the 6502 to access addressable memory, one addressing mode lets the 6502 access a 256-byte portion of memory called the *stack*.

You may think of this stack as a stack of trays in a cafeteria. The only way a tray can be added is to place it on top of the existing stack. Similarly, the only way to get a tray from the stack is to remove one from the top. This is the LIFO (Last-In, First-Out) method. The last tray placed onto the stack must be the first tray removed.

In our case, when an item is placed onto the top of the stack, it is called a *push*, and when an item is removed from the top of the stack, it is called a *pop*. The last item onto the stack is said to be at the *top* of the stack.

For example, let's say we want to place two items onto the stack. (Each item has an 8-bit value, perhaps a number or an ASCII character; see figure 3.2a.) First we push item 1 onto the stack, as illustrated in figure 3.2b. All positions above item 1 on the stack are said to be *empty*, the item 1 is on the top of the stack.

Now, push item 2 onto the stack (see figure 3.2c). What happens? Item 2 is now at the top of the stack, not item 1, although item 1 is still on the stack.

Next, to get item 2 back off the stack, we do a pop (see figure 3.2d). This makes item 1 the top of the stack again. Finally, another pop will remove item 1 from the stack, leaving the stack completely empty. Note that we had to pop item 2 from the stack before we could get to item 1 again. This is the LIFO principle.

The instruction PHA lets you push the contents of the accumulator onto the stack. PLA lets you load the accumulator from the top of the stack (a pop). PHP lets you push the processor status register onto the stack. PLP lets you load the processor status register from the stack.

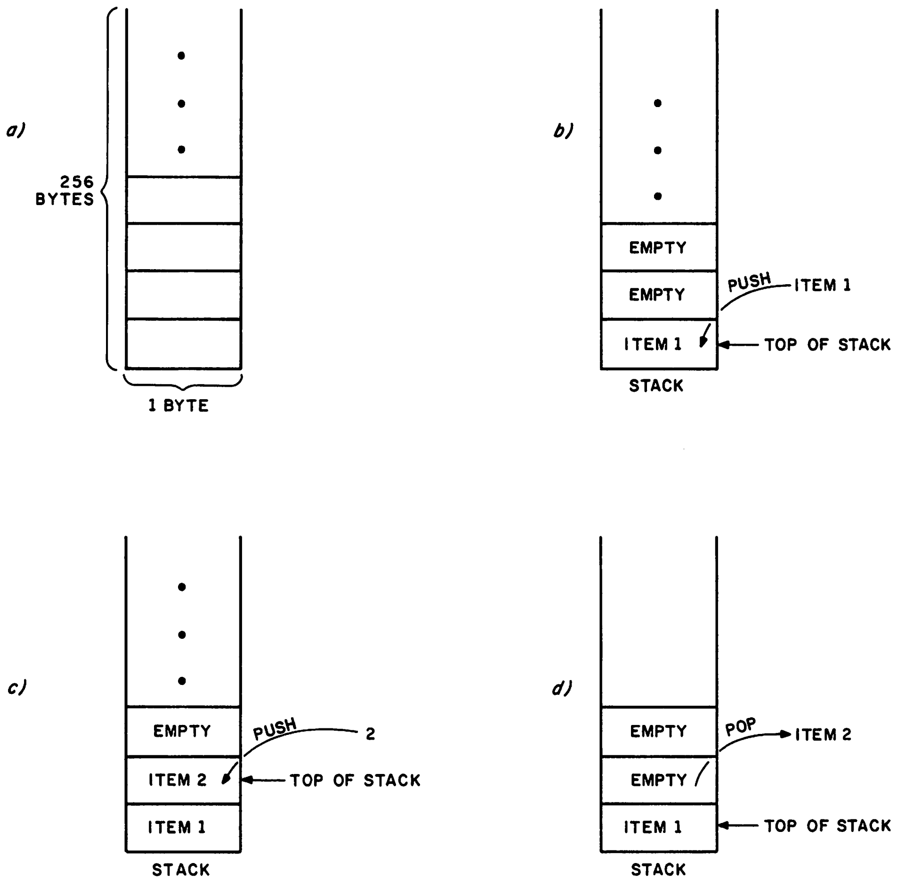

Figure 3.2: *Pushing and popping the stack.*

The stack is a very convenient "pocket" to use when you want to store one or a few bytes temporarily without using an absolute place in memory. Subroutines may pass information to the calling routines by using the stack, but be careful: if a subroutine pushes data onto the stack, and fails to pop that data from the stack before executing an RTS instruction, then that subroutine will *not* return to its caller. This happens because when the 6502 executes a JSR instruction, it pushes the return address—that is, the address of the opcode following the JSR instruction—onto the stack. A subroutine can return to its caller only because its return address is on the stack. If its return address is not at the top of the stack when the subroutine executes an RTS, it will not return to its caller. So a subroutine should always restore the stack before trying to return.

Chapter 4:

Arithmetic and Logic

Character Translation

As demonstrated by Examples 7 and 8, indexed addressing is handy for performing a given operation (such as a move) on a contiguous group of bytes. But it also has another important application: table lookup. For example, let's say you and a friend have decided to write notes to one another using a substitution code. For every letter, number, and punctuation mark in a message, you've agreed to substitute a different character. A "W" will be replaced with a "Y;" a semicolon may be replaced with a "9," etc.

You each have the same table showing you what to substitute for each character that may appear in a message. So you write a note to your friend in English, and then, using this table (which might be in the form of a Secret Agent Decoding Ring) you code, or encrypt, your note. You send the note in its encrypted form to your friend. Anyone else looking at the note would just see garbage, but your friend knows that a message can be found in it. So he gets his copy of the character translation table (which may be in *his* Secret Agent Decoding Ring), and he translates the encrypted message back into English, looking up the characters that correspond to each character in the coded message.

Children often enjoy coding and decoding messages in this way, but I find it about as much fun as filling out forms — which is no fun at all. Unfortunately, programming often involves character translation. Fortunately, I don't have to do it myself. I let my computer perform any necessary character translation by having it do what our two secret agents were doing: look up answers in a table.

Example 12
Character Translation Subroutine

XLATE	TAX	Use character to be translated as an index into the table.
	LDA TABLE,X	Look up value in table.
	RTS	Return to caller, bearing translated character in A and original character in X.

Transfer Register

In Example 12, the subroutine XLATE assumes when it is called that the accumulator holds the byte to be translated. This byte might be a letter, a number, a punctuation mark, a control code, or a graphic character, but however you think of it, it's an 8-bit value. Line 1 of XLATE transfers that 8-bit value from the accumulator to the X register, using the register-transfer instruction TAX.

Register-transfer instructions operate only on registers; they do not affect addressable memory. These instructions allow the contents of one register to be copied, or transferred, to another. The results of a transfer leave the source register unchanged, and the destination register holding the same value as the source register. The 6502's register-transfer instructions are:

TAX	Transfer accumulator to X register.
TAY	Transfer accumulator to Y register.
TXA	Transfer X register to accumulator.
TYA	Transfer Y register to accumulator.

Register transfers affect flags N and Z.

These instructions let you transfer A to X or Y, or to transfer X or Y to A. But how would you transfer X to Y, or Y to X? (Hint: it will take two lines of source code, each line an instruction from the list above.)

Table Lookup

In Example 12, line 2 of XLATE actually performs the character translation by looking up the desired data in a table. The label, TABLE, identifies the base address for a table that we've previously entered into memory. The indexed addressing mode allows line 2 to get the Xth byte above the base address (ie: to get the Xth byte of the table). When that line is executed, the table lookup is complete. The 6502 has looked up and now holds in the accumulator the Xth byte in the table. Now all the 6502 must do is return to its caller, bearing the translated character in A and the original character in X. It accomplishes this with the RTS instruction.

Now you can perform this character translation at any point in any program with just one line of source code:

```
JSR XLATE
```

Table lookup gives me great flexibility as a programmer. If a program uses a table lookup and for some reason I want the program to behave differently, I will probably only have to change some values in the table; it's unlikely that I'll have to change the table lookup code itself. If I've set up my table well, I might not have to change anything in the program except the data in the table.

Table lookup is therefore a very fast and flexible means of performing data translation. But the cost of that speed and flexibility can be size. You might be able to solve any problem with the right tables in memory, but not if you can't afford the memory necessary to hold all those tables. It's great when a program can just look up the answers it needs, but sometimes a program will actually have to *compute* its answers.

Arithmetic Operations

The 6502 can perform the following 8-bit arithmetical operations:

Shift
Rotate
Increment
Decrement
Add
Subtract

To understand how the 6502 operates on a byte, you must *think of the bits* in that byte. Even if the byte represents a number or a letter, don't think about what you can do to that number or letter. Think about what you can do to the pattern of bits in that byte.

What *can* you do to those bits?

Shift

You can shift the bits in a byte one position to the left or to the right. An ASL (Arithmetic Shift Left) operates on a byte in this manner: it moves each bit one bit to the left; it moves the leftmost bit (bit 7) into the carry flag, and it sets the rightmost bit (bit 0) to zero. See figure 4.1.

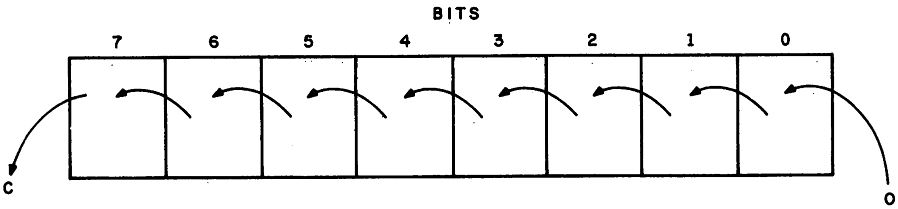

Figure 4.1: *Effect of the ASL instruction.*

For example, if the byte at location TMP has the following bit pattern:

address TMP   0   1   0   1   0   1   1   0

then after the instruction "ASL TMP" is executed, the data would look like:

address TMP   1   0   1   0   1   1   0   0

with the carry flag being set to the previous value of bit 7, in this case 0. If the same instruction is again executed, the data becomes:

address TMP   0   1   0   1   1   0   0   0

and the carry flag is set to 1.

A LSR (Logical Shift Right) has just the opposite effect of the ASL. All bits are shifted to the right towards the carry flag, introducing zeroes through bit 7. See figure 4.2.

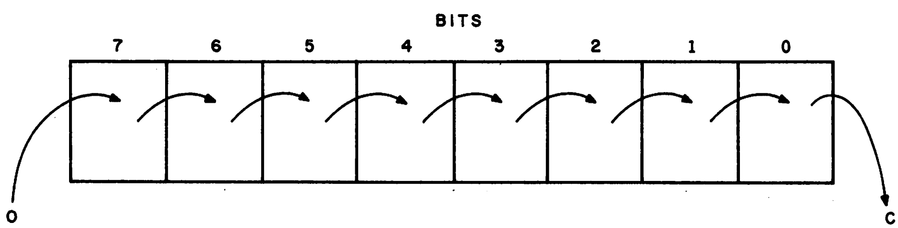

Figure 4.2: *Effect of the LSR instruction.*

For example, if the byte at location TMP is as originally given above, then after the instruction "LSR TMP" is executed, the data at TMP becomes:

address TMP   0 0 1 0 1 0 1 1

with the carry flag being set to the previous value of bit 0, in this case zero. If the same instruction is executed again, the data becomes:

address TMP   0 0 0 1 0 1 0 1

with the carry flag set to 1.

Because a number is represented in binary (each bit represents a successive power of two), some arithmetic operations are simple. To divide a byte by two, simply shift it right; to multiply a value in a byte by two, simply shift it left.

Rotate

You can also rotate the bits in a byte to the left or to the right *through* the carry flag. Unlike shifting, rotating a byte preserves all the information originally contained by a byte.

Figure 4.3 shows how a ROL (rotate left) instruction works. For instance, let's say the data at address TMP is originally the same as in previous examples:

address TMP   0 1 0 1 0 1 1 0

and let's say that the carry flag is set (ie: it holds a 1).

After a "ROL TMP" instruction is executed, the data becomes:

address TMP   1 0 1 0 1 1 0 1

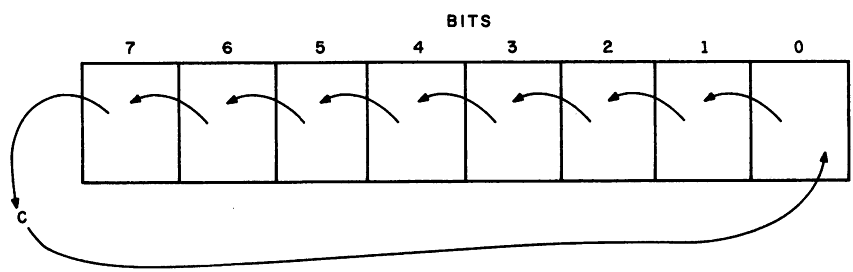

Figure 4.3: *Effect of the ROL instruction.*

and the carry bit is set to the previous value of bit 7, namely 1. Notice that bit 0 in TMP now holds the original contents of the carry flag, and the carry flag holds the original contents of bit 7. Otherwise, everything looks just the same as in the ASL operation. After a second execution of the instruction "ROL TMP," the data becomes:

address TMP   0   1   0   1   1   0   1   1

with the carry flag set to 1.

In a rotate left instruction, bit 0 is always set from the carry flag. (In the ASL instruction, bit 0 is always set to 0.) If this had been an ASL instruction, what would the bit pattern at TMP be?

Figure 4.4 shows how a ROR (rotate right) instruction works. It is similar to ROL, except that the carry flag is set *from* bit 0, and bit 7 is set from the carry flag.

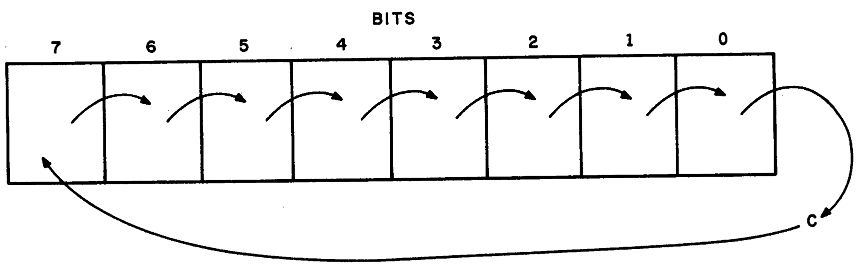

Figure 4.4: *Effect of the ROR instruction.*

Rotate a byte left nine times and you'll still have the original byte. The same is true if you rotate a byte right nine times. But *shift* a byte left nine times, or right nine times, and you know what you've got left? Nothing!

Increment, Decrement

You can increment or decrement a byte in three ways: using the INC and DEC instructions to operate on a byte in memory, using INX and DEX to operate on the X register, or using INY and DEY to operate on the Y register. None of these instructions affects the carry flag. They do affect the zero flag: Z is set if the result of an increment or decrement is zero; otherwise Z is cleared. The negative flag is set if the result of an increment or decrement is a byte with bit 7 set; otherwise N is cleared.

Note that if you increment a register or address holding $FF, it will hold zero. And similarly, if you decrement a register or address holding a zero, it will hold $FF.

You cannot increment or decrement the accumulator, but you can add or subtract a byte from the accumulator.

Addition

Example 13 shows how to add a byte from the location labeled NUMBER to the accumulator:

Example 13

CLC	Clear the carry flag.
ADC NUMBER	Add the contents of location NUMBER to the accumulator.

After these instructions are executed, the accumulator will hold the low 8 bits of the result of the addition. If, following the addition, the carry flag is set, then the result of the addition was greater than 255; if the carry flag is clear, then the result was less than 256, and, therefore, the accumulator is holding the full value of the result. Remember, the carry flag must be cleared before performing the ADC instruction.

Subtraction

Subtraction is as easy as addition. To subtract a byte from the accumulator, first set the carry flag (using the SEC instruction) and then subtract from the accumulator a constant or the contents of some address, using the instruction SBC (subtract with carry):

SEC	Set the carry flag.
SBC OPERND	Subtract from accumulator the value of OPERND.

If the operand is greater than the initial value of the accumulator, the subtract operation will clear the carry flag; otherwise the carry flag will remain set. In either case, the accumulator will bear the 8-bit result.

Thus, you clear the carry flag before adding and set the carry flag before subtracting. If the carry flag doesn't change state, then the accumulator bears the entire result. But if the addition or subtraction changes the state of the carry flag, then your result is greater then 255 (for an addition) or less than zero (for a subtraction).

Decimal Mode

The processor status register includes a bit called the *decimal flag*. If the decimal flag is set, then the 6502 will perform addition and subtraction in decimal mode. If the decimal flag is clear, then the 6502 will perform addition and subtraction in binary mode. Decimal mode means the bytes are treated as BCD (Binary Coded Decimal), meaning that the low 4 bits of a byte represent a value of 0 thru 9, and the high 4 bits of the byte represent a value of 0 thru 9. Neither *nybble* (4 bits) may contain a value of A-F. So, each nybble represents a decimal digit.

The instructions SED and CLD set the decimal flag and clear it, respectively. Unless you'll be operating with figures that represent dollars and cents, you won't need to use the decimal mode. All software in this book assumes that the decimal mode is not used.

Decimal 255 is the biggest value that can be represented by a binary-coded byte, but decimal 99 is the biggest value that can be represented by a byte using Binary Coded Decimal.

Logical Operations

What if you want to set, clear, or change the state of one or more bits in a byte without affecting the other bits in that byte? Input and output operations often demand such "bit-twiddling," which can be performed by the 6502's logical operations ORA, AND, and XOR.

Setting Bits

The ORA instruction lets you set one or more bits in the accumulator without affecting the state of the other bits. ORA logically OR's the accumulator with a specified byte, or *mask*, setting bit n in the accumulator if bit n in the accumulator is initially set *or* if bit n in the mask is set, or if both of these bits are set. A logical OR will leave bit n of the accumulator clear only if bit n is initially clear in both the accumulator and the mask. Table 4.1 shows a *truth table* for the logical operator OR. A truth table gives all possible combinations of 2 bits that can be operated upon (in this case, ORed) and the results of these combinations.

Table 4.1: *Truth table for the logical OR operand.*

Bit 1		Bit 2		Result
0	OR	0	=	0
0	OR	1	=	1
1	OR	0	=	1
1	OR	1	=	1

For example, suppose we executed the instruction "ORA #$80." Here the mask is $80, or the bit pattern 10000000. This instruction would therefore set bit 7 of the accumulator while leaving all other bits unchanged. So, if the accumulator had a value of 00010010 before the above instruction was executed, it would have the value of 10010010 afterwards.

Another example would be "ORA #3." Since a decimal 3 becomes 00000011 when converted to an 8-bit binary mask, the above instruction would set bits 0 and 1 in the accumulator, leaving bits 2 thru 7 unchanged.

How would you set the high 4 bits in the accumulator? The low 4 bits?

Clearing Bits

You can clear one or more bits in the accumulator without affecting the state of the other bits through the use of the AND instruction. AND performs a logical AND on the accumulator and the mask specified by the operand. AND will set bit n of the accumulator only if bit n of the accumulator is set initially *and* bit n is set in the mask. If bit n is initially clear in the accumulator or if bit n is clear in the mask, then AND will clear bit n in the accumulator. Table 4.2 gives the truth table for the logical AND operation.

Table 4.2: *The truth table for the logical AND.*

Bit 1		Bit 2		Result
0	AND	0	=	0
0	AND	1	=	0
1	AND	0	=	0
1	AND	1	=	1

For instance, the line of source code "AND #1" will clear all bits except bit 0 in the accumulator; bit 0 will remain unchanged. "AND #$F0" will clear the low 4 bits of the accumulator, leaving the high 4 bits unchanged. Select the right mask, and you can clear any bit or combination of bits in the accumulator without affecting the other bits in the accumulator.

Toggle Bits

The exclusive OR operation, XOR, lets you "flip," or toggle, one or more bits in the accumulator (ie: change the state of one or more bits without affecting the state of other bits). XOR will set bit n of the accumulator if bit n is set in the accumulator but not in the mask, or if bit n is set in the mask but not in the accumulator. If bit n has the same state in both the accumulator and in the mask, then XOR will clear bit n in the accumulator. Table 4.3 shows the truth table for this operation.

Table 4.3: *The truth table for the exclusive OR (XOR).*

Bit 1		Bit 2		Result
0	XOR	0	=	0
0	XOR	1	=	1
1	XOR	0	=	1
1	XOR	1	=	0

To toggle bit n in the accumulator, simply XOR the accumulator with a mask which has bit n set but all other bits clear. Bit n will change state in the accumulator, but all other bits in the accumulator will remain unchanged.

The logical operators, combined with the 6502's relative branch instructions, make it possible for a program to take one action or another depending on the state of a given bit in memory. Let's say you want a piece of code that will take one action (Action A) if a byte, called FLAG, has bit 6 set; yet take another action (Action B) if that bit is clear. The code of Example 14 shows one way to ignore all other bits in FLAG, and still preserve FLAG.

Example 14

```
LDA FLAG          Get flag byte.
AND #$40          Clear all bits but bit 6.
BEQ PLAN.B
```

```
PLAN.A      xxxxx       Take Action A, since bit 6 was set
                        in flag.
             .
             .
             .
PLAN.B                  Take Action B, since bit 6 was
                        clear in flag.
```

What good are flags? Let me give an example. The flag on a rural mailbox may be either raised or lowered to indicate that mail is or is not awaiting pickup. Raising and lowering those flags requires a little bit of effort (no pun intended), but it enables the mail carrier to complete the route much more quickly than would be possible if every mailbox had to be checked every time around. Presumably, this provides better service for everyone on the route.

That mail carrier's routine is a very sophisticated piece of programming. If we think of the mail carrier as a person following a program, then we can see some of the power and flexibility that come from the use of flags.

The mail carrier's program has two parts: *What must be done at the post office* and *What must be done on the route*. At the post office, the mail carrier sorts the mail, bundles letters for the same address and puts the bundles for a given route into a mail sack in some order. This sorting at the post office means the mail carrier on the route can make his or her rounds more quickly, because no further sorting and searching is required. (We won't go into sorting and searching in this book; that's a volume in itself. For a helpful reference see Donald E Knuth's *Searching and Sorting*.)

Now comes the second part of the mail carrier's program: *What must be done on the route*. The mail carrier picks up the mail sack and leaves the post office. Driving down country roads, the mail carrier sees a mailbox ahead. *Do I have any mail for the people at this address? If so,* the mail carrier's mental program says, *I'll slow down and deliver it. But what if I don't have any mail now for these people? Do I just keep driving? Do I go to the next address?*

Not if I want to keep my job.

The mail carrier looks a little more closely at the mailbox. *Is the flag up or down? If it's down, I can just drive by, but if the flag is up I must stop and pick up the outgoing mail.*

A flag is just a single bit of information, but by interpreting and responding to the state of flags, even a simple program can respond to many changing conditions. If your computer has 8,000 bytes of programmable memory, that means it has 64,000 bits of memory. Conceivably, you could use most of those bits as flags, perhaps simulating the patterns of outgoing mail in a community of more than 50,000 households.

But you didn't buy a computer to play post office. And you know enough now to follow the programs presented in the following chapters. These programs will include examples of all the instructions and programming techniques presented in this very fast course in assembly-language programming. The programs in the following chapters will also give you some tools to use in developing your own programs.

(Incidentally, there is one 6502 instruction which doesn't do anything at all. The instruction NOP performs NO operation. Why would you want to perform no operation? Occasionally, it's handy to replace an unwanted instruction with a dummy instruction. When you want to disable some code, simply replace the unwanted code with NOP's. A NOP is represented in memory by $EA.)

Chapter 5:

Screen Utilities

Now let's consider how to display something on the video screen. On the Commodore 64 and VIC-20 computers, the video-display circuitry scans a particular bank of memory, called the display memory. Every address in the display memory represents, or is mapped to, a different screen location (hence the term *memory-mapped display*). For each character in the display memory, the display circuitry puts a particular image, or graphic, on the screen (hence the term *character graphics*). To display a character in a given screen location, you need only store that character in the one address within a display memory that corresponds to the desired screen location.

To know which address corresponds to a given screen location you must consult a display-memory map. Appendices B1 and B2 describe how display memory is mapped on the Commodore 64 and VIC-20 computers. Note that two different systems may have two different addresses for the same screen location. Also note how burdensome it can be to look up the addresses of even a few screen locations just to display a few characters on the video screen.

Rather than address the screen in an absolute manner, we'd like to be able to do so indirectly. Ideally, we'd like a software-controlled "hand" that we can move about the screen. Then we could pick up the character under the hand, or place a new character under the hand, without being concerned with the absolute address of the screen location under the hand at the moment. Such a hand can be implemented quite easily as a zero-page pointer.

Pointers

A pointer is just a pair of contiguous bytes in memory. Since 1 byte contains 8 bits, a pointer contains 16 bits, which means a pointer can specify any one of more than 65,000 (specifically: 2^{16}) different addresses.

A pointer can specify, or point to, only one address at a time. The low byte of a pointer contains the 8 LSB (least-significant bits) of the address it specifies, and the high byte of the pointer contains the 8 MSB (most-significant bits) of the address it specifies.

Let's say we want a pointer at location $1000. We must allocate 2 bytes for the pointer, which means it will occupy the bytes at $1000 and $1001. $1000 will hold the low byte, and $1001 will hold the high byte. If we want this pointer to specify address $ABCD, then we may set it as follows:

POINTR = $1000				This assembler directive equates the label POINTR with the value $1000. (It's POINTR and not POINTER only because the assembler used in preparing this book chokes on labels longer than six characters — a common, if arbitrary, limitation.)
LDA #$CD	A9	CD		Set the
STA POINTR	8D	00	10	low byte.
LDA #$AB	A9	AB		Set the
STA POINTR+1	8D	01	10	high byte.

Now POINTR points to $ABCD.

Although a pointer may be anywhere in memory, it becomes especially powerful when it's in the zero page (the address space from 0000 to $00FF). The 6502's indirect addressing modes allow a zero-page pointer to specify the address on which certain operations may be performed. A zero-page pointer must be located in the zero page, but it may point to any location in memory. For example, a zero-page pointer may be used to specify the address in which data will be loaded or stored. Since display memory looks like any other random-access memory to the processor, we may implement our television hand as a zero-page pointer.

TV.PTR

We want a zero-page pointer that can point to particular screen locations. Let's call it TV.POINTER, or TV.PTR for short. Whenever we examine or modify the screen, we'll do it through the TV.PTR.

Because the Commodore 64 and VIC-20 don't use zero page bytes $00FB–$00FE, we'll use $00FB and $00FC for TV.PTR. We can do that with the following assembler directive:

```
TV.PTR = $FB
```

TV.PUT

The TV.PTR always specifies the current location on the screen. Thus, to display a graphic at the current location on the screen, we need only load the accumulator with the 8-bit code for that graphic and then execute the following two lines of code:

```
LDY #0            A0  00
STA (TV.PTR),Y    91  FB
```

The two lines of above code are sufficient to display a given graphic in the current screen location. But what if you want to display a given *character* in the current screen location? The ASCII code for a character is not necessarily the same as your system's display code for that character's *graphic*. To display an "A" in the current screen location, we cannot simply load the accumulator with an ASCII "A" (which is $41) and then execute the two lines of above code, because the graphic "A" has a different display code on your system. Instead of displaying an "A," we would display something else. Perhaps to make life difficult for assembly-language programmers, the Commodore computers do not provide a one-to-one correspondence between any character's ASCII code and that character's graphic code.

How then can we display a given ASCII character in the current screen location? We can do it by assuming that there exists a subroutine called FIXCHR, which will "fix" any given ASCII code, by translating it to its corresponding graphic or display code. FIXCHR will be different for each system, so we won't go into its details here (see the appendix pertaining to your computer for a description and listing of FIXCHR for your system). At this point we will assume only that FIXCHR exists, and that if we call it with an ASCII character in the accumulator, it will return with the corresponding display code in the accumulator.

We already know how to display a given graphic in the current screen location. With FIXCHR we now know how to display any given ASCII *character* in the current screen location. And since displaying any given ASCII character in the current screen location is something we're likely to do more than once, let's make it a subroutine. We'll call that subroutine TV.PUT since it will let us *put* a given ASCII character up on the TV screen:

```
TV.PUT    JSR FIXCHR          Convert ASCII character to your
                              system's display code for that character.
          LDY #0              Put that graphic in the
          STA (TV.PTR), Y     current screen location.
          RTS                 Return to caller.
```

The Screen Location

However, these examples of modifying and examining screen locations through the TV.PTR will work only if the TV.PTR is actually pointing at a screen location. Therefore, before executing code such as the examples given above, we must be sure the TV.PTR points to a screen location.

There are several ways to do this. If you want to write code that will run on only one machine (or on several machines whose display memory is mapped the same way), then you can use the immediate mode to set the TV.PTR to a given address on the screen. Let's say you want to set the TV.PTR to point to the third column of the fourth row (counting right and down from an origin in the upper-left corner). If you have VIC-20 with 8 K of expansion RAM, then you can consult your system's documentation and determine that address $1044 in display memory corresponds to your desired screen location. $10 is the high byte of this screen location; $44 is the low byte of this screen location. Thus, you can set TV.PTR with the following lines of code:

```
LDA #$44        A9  44  Set
STA TV.PTR      85  FB  low byte.
LDA #$10        A9  10  Set
STA TV.PTR+1    85  FC  high byte.
```

This code is fast and relocatable. But it's not very convenient to have to look up a display address every time we write code that displays something on the screen. It would be much more convenient if we could address the screen as a series of X and Y coordinates. Why not have a subroutine that sets the TV.PTR for us, provided we supply it with the desired X and Y coordinates?

TVTOXY

TVTOXY is a subroutine that sets the value of the TV.PTR to the display address whose X and Y coordinates are given by the X and Y registers. (Note that we count the columns and rows from zero.) To make the TV.PTR point to the third column from the left in the fifth row from the top, a calling program need only include the following code:

LDY #2	The leftmost column is column zero, so the third column is column two.
LDY #4	The topmost row is row zero, so the fifth row is row four.
JSR TVTOXY	Set TV.PTR to screen location whose X and Y coordinates are given by the X and Y registers.

.
.
.

How will TVTOXY work? We could have TVTOXY do just what we were doing: look up the desired address in a table. A computer can look up data in a table very quickly, but the speed may not be worth it if the table requires a lot of memory. If we don't mind waiting a little longer for TVTOXY to do its job, we can have TVTOXY *calculate* the desired value of TV.PTR, rather than look it up in a table. But how can you calculate the address of a given X and Y location on the screen?

You can't do it without data. But you don't need a large amount of data to determine the address of a given X,Y location in screen memory; you need only have access to the following facts:

HOME	The address of the character in the upper-left corner of the screen (ie: the lowest address in screen memory).
ROWINC	ROW INCrement: the address difference from one row to the next.

Knowing the values of HOME and ROWINC for a given system, you can calculate the address corresponding to any X,Y location:

HOME	Address of character in upper-left corner
+ X Register	+ X coordinate
+ (Y Register) × ROWINC	+ (Y coordinate) × ROWINC
TV.PTR	Address of screen location at column X, row Y.

Run through this calculation for several screen locations and compare the results with the addresses you look up in the display-memory map for your system. (Remember that we count columns and rows from zero, not from one.) Now if TVTOXY can run through this calculation for us, we'll never have to look at a display-memory map again; we can write all our display code in terms of cartesian coordinates.

But we shouldn't be satisfied with TVTOXY if it only runs through the above calculation. After all, what happens if TVTOXY is called and the Y register holds a very large number? If the Y register is greater than the number of rows on the screen, then the above calculation will set the TV.PTR to an address outside of display memory. We don't want that. Maybe a calling program will have a bug and call TVTOXY with an illegal value in X or in Y. If TVTOXY doesn't catch the error, the calling program may end up storing characters in memory that is not display memory. It might end up over-writing part of itself, which would almost certainly invite long and arduous debugging.

I hate debugging. I know I'm going to make mistakes, but I'd like my software to catch at least some bugs before they run amuck. So let's have TVTOXY check the legality of X and Y before blindly calculating the value of TV.PTR.

How can TVTOXY check the legality of X and Y? How big can X or Y get before it's too big? We need some more data:

TVCOLS	The number of columns on the display screen, counting from zero.
TVROWS	The number of rows on the display screen, counting from zero.

Now TVTOXY requires the following four facts about the host computer:

HOME
ROWINC
TVROWS
TVCOLS

If we store these facts about the host system in a particular block of memory, then TVTOXY need only consult that block of memory to learn all it needs to know about the screen. TVTOXY can then work as follows:

TVTOXY

```
TVTOXY    SEC             Is X out of range?
          CPX TVCOLS
          BCC X.OK        If not, leave it alone.
                          If X is out of range, give
          LDX TVCOLS      it its maximum legal value.
                          Now X is legal.

X.OK      SEC             Is Y out of range?
          CPY TVROWS
          BCC Y.OK        If not, leave it alone.
                          If Y is out of range, give
          LDY TVROWS      it its maximum legal value.
                          Now Y is legal.

Y.OK      LDA HOME        Set TV.PTR = HOME.
          STA TV.PTR
          LDA HOME+1
          STA TV.PTR+1

          TXA             Add X to TV.PTR.
          CLC
          ADC TV.PTR
          BCC COLSET
          INC TV.PTR+1
          CLC

COLSET    CPY #0          Add Y*ROWINC to TV.PTR.
          BEQ EXIT
LOOP      CLC
          ADC ROWINC
          BCC NEXT
```

```
                INC TV.PTR+1
NEXT            DEY
                BNE LOOP
EXIT            STA TV.PTR
                RTS                 Return to caller.
```

TVDOWN, TVSKIP, TVPLUS

Using TVTOXY, we can set TV.PTR to a screen location with any desired X,Y coordinates. But it would also be convenient to be able to modify TV.PTR relative to its current value. For example, after placing a character on the screen, we might want to make TV.PTR point to the next screen location to the right, or perhaps to the screen location directly below the current screen location. We might even want to make TV.PTR skip over several screen locations to make it point to "the nth screen location from here," where "here" is the current screen location. For these occasions, the subroutines TVDOWN, TVSKIP, and TVPLUS come in handy.

TVDOWN, TVSKIP, TVPLUS

```
TVDOWN          LDA ROWINC          Move TV.PTR down by one row.
                CLC
                BCC TVPLUS          Unconditionally branch.

TVSKIP          LDA #1              Skip one screen location by increment-
                                    ing TV.PTR.

TVPLUS          CLC                 Add the contents of the accumulator
                ADC TV.PTR          to the two zero-page bytes
                BCC NEXT            comprising the TV.PTR.
                INC TV.PTR+1

NEXT            STA TV.PTR

                RTS                 Return to caller.
```

Note that the routines TVDOWN and TVSKIP make use of the routine TVPLUS, which assumes that the accumulator has been set to the number of locations to be skipped. For TVDOWN and TVSKIP, the accumulator is set to ROWINC and 1, respectively.

Right now TVPLUS might not seem long enough to be worth making into a subroutine. Any program that calls TVPLUS could perform the addition itself, at a cost of only a few bytes, and at a saving of several machine cycles in the process. However, we may make TVPLUS more sophisticated later on.

For example, we could enhance TVPLUS so it performs error checking automatically, to ensure that TV.PTR will never point to an address outside of screen memory. Such error checking would be very burdensome for every calling program to perform, but if and when we insert it into TVPLUS, every caller will automatically get the benefit of that modification.

VUCHAR

With TV.PUT we can display an ASCII character in the current screen location, and with TVSKIP we can advance to the next screen location. So why not combine the two, creating a subroutine that displays in the current screen location the graphic for a given ASCII character, and then automatically advances TV.PTR so it points to the next screen location? This would make it easy for a calling program to display a string of characters in successive screen positions. Since this subroutine will let the user *view* a *char*acter, let's call it VUCHAR:

```
VUCHAR   JSR TV.PUT      Display, in the current screen location,
                         the graphic for the character whose
                         ASCII code is in the accumulator.
         JSR TVSKIP      Advance to the next screen location.
         RTS
```

We could even squeeze VUCHAR into the code presented above for TVDOWN, TVSKIP, and TVPLUS, by inserting one new line of source code immediately above TVSKIP. (See Appendix C1, the assembler listing for the Screen Utilities, which also includes some error checking within TVPLUS.)

VUBYTE

With the screen utilities presented thus far, we can display a character on the screen in the current location, but we don't have a utility to display a *byte* in hexadecimal representation. Let's make one.

We'll call this utility VUBYTE, since it will let the user *view* a given *byte*. With VUBYTE, a calling program must take only three steps to display a byte in hexadecimal representation anywhere on the screen:

1) Set a zero-page pointer (TV.PTR) to point to the screen location where the byte should be displayed; 2) load the accumulator with the byte to be displayed; and then 3) call VUBYTE.

Figure 5.1 shows how VUBYTE will work.

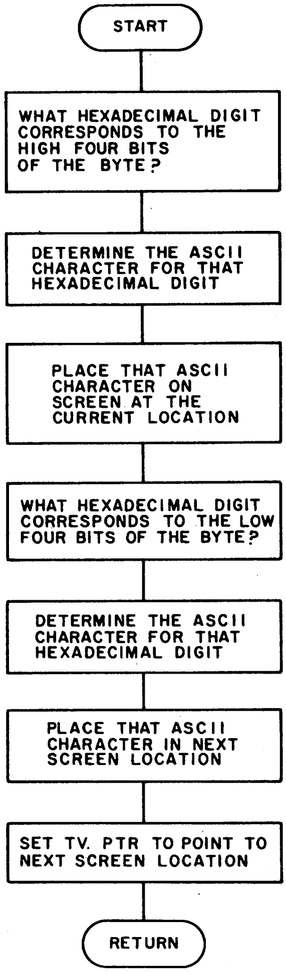

Figure 5.1: *Flowchart of the routine VUBYTE, which displays a byte in hexadecimal representation on the video screen.*

VUBYTE will display the given byte as two ASCII characters in the current position on the screen, and when VUBYTE returns, TV.PTR will be pointing to the screen location immediately following the two screen locations occupied by the displayed characters.

VUBYTE need only determine the ASCII character for the hexadecimal value of the 4 MSB (most-significant bits), store that ASCII character in the screen location pointed to by TV.PTR, then display the ASCII character for the hexadecimal value of the accumulator's 4 LSB (least-significant bits) in the next screen location. See figure 5.1 for a flowchart outlining this.

VUBYTE seems to be asking for a utility subroutine to return the ASCII character for a given 4-bit value. Let's call this subroutine ASCII. ASCII will return the ASCII character for the hexadecimal value represented by the 4 least-significant bits in the accumulator. It will ignore the 4 most-significant bits in the accumulator.

If we assume that ASCII exists, then we can write VUBYTE:

VUBYTE

VUBYTE	PHA	Save accumulator.
	LSR A LSR A LSR A LSR A	Move 4 MSB into positions occupied by 4 LSB.
	JSR ASCII	Determine ASCII for accumulator's 4 LSB (which *were* its 4 MSB).
	JSR VUCHAR	Display the ASCII character in the current screen location and advance to next screen location.
	PLA	Restore original value of accumulator.
	JSR ASCII	Determine ASCII for accumulator's 4 LSB (which were its 4 LSB).
	JSR VUCHAR	Display this ASCII character just to the right of the other ASCII character and advance to next screen location.
	RTS	Return to caller.

Of course, ASCII doesn't exist yet. So let's write it, and then VUBYTE should be complete.

ASCII

ASCII	AND #$0F	Clear the 4 MSB in accumulator.
	CMP #$0A	Is accumulator greater than 9?
	BMI DECIML	
	ADC #6	If so, it must be A thru F. Add $36 to accumulator to convert it to corresponding ASCII character. (We'll add $36 by adding $6 and then adding $30.)
DECIML	ADC #$30	If accumulator is 0 thru 9, add $30 to it to convert it to corresponding ASCII character.
	RTS	Return to caller, bearing the ASCII character corresponding to the hexadecimal value initially in the 4 LSB of the accumulator.

TVHOME, CENTER

Now we can display a character or a byte at the current screen location, and we can set the current screen location to any given X,Y coordinates or modify it relative to its current value. It would also be handy if we could set the TV.PTR to certain fixed locations: locations that more than one calling program might need as points or origin. For example, a calling program might need to set the TV.PTR to the HOME location (position 0,0), or to the CENTER of the screen:

TVHOME, CENTER

TVHOME	LDX #0	Set TV.PTR to the leftmost column
	LDY #0	of the top row
	JSR TVTOXY	of the screen.
	RTS	Then return to caller.

CENTER	LDA TVROWS LSR A TAY	Load A with total rows. Divide it by two. Y now holds the number of the central row on the screen.
	LDA TVCOLS LSR A TAX	Load A with total columns. Divide it by two. X now holds the number of the central column on the screen.
		Now X and Y registers hold X, Y coordinates of center of screen.
	JSR TVTOXY	Set the TV.PTR to X,Y coordinates.
	RTS	Return to caller.

TVPUSH, TV.POP

The screen utilities presented thus far enable us to set or modify the current position on the screen. We might also want to save the current position on the screen and then restore that position later. We can do this by pushing TV.PTR onto the stack and then pulling it from the stack:

TVPUSH

TVPUSH	PLA	Pull return address from stack.
	TAX PLA TAY	Save it in X... ...and in Y.
	LDA TV.PTR+1 PHA LDA TV.PTR PHA	Get TV.PTR and save it on the stack.
	TYA PHA TXA PHA	Place return address back... ... on stack.
	RTS	Then return to caller.

TVPOP

```
TV.POP      PLA              Pull return address from stack.
            TAX              Save it in X...
            PLA
            TAY              ...and in Y.

            PLA              Restore...
            STA TV.PTR       ...TV.PTR
            PLA                ...from
            STA TV.PTR+1         ...stack.

            TYA              Place return
            PHA              address back...
            TXA
            PHA              ... on stack.

            RTS              Then return to caller.
```

Now a calling program can save its current screen position with one line of source code: "JSR TVPUSH." That calling program can then modify TV.PTR and later restore it to its saved value with one line of source code: "JSR TV.POP."

CLEAR SCREEN

Now that we can set TV.PTR to any X,Y location on the screen, and display any byte or character in the current location, let's write some code to clear all or part of the screen. One subroutine, CLR.TV, will clear all of the video screen for us while preserving the zero page. A second routine, CLR.XY, will start from the current screen location and clear a rectangle, whose X,Y dimensions are given by the X,Y registers. Thus, a calling program can call CLR.TV to clear the whole screen; or a calling program can clear any rectangular portion of the screen, leaving the rest of the screen unchanged, just by making TV.PTR point to the upper left-hand corner of the rectangle to be cleared, and then calling CLR.XY with the X and Y registers holding, respectively, the width and height of the rectangle to be cleared.

```
CLR.TV    JSR TVPUSH        Save the zero-page bytes that will be
                            changed.
          JSR TVHOME        Set the screen location to upper-left cor-
                            ner of the screen.
```

```
         LDX TVCOLS       Load X,Y registers with
         LDY TVROWS       X,Y dimensions of the screen.
         JSR CLR.XY       Clear X columns, Y rows from current
                          screen location.
         JSR TV.POP       Restore zero-page bytes that were
                          changed.
         RTS              Return to caller, with screen clear and
                          with zero page preserved.

CLR.XY   STX COLS         Set the number of columns to be
                          cleared.
         TYA
         TAX              Now X holds the number of rows to be
                          cleared.
CLRROW   LDA BLANK        Load accumulator with your system's
                          graphic code for a blank.
         LDY COLS         Load Y with number of columns to be
                          cleared.
CLRPOS   STA (TV.PTR),Y   Clear a position by writing a blank into
                          it.
         DEY              Adjust index for next position in the
                          row.
         BPL CLRPOS       If not done with row, clear next posi-
                          tion...
         JSR TVDOWN       If done with row, move current screen
                          location down by one row.
         DEX              Done last row yet?
         BPL CLRROW       If not, clear next row...
         RTS              If so, return to caller.
COLS     .BYTE 0          Variable: holds number of columns to
                          be cleared.
```

There are many more screen utilities you could develop, but the utilities presented in this chapter are a good basic set. Now programs can call the following subroutines to perform the following functions:

ASCII:	Return ASCII character for 4 LSB in A.
CENTER:	Set current screen position to center of screen.
CLR.TV:	Clear the entire video display, preserving TV.PTR.
CLR.XY:	Clear a rectangle of the screen, with X,Y dimensions specified by the X,Y registers.
TVDOWN:	Move current screen position down by one row.

TVHOME:	Set current screen position to the upper-left corner of the screen.
TVPLUS:	Add A to TV.PTR.
TV.POP:	Restore previously saved screen position from stack.
TVPUSH:	Save current screen location on stack.
TV.PUT:	Display ASCII character in A at current screen location.
TVSKIP:	Advance to next screen location.
TVTOXY:	Set current screen position to X,Y coordinates given by X,Y registers.
VUBYTE:	Display A, in hexadecimal form, at current screen location. Advance current screen location past the displayed byte.
VUCHAR:	Display A as an ASCII character in current screen location; then advance to next screen location.

With these screen utilities, a calling program can drive the screen display without ever dealing directly with screen memory or even with the zero page. The calling program need not concern itself with anything other than the current position on the screen, which can be dealt with as a concept, rather than as a particular address hard-wired into the code.

Chapter 6:

The Visible Monitor

Hand Assembling Object Code

An assembler is a wonderful software tool, but what if you don't have one? Is it possible to write 6502 code without an assembler?

You bet!

Not only is it possible to write machine code by hand, but *all* of the software in this book was originally assembled and entered into the computer by hand. In fact, I hand assembled my code long after I had purchased a cassette-based assembler, because I could hand assemble a small subroutine faster than I could load in the entire assembler.

Hand assembling code imposes a certain discipline on the programmer. Because branch addresses must be calculated by counting forward or backward in hexadecimal, I tried to keep my subroutines very small. (How far can *you* count backward in hexadecimal?) I wrote programs as many nested subroutines, which I could assemble and test individually, rather than as monolithic, in-line code. This is a good policy even for programmers who have access to an assembler, but it is essential for any programmer who must hand assemble code.

Yet once you've written a program consisting of machine-language instructions, how can you enter it into memory? You can read your program on paper, but how can you present it to the 6502?

A program called a *machine-language monitor* allows you to examine and modify memory. It also allows you to execute a program stored in memory. The Commodore 64 and VIC-20 do not feature a built-in (ROM) machine language monitor. Very good monitors are available on plug-in cartridges and disks, but these can cost $40 or more. So before you run out and buy a full-featured machine language monitor, let's take a look at a very simple monitor.

We'll look at a monitor stored in ROM on the Ohio Scientific (OSI) Challenger I-P. It is presented here only for purposes of illustration, since it is not available for Commodore computers.

A Minimal Machine-Language Monitor

You can invoke the OSI ROM monitor quite easily by pressing the BREAK key and then the "M" key. The monitor clears the video screen and presents the display shown in figure 6.1.

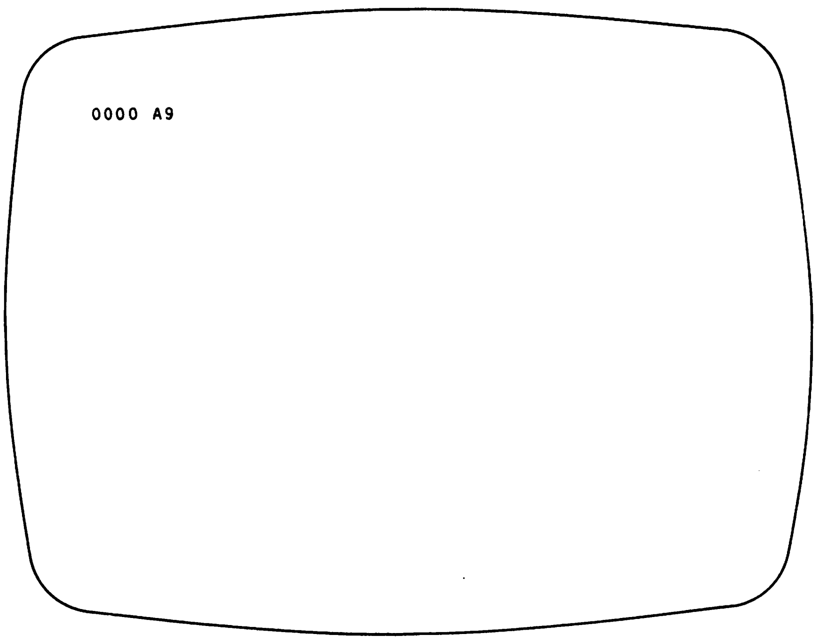

Figure 6.1: *Ohio Scientific ROM (read-only memory) monitor display.*

The display consists of two fields of hexadecimal characters: an address field and a data field. Figure 6.1 indicates that $A9 is the current value of address $0000.

The OSI ROM monitor has two modes: *address mode* and *data mode*. When the monitor is in address mode, you can display the contents of any address simply by typing the address on the keyboard. Each new hexadecimal character will roll into the address field from the right. To display address $FE0D, you simply type the keys F, E, 0, and then D.

To change the contents of an address, you must enter the *data* mode. When the

OSI ROM monitor is in the data mode, hexadecimal characters from the keyboard will roll into the data field on the screen. For your convenience, when the monitor is in the data mode you can step forward through memory (ie: increment the displayed address) by depresssing the RETURN key. Unfortunately, this convenience is not available in address mode, and neither mode allows you to step backward through memory (ie: to decrement the address field).

Beware: the OSI ROM monitor can mislead you. If the monitor is in the data mode and you type a hexadecimal character on the keyboard, that character will roll into the data field on the screen. Presumably that hexadecimal character also rolls into the memory location displayed on the screen. Yet, this might not be the case. In fact, the OSI ROM monitor displays the data you *intended* to store in an address, rather than the actual contents of that address. If you try to store data in a read-only memory address, for example, the OSI ROM monitor will confirm that you've stored the intended data in the displayed address, yet if you actually inspect that address (by entering address mode and typing in the address), you'll see that you changed nothing. This makes sense — you can't write to read-only memory. But the OSI ROM monitor leads you to think that you can.

The OSI ROM monitor can be confusing in other ways. For example, the display does not tell you whether you're in data mode or address mode; you've got to remember at all times which mode you last told the monitor to use. Furthermore, to escape from address mode you must use one key, while to escape from data mode you must use another key. Therefore you must always remember two escape codes as well as the current mode of the monitor.

Furthermore, the OSI ROM monitor does not make it very easy for you to enter ASCII data into memory. To enter an ASCII message into memory, you must consult an ASCII table (such as Appendix A2 in this book), look up the hexadecimal representation of each character in your message, and then enter each of those ASCII characters via two hexadecimal keystrokes. Then, once you've got an ASCII message in memory, the OSI ROM monitor won't let you read it as English text; you'll have to view that message as a series of bytes in hexadecimal format, and then look up, again in Appendix A2 or its equivalent, the ASCII characters defined by those bytes. That won't encourage you to include a lot of messages in your software — even though meaningful prompts and error messages can make your software much easier to maintain and use.

Finally, it is worth examining the way the OSI ROM monitor executes programs in memory. When you type "G" on the Ohio Scientific Challenger I-P, the OSI ROM monitor executes a JMP (unconditional jump) to the displayed address. That transfers control to the code selected, but it does so in such a way that the code must end with another unconditional jump if control is to return to the OSI ROM monitor. This forces you to write programs that end with a JMP, rather than subroutines that end with an RTS.

Programs that end with a JMP are not used easily as building blocks for other programs, whereas *subroutines* are incorporated quite easily into software structures of ever-greater power. So wouldn't it be nice if a machine-language monitor executed a JSR to the displayed address? This would call the displayed address as a subroutine, encouraging users to write software as subroutines, rather than as code that jumps from place to place. Such a monitor might actually encourage good programming habits, inviting the user to program in a structured manner, rather than daring the user to do so. In this chapter we'll develop such a monitor.

Objectives

If you've spent any time using a minimal machine-language monitor, you've probably thought of some ways to improve it. Based on my own experience, I knew that I wanted a monitor to be:

1) Accurate

The data field should display the actual contents of the displayed address, not the *intended* contents of that address.

2) Convenient

It should be possible to step forward or backward through memory, in any mode. It should also be possible to enter ASCII characters into memory directly from the keyboard, without having to look up their hexadecimal representations first, and it should be possible to display such characters as ASCII characters, rather than as bytes presented as pairs of hexadecimal digits.

3) Encourage Structured Programming

The monitor should *call* the displayed address as a subroutine, rather than *jump* to the displayed address. This will encourage the user to write subroutines, rather than monolithic programs that jump from place to place.

4) Simplify Debugging

The monitor should load the 6502 registers with user-defined data before calling the displayed address. Thus a user can initially test a subroutine with different values in the registers. Then, when the called subroutine returns, the monitor should display the new contents of the 6502 registers. Thus, by seeing how it changes or preserves the values of the 6502 registers, the user could judge the performance of the subroutine.

Because my objective was to make the 6502 registers visible to the user by displaying the 6502 registers before and after any subroutine call, I've chosen to call this monitor the *Visible Monitor*. Figure 6.2 shows its display format.

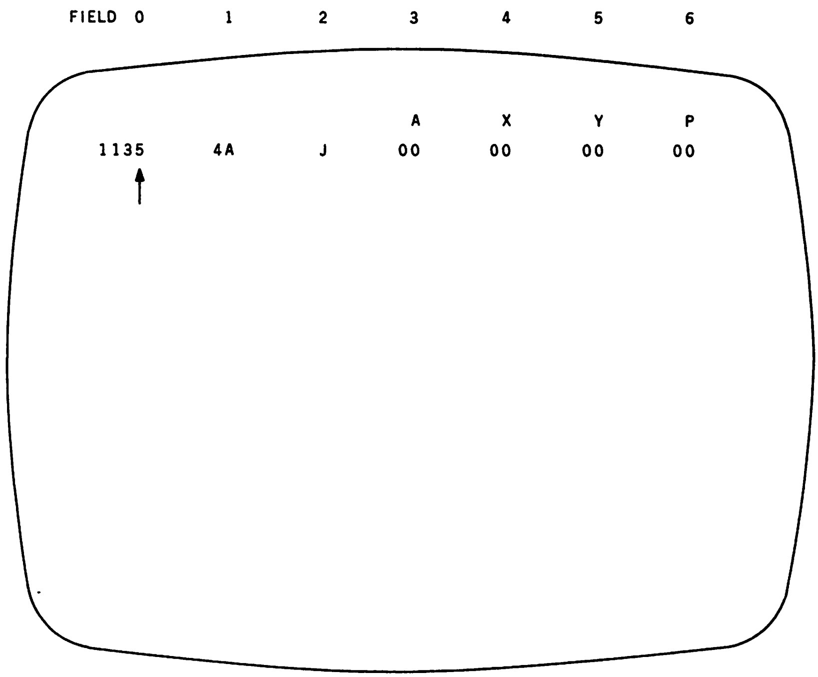

Figure 6.2: *Visible Monitor Display with fields numbered.*

VISIBLE MONITOR DISPLAY

The Visible Monitor Display

Notice that the display in figure 6.2 has seven fields, not two as in the OSI ROM monitor display. The first two fields (fields 0 and 1) are the same as the two fields in the OSI ROM monitor — that is, they display an address and a hexadecimal representation of the contents of that address. Field 2 is a graphic representation of the contents of the displayed address. If that address holds an ASCII character, then the graphic will be the letter, number, or punctuation mark specified by the byte. Otherwise, that graphic will probably be a special graphic character from your computer's nonstandard (ie: nonASCII) character set.

Fields 3 thru 6 represent four of the 6502 registers: A (the Accumulator), X (the X Register), Y (the Y Register), and P (the Processor Status Register). When you type

G to execute a program, the 6502 registers will be loaded with the displayed values before the program is called; when control returns to the monitor, the contents of the 6502 registers at that time will be displayed on the screen.

In addition to the seven fields mentioned above, the Visible Monitor's display includes an arrow pointing up at one of the fields. In order to modify a field, you must make the arrow point to that field. To move the arrow from one field to another, I've chosen to use the RIGHT-ARROW and LEFT-ARROW keys. Touching the RIGHT-ARROW key will move the arrow one field to the right, and depressing the LEFT-ARROW key will move the arrow one field to the left. (Note that the RIGHT-ARROW and LEFT-ARROW keys are both on the same *physical* key. Press that key *without* shifting to move to the right; press it while *holding down* the SHIFT key to move to the left.)

I've chosen to use the space bar to step forward through memory and the return key to step backward through memory, but you may choose other keys if you prefer (eg: the "+" and "−" keys). The space bar seems reasonable to me for stepping forward through memory, because on a typewriter I press the space bar to bring the *next* character into view; RETURN seems reasonable for stepping backward through memory because RETURN is almost synonymous with "back up," and that's what I want it for: to back up through memory. With such a display and key functions, we ought to have a very handy monitor.

Data

Before we develop the structure and code of the Visible Monitor, let's decide what variables and pointers it must have.

The Visible Monitor must have some way of knowing what address to display in field 0. It can do this by maintaining a pointer to the currently selected address. Because it will specify the currently selected address, let's call this pointer SELECT. Then, when the user presses the spacebar, the Visible Monitor need only increment the SELECT pointer. When the user presses RETURN, the Visible Monitor need only decrement the SELECT pointer. That will enable the user to step forward and backward through memory.

The user will also want to modify the 6502 register images. Since there are four register images shown in figure 6.2, let's have 4 bytes, one for each register image. If we keep them in contiguous memory, we can refer to the block of register images as REGISTERS, or simply as REGS (since REGISTERS is longer than six characters, the maximum label length acceptable to the assembler used in the preparation of this book).

Finally, the Visible Monitor must keep track of the current field. Since there can only be one current field at a time, we can have a variable called FIELD, whose value tells us the number of the current field. Then, when the user wants to select the next field, the Visible Monitor need only increment FIELD, and when the user wants to move the arrow to the previous field, the Visible Monitor need only decrement FIELD. If FIELD gets out of bounds (any value that is not 0 thru 6), then the Visible Monitor should assign an appropriate value to FIELD. The following code declares these variables in the form acceptable to an OSI 6500 Assembler:

Variables

```
SELECT      .WORD 0       This points to the currently selected
                          byte.
REG.A       .BYTE 0       REG.A holds the image of Register A
                          (the Accumulator).
REG.X       .BYTE 0       REG.X holds the image of Register X.
REG.Y       .BYTE 0       REG.Y holds the image of Register Y.
REG.P       .BYTE 0       REG.P holds the image of the Processor
                          Status Register.
FIELD       .BYTE 0       FIELD holds the number of the current
                          field.
            REGS = REG.A
```

Structure

I want to keep the Visible Monitor highly modular, so it can be easily extended and modified. I have therefore chosen to develop the Visible Monitor according to the structure shown in figure 6.3. Clearly, the Visible Monitor loops. It places the monitor *display* on the screen. It then *updates* the information in that display by getting a keystroke from the user and performing an action based on that keystroke. It does this over and over.

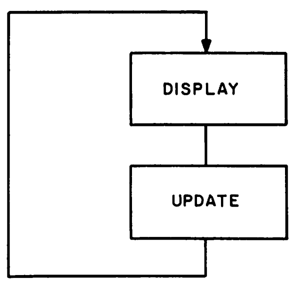

Figure 6.3: *A simple structure for interactive display programs.*

With this flowchart as a guide, we can now write the source code for the top level of the Visible Monitor:

VISMON

```
VISMON    PHP            Save caller's status flags.
LOOP      JSR DSPLAY     Put monitor display on screen.
          JSR UPDATE     Get user request and handle it.
          CLC
          BCC LOOP       Loop back to display...
```

This is only the top level of the Visible Monitor; it won't work without two subroutines: DSPLAY and UPDATE. So it looks as if we've traded the task of writing one subroutine for the task of writing two. But by structuring the monitor in this way, we make the monitor much easier to develop, document, and debug.

Which subroutine should we write first? Let's start with the DSPLAY module, since the display is visible to the user, and the Visible Monitor must meet the user's needs. Once we know how to drive the display, we can write the UPDATE routine.

Monitor Display

Figure 6.2 shows the display we want to present on the video screen. As you can see, this display consists of three lines of characters: the label line, the data line, and the arrow line. The label line labels four of the fields in the data line, using the characters A, X, Y, and P. The data line displays an address, the contents of that address (both in hexadecimal representation and in the form of a graphic), and then displays the values of the four registers in the 6502. Underneath the data line, the arrow line provides one arrow pointing up at one of the fields in the data line.

Since the display is defined totally in terms of the label line, the data line, and the arrow line, we are ready now to diagram the top level of monitor display. See figure 6.4.

With the flowchart in figure 6.4 as a guide, we can now write source code for the top level of the DSPLAY subroutine:

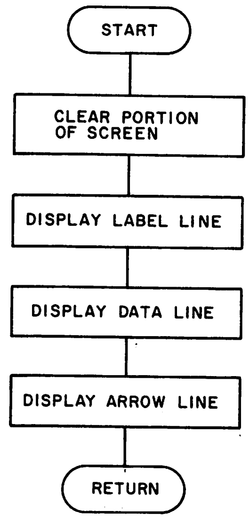

Figure 6.4: *Routine to display the monitor information.*

DSPLAY

```
DSPLAY      JSR CLRMON      Clear monitor's portion of screen.
            JSR LINE.1      Display the Label Line.
            JSR LINE.2      Display the Data Line.
            JSR LINE.3      Display the Arrow Line.
            RTS             Return to caller.
```

Now instead of one subroutine (DSPLAY), it looks as if we must write four subroutines: CLRMON, LINE.1, LINE.2, and LINE.3. But as the subroutines grow in number, they shrink in difficulty.

Before we put up any of the monitor's display, let's clear that portion of the screen used by the monitor's display. Then we can be sure we won't have any garbage cluttering up the monitor display.

Since we already have a utility to clear X columns and Y rows from the current location on the screen, CLRMON can just set TV.PTR to the upper-left corner of the screen, load X and Y with appropriate values, and then call CLR.XY. Here's source code:

```
CLRMON      LDX #2          Set TV.PTR to column 2, row 2 of
            LDY #2          screen.
            JSR  TVTOXY
            LDX  TVCOLS     We'll clear the full width of the
            LDY #3          screen for 3 rows.
            JSR  CLR.XY     Here we clear them.
            RTS             Return to caller.
```

Display Label Line

The subroutine LINE.1 must put the label line onto the screen. We'll store the character string "A X Y P" somewhere in memory, at a location we may refer to as LABELS. Then LINE.1 need only copy 10 bytes from LABELS to the appropriate location on the screen. That will display the LABEL line for us:

LINE.1

```
LINE.1      LDX #11         X-coordinate of Label "A".
            LDY #0          Y-coordinate of Label "A".
            JSR  TVTOXY     Place TV.PTR at coordinates given by
                            X,Y registers.
            LDY #0          Put labels on the screen:
            STY LBLCOL      Initialize label column counter.
LBLOOP      LDA LABELS,Y    Get a character and
            JSR VUCHAR      put its graphic on the screen.
            INC LBLCOL      Prepare for next character.
            LDY LBLCOL      Use label column as an index.
            CPY #10         Done last character?
            BNE LBLOOP      If not, do next one.
            RTS             Return to caller.
LABELS      .BYTE 'A X '    These are the characters
            .BYTE 'Y P'     to be copied to the screen.
LBLCOL      .BYTE 0         This is a counter.
```

Display Data Line

Displaying the data line will be more difficult than displaying the label line, for two reasons. First, the data to be displayed will change from time to time, whereas the labels in the label line need never change. Second, most fields in the data line display data in hexadecimal representation. To display 1 byte as two hexadecimal digits requires more work than is needed to display 1 byte as one ASCII character. However, we have a screen utility (VUBYTE) to do that work for us. In fact, we have enough screen utilities to make even the display of seven fields of data quite straightforward. Following, then, is the display data-line routine:

LINE.2

```
LINE.2    LDX #0           Load X register with X-coordinate for
                           start of data line.
          LDY #1           Load Y register with Y-coordinate for
                           data line.
          JSR  TVTOXY      Set TV.PTR to point to the start of the
                           data line.
          LDA SELECT+1     Display high byte of the
          JSR VUBYTE       currently selected address.
          LDA SELECT       Display low byte of the
          JSR VUBYTE       currently selected address.
          JSR TVSKIP       Skip one space after address field.
          JSR GET.SL       Look up value of the currently selected
                           byte.
          PHA              Save it.
          JSR VUBYTE       Display it, in hexadecimal format, in
                           field 1.
          JSR TVSKIP       Skip one space after field 1.
          PLA              Restore value of currently selected byte.
          JSR VUCHAR       Display that byte, in graphic
                           form, in field 2.
          JSR TVSKIP       Skip one space after field 2.
                           Display 6502 register images in fields 4
                           thru 7:
          LDX #0
VUREGS    LDA REGS,X       Look up the register image.
          JSR VUBYTE       Display it in hexadecimal format.
          JSR TVSKIP       Skip one space after hexadecimal field.
          INX              Get ready for next register...
          CPX #4           Done 4 registers yet?
          BNE VUREGS       If not, do next one...
          RTS              If all registers displayed, return.
```

Get Currently Selected Byte

Note that the subroutine LINE.2, which puts up the second line of the Visible Monitor's display, does not itself "know" the value of the currently selected byte. Rather, it calls a subroutine, GET.SL, which returns the contents of the address pointed to by SELECT. That makes life easy for LINE.2, but how does GET.SL work?

If SELECT were a zero-page pointer, GET.SL could be a very simple subroutine and take advantage of the 6502's indirect addressing mode:

```
GET.SL    LDY #0             Get the zeroth byte above
          LDA (SELECT),Y     the address pointed to by SELECT.
          RTS                Return to caller.
```

However, SELECT is not a zero-page pointer; it's up in page $32. And the 6502 doesn't have an addressing mode that will let us load a register using any pointer not in the zero page. So how can we see what's in the address pointed to by SELECT?

We can do it in two steps. First, we'll set a zero-page pointer equal in value to the SELECT pointer, so it points to the same address; and then, since we already know how to load the accumulator using a zero-page pointer, we'll load the accumulator using the zero-page pointer that now equals SELECT. Let's call that zero-page pointer GETPTR, since it will allow us to *get* the selected byte. Using such a strategy, GET.SL can look like this:

```
GET.SL    LDA SELECT         Set GETPTR equal to
          STA GETPTR         SELECT: first the low byte;
          LDA SELECT+1       then the
          STA GETPTR+1       high byte.
          LDY #0             Get the zeroth byte above
          LDA (GETPTR),Y     the address pointed to by GETPTR.
          RTS                Return to caller, with A bearing the con-
                             tents of the address specified by
                             SELECT.
```

This second attempt at GET.SL will load the accumulator with the currently selected byte, even when SELECT is not in the zero page. However, beware because by setting GETPTR equal to SELECT, GET.SL changes the value of GETPTR. This can be very dangerous. What, for example, if some other program were using GETPTR for something? That other program would be sabotaged by GET.SL's actions. If we let GET.SL change the value of GETPTR, then we must make sure that no other program ever uses GETPTR.

Such policing is hard work — and almost impossible if you want your software to run on a system in conjunction with software written by anyone else. Since I want the Visible Monitor to share your system's ROM input/output routines, and since I have no way of knowing what zero-page addresses those routines may use, I must refrain from using any of those zero-page bytes myself. When I have to use zero-page bytes — as now, so that GET.SL can use the 6502's indirect addressing mode — I must restore any zero-page bytes I've changed.

Therefore, GET.SL must be a four-part subroutine, which will: 1) save GETPTR; 2) set GETPTR equal to SELECT; 3) load the accumulator with the contents of the address pointed to by GETPTR; and finally, 4) restore GETPTR to its original value. This larger, slower, but infinitely safer version of GET.SL looks like this:

```
GET.SL    LDA GETPTR        Save GETPTR
          PHA               on stack and
          LDX GETPTR+1      in X register.

          LDA SELECT        Set GETPTR
          STA GETPTR        equal to
          LDA SELECT+1      SELECT.
          STA GETPTR+1

          LDY #0            Get the contents of the
          LDA (GETPTR),Y    byte pointed to by SELECT,
          TAY               and save it in Y register.

          PLA               Restore GETPTR
          STA GETPTR        from stack
          STX GETPTR+1      and from X register.
          TYA               Restore contents of current byte from
                            temporary storage in Y to A.
          RTS               Return with contents of currently
                            selected byte in accumulator and with
                            the zero page preserved.
```

Display Arrow Line

This routine displays an up-arrow directly underneath the current field:

LINE.3

```
LINE.3      LDY   FIELD       Look up current field.
            SEC               If it is out of bounds,
            CPY   #7          set it to
            BCC FLD.OK        default field
            LDY #0            (the address field).
            STY FIELD
FLD.OK      LDA   FIELDS,Y    Look up column number for current
                              field.
            TAX               That will be the arrow's
                              X-coordinate.
            LDY #2            Set arrow's Y-coordinate.
            JSR   TVTOXY      Make TV.PTR point to arrow
                              location.

            LDA   ARROW       Place an up-arrow in
            JSR   VUCHAR      that location.
            RTS               Return to caller.
FIELDS      .BYTE 3,6,8       This data area shows which column
            .BYTE $0B,$0E     should get an up-arrow to indicate
            .BYTE $11,$14     any one of fields 0 thru 6. Changing one
                              of these values will cause the up-arrow
                              to appear in a different column when in-
                              dicating a given field.
```

Now that we have all the routines we need for the monitor display, let us look at how they fit together to form a structure. Here is the hierarchy of subroutines in DSPLAY:

```
MONITOR DISPLAY
        DISPLAY LABEL LINE
        DISPLAY DATA LINE
              GET.SL
              VUBYTE
                   ASCII
                   TVPLUS
              TVSKIP
        DISPLAY ARROW LINE
```

When DSPLAY is called, it will clear the top four rows of the screen, display labels, data, the arrow, and then return. How long do you think it will take to do all this? The code may look cumbersome, but it executes in the blink of an eye!

Monitor Update

The UPDATE routine is the monitor subroutine that executes functions in response to various keys. The basic key functions we want to implement are as follows:

Key	Function
RIGHT-ARROW	Move arrow one field to the right.
LEFT-ARROW	Move arrow one field to the left.
SPACEBAR	Increment address being displayed. (Step forward through memory.)
RETURN	Decrement address being displayed. (Step backward through memory.)

If the arrow is in fields 1, 3, 4, 5, or 6, then, for

keys 0 thru 9, A thru F	Roll a hexadecimal character into the field pointed to by the arrow.

If the arrow is under field 2 (the graphic field) then, for

All keys	Enter the key's character into field 2 (ie: enter the key's character into the displayed address).

Since the video display need not be refreshed (redisplayed within a given time) by the processor, the UPDATE routine need not return within a given amount of time. The UPDATE routine, therefore, can wait indefinitely for a new character from the keyboard, and then take appropriate action.

We can diagram these functions as shown in figure 6.5. You add additional functions to this routine by adding additional code to test the input character. You then call the appropriate function subroutine which you write.

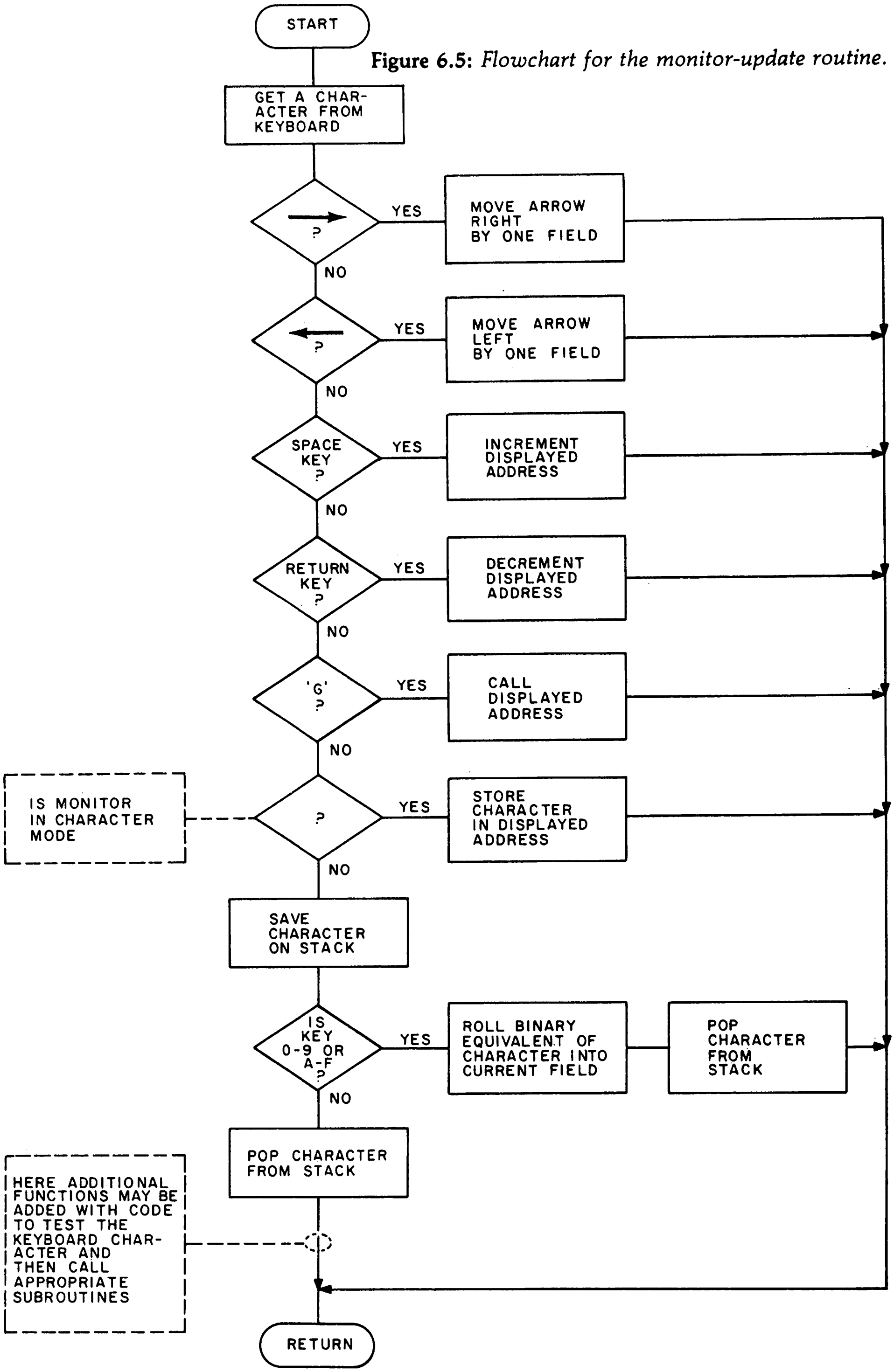

Figure 6.5: *Flowchart for the monitor-update routine.*

Get a Key

First we need a way to get a key from the keyboard. Of course, your system has a read-only memory routine to perform this function. As you can see in appendices B1 and B2, we have placed the address of that routine into a pointer called ROMKEY located at address $3008. Once you have set the ROMKEY pointer, you can get a key by calling a subroutine labeled GETKEY, which simply transfers control to the ROM routine whose address you placed in ROMKEY:

```
GETKEY     JMP (ROMKEY)
```

Now that we have a way to get a key from the keyboard, we should be able to write source code for the monitor-update routine:

Update

```
UPDATE     JSR GETKEY        Get a character from the keyboard.
IF.GRTR    CMP #$1D          Is it the RIGHT-ARROW key?
           BNE IF.LSR        If not, perform text test.
NEXT.F     INC FIELD         If so, select the next field.
           LDA FIELD         If arrow was at the right-most field,
           CMP #7            place it underneath the left-most
           BNE EXIT.1        field.
           LDA #0
           STA FIELD
EXIT.1     RTS               Then return.
IF.LSR     CMP #$9D          Is it the LEFT-ARROW key?
           BNE IF.SP         If not, perform next test.
PREV.F     DEC FIELD         If so, select previous field:
           BPL EXIT.2        the field to the left of the
           LDA #6            current field. If arrow was at
           STA FIELD         left-most field, place it under
                             right-most field.
EXIT.2     RTS               Then return.
IF.SP      CMP #SPACE        Is it the space bar?
           BNE IF.CR         If not, perform next test.
INC.SL     INC SELECT        If so, step forward through
           BNE EXIT.3        memory, by incrementing the
           INC SELECT+1      pointer that specifies the displayed
                             address.
EXIT.3     RTS               Then return
IF.CR      CMP #CR           Is it carriage return?
           BNE IFCHAR        If not, perform next test.
```

```
DEC.SL     LDA SELECT       If so, step backward through
           BNE NEXT.1       memory by decrementing the
           DEC SELECT+1     pointer that selects the
NEXT.1     DEC SELECT       address to be displayed.
           RTS              Then return.
IFCHAR     LDX FIELD        Is arrow underneath the
           CPX #2           character field (field 2)?
           BNE IF.GO        If not, perform next test.
                            Put the contents of A into the currently
                            selected address.
PUT.SL     TAY              Use Y to hold the character we'll put in
                            the selected address.
           LDA TV.PTR       Save zero-page pointer TV.PTR
           PHA              on stack and in X before we
           LDX TV.PTR+1     use it to put character in selected ad-
                            dress.
           LDA SELECT       Set TV.PTR equal to SELECT,
           STA TV.PTR       so it points to the
           LDA SELECT+1     currently selected
           STA TV.PTR+1     address.
           TYA              Restore to A the character we'll put in
                            the selected address.
           LDY #0           Store it in the
           STA (TV.PTR),Y   selected address.
           STX TV.PTR+1     Restore TV.PTR to
           PLA              its original value.
           STA TV.PTR
           RTS              Return to caller, with character origi-
                            nally in A now in the selected address
                            and with zero page unchanged.

IF.GO      CMP #'G          Is it 'G' for GO?
           BNE IF.HEX       If not, perform next test.
GO         LDY REG.Y        If so, load the 6502 registers
           LDX REG.X        with their displayed images.
           LDA REG.P
           PHA
           LDA REG.A
           PLP
           JSR CALLSL       Call the subroutine at the selected ad-
                            dress.
           PHP              When subroutine returns,
           STA REG.A        save register values in register
           STX REG.X        images.
```

	STY REG.Y	
	PLA	
	STA REG.P	
	RTS	Then return to caller.
CALLSL	JMP (SELECT)	Call the subroutine at the selected address.
IF.HEX	PHA	Save keyboard character.
	JSR BINARY	If accumulator holds ASCII character for 0 thru 9 or A thru F, BINARY returns the binary representation of that hexadecimal digit. Otherwise BINARY returns with A = FF and the minus flag set.
	BMI OTHER	If accumulator did not hold a hexadecimal character, perform next test.
	TAY	
	PLA	
	TYA	
ROLLIN	LDX FIELD	Roll A into a hexadecimal field.
	BNE NOTADR	Is arrow underneath the address field (field 0)? If not, the arrow must be under another hexadecimal field.
ADRFLD	LDX #3	Since arrow is underneath the address field, roll accumulator's hexadecimal digit into the address field by rolling it into the pointer that selects the displayed address.
LOOP.1	CLC	
	ASL SELECT	
	ROL SELECT+1	
	DEX	
	BPL LOOP.1	
	TYA	
	ORA SELECT	
	STA SELECT	
	RTS	Then return.
NOTADR	CPX #1	Is arrow underneath field 1?
	BNE REGFLD	If not, it must be underneath a register image.
ROL.SL	AND #$0F	Roll A's 4 LSB into contents of currently selected byte.
	PHA	
	JSR GET.SL	Get the contents of the selected address and shift left 4 times.
	ASL A	
	ASL A	
	ASL A	
	ASL A	
	AND #$F0	
	STA TEMP	Save it in a temporary variable.

```
          PLA               Get original A's 4 LSB and
          ORA TEMP          OR them with shifted contents of
                            selected address.
          JSR PUT.SL        Store the result in the selected
          RTS               address and return.

TEMP      .BYTE 0           This byte holds the temporary variable
                            used by ROL.SL.
REGFLD    DEX               The arrow must be underneath a
          DEX               register image — field 3, 4, 5, or 6.
          DEX
          LDY #3
LOOP.2    CLC               Roll accumulator's hexadecimal digit
          ASL REGS,X        into appropriate register image...
          DEY
          BPL LOOP.2
          ORA REGS,X
          STA REGS,X
          RTS               ...Then return.
OTHER     PLA               Restore the raw keyboard character that
                            we saved on the stack.
          CMP#'Q            Is it 'Q' for Quit?
          BNE NOT.Q         If not, perform next test.
          PLA               If so, return to
          PLA               the caller of
          PLP
          RTS               VISMON.
NOT.Q     JSR DUMMY         Replace this call to DUMMY with a call
                            to any other subroutine that extends the
                            functionality of the Visible Monitor.
DUMMY     RTS               Return to caller.
```

ASCII to BINARY Conversion

The Visible Monitor's UPDATE subroutine requires a subroutine called BINARY, which will determine if the character in the accumulator is an ASCII 0 thru 9 or A thru F, and, if so, return the binary equivalent. On the other hand, if the accumulator does not contain an ASCII 0 thru 9 or A thru F, BINARY will return an error code, $FF. Thus:

If accumulator holds	BINARY will return
$30 (ASCII "0")	$00
$31 (ASCII "1")	$01
$32 (ASCII "2")	$02
$33 (ASCII "3")	$03
$34 (ASCII "4")	$04
$35 (ASCII "5")	$05
$36 (ASCII "6")	$06
$37 (ASCII "7")	$07
$38 (ASCII "8")	$08
$39 (ASCII "9")	$09
$41 (ASCII "A")	$0A
$42 (ASCII "B")	$0B
$43 (ASCII "C")	$0C
$44 (ASCII "D")	$0D
$45 (ASCII "E")	$0E
$46 (ASCII "F")	$0F
Any other value	$FF

We could solve this problem with a table, BINTAB, for BINary TABle. If BINTAB is at address $2000, then $2000 would contain a $FF, as would $2001, $2002, and all addresses up to $202F, because none of the ASCII codes from $00 thru $2F represent any of the characters 0 thru 9 or A thru F. On the other hand, address $2030 would contain 00, because $30 (its offset into the table) is an ASCII zero, so $2030 gets its binary equivalent: $00, a binary zero. Similarly, since $31 is an ASCII '1,' address $2031 would contain a binary '1:' $01. $2032 would contain a $02; $2033 would contain a $03, and so on up to $2039, which would contain a $09.

Addresses $203A thru $2040 would each contain $FF, because none of the ASCII codes from $3A thru $40 represent any of the characters 0 thru 9 or A thru F. On the other hand, address $2041 would contain a $0A, because $41 is an ASCII 'A' and $0A is its binary equivalent: a binary 'A.' By the same reasoning, $2042 would contain $0B; $2043 would contain $0C, and so on up to $2046, which would contain $0C, and so on up to $2046, which would contain $0F. Addresses $2047 thru $20FF would contain $FFs because none of the values $47 thru $FF is an ASCII 0 thru 9 or A thru F.

To use such a table, BINARY need only be a very simple routine:

```
BINARY    TAY              Use ASCII character as an index.
          LDA BINTAB,Y     Look up entry in BINary TABle.
          RTS              Return with it.
```

This is a typical example of a fast and simple table lookup code. But it requires a 256-byte table. Perhaps slightly more elaborate code can get by with a smaller table, or do away altogether with the need for a table. Such code must calculate, rather than look up, its answers. Let's look closely at the characters we must convert.

Legal inputs will be in the range $30 thru $39 or the range $41 thru $46. An input in the range $30 thru $39 is an ASCII 0 thru 9, and subtracting $30 from such an input will convert it to the corresponding binary value. An input in the range $41 thru $46 is an ASCII A thru F, so subtracting $37 will convert it to its corresponding binary value. For example, $41 (an ASCII 'A') minus $36 equals $0A (a binary 'A'). Any value not in either of these ranges is illegal and should cause BINARY to return a $FF.

Given these input/output relationships, BINARY need only determine whether the character in the accumulator lies in either legal range, and if so perform the appropriate subtraction, or, if the accumulator is not in a legal range, then return a $FF.

Here's some code for BINARY which makes these judgments, thus eliminating the need for a table:

```
BINARY  SEC             Prepare to subtract.
        SBC #$30        Subtract $30 from character.
        BCC BAD         If character was originally less than $30,
                        it was bad, so return $FF.
        CMP #$0A        Was character in the range $30 thru
                        $39?
        BCC GOOD        If so, it was a good input, and we've
                        already converted it to binary by sub-
                        tracting $30, so we'll return now with
                        the character's binary equivalent in the
                        accumulator.
        SBC #7          Subtract 7.
        CMP #$10        Was character originally in the range
                        $41 thru $46?
        BCS BAD         If so, it was a bad input.
        SEC
        CMP #$0A
        BCS GOOD

BAD     LDA #$FF        Indicate a bad input by returning
        RTS             minus, with A holding $FF.
GOOD    LDX #0          Indicate a good input by returning
        RTS             plus, with A holding the character's
                        binary equivalent.
```

Visible Monitor Utilities

The Visible Monitor makes the following subroutines available to external callers:

BINARY	Determine whether accumulator holds the ASCII representation for a hexadecimal digit. If so, return binary representation for that digit. If not, return an error code ($FF).
CALLSL	Call the currently selected address as a subroutine.
DEC.SL	Select previous address, by decrementing SELECT pointer.
GETKEY	Get a character from the keyboard by calling machine's read-only memory routine indirectly.
GET.SL	Get byte at currently selected address.
GO	Load registers from displayed images and call displayed address. Upon return, restore register images from registers.
INC.SL	Select next byte (increment SELECT pointer).
PUT.SL	Store accumulator at currently selected address.
VISMON	Let user give the Visible Monitor commands until user presses 'Q' to quit.

Figure 6.6 illustrates the hierarchy of the various routines of the Visible Monitor, some of which are detailed in later chapters.

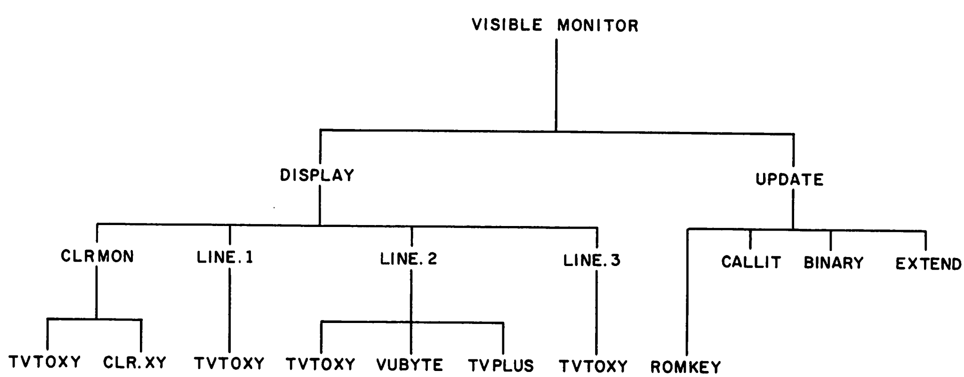

Figure 6.6: *A hierarchy of the routines of the Visible Monitor.*

Using the Visible Monitor

Chapter 13 shows you how to enter the object code for the Visible Monitor into your computer. To run the Visible Monitor with all the features described thus far, you must use the BASIC OBJECT CODE LOADER (described in Chapter 13) to load the object code represented by the following appendices:

	Appendix	Contains
	E1	Screen utilities
	E2	The Visible Monitor (Top Level and Display Subroutines)
	E3	The Visible Monitor (Update Subroutine)
PLUS:	E12	System Data block for the VIC-20
OR:	E13	System Data Block for the Commodore 64.

Entering the object code contained in the above appendices will give you the full functionality of the Visible Monitor. But that's just the beginning. There are many functions we can add to the Visible Monitor, to make it even more useful. We'll add those functions in the following chapters.

Chapter 7:

Print Utilities

The Visible Monitor is a useful tool for examining and modifying memory, but at the moment it's mute: it can't "talk" to you except through the limited device of the fields in its display. You can use the Visible Monitor's character entry feature to place ASCII characters directly into screen memory, thus putting messages on the screen manually. However, as yet we have no subroutines to direct a complete message, report, or other string of characters to the screen, to a printer, or to any other output device.

Most programs require some means of directing messages to the screen, thus providing the user with the basis for informed interaction, or to a printer, thus providing a record of that interaction. This chapter presents a set of print utilities to perform these functions.

Fortunately, there are subroutines in your computer's operating system to perform character output. The Commodore 64 and VIC-20 computers each feature a routine to print a character on the screen, thus simulating a TVT (*T*ele*V*ision *T*ypewriter), and they each feature another routine to send a character to the device connected to the serial output port: usually a printer. I don't plan to reinvent those wheels in this chapter. Rather, the chapter's software will funnel all character output through code that calls the appropriate subroutine in your computer's operating system. And since we're going to have code that calls the two standard character output routines, why not provide a hook to a user-written character output routine, as well? Such a feature will make it trivial for you to direct any character output (eg: messages, hexdumps, disassembler listings, etc) to the screen and the printer, or to any special output device you may have on your system, provided that you've written a subroutine to drive that device.

Selecting Output Devices

It should be possible for any program to direct character output to the screen, and/or to the printer, and/or to the user-written subroutine. Therefore, we'll need subroutines to select and deselect (stop using) each of these devices and to select and deselect *all* of these devices. Let's call these routines TVT.ON, TVTOFF, PR.ON, PR.OFF, USR.ON, USR.OFF, ALL.ON, and ALLOFF. With these subroutines, a calling program can select or deselect output devices individually or globally.

The line of source code which will select the TVT as an output device follows:

```
JSR TVT.ON
```

This line will deselect the TVT:

```
JSR TVTOFF
```

That's a pretty straightforward calling sequence.

The select and deselect subroutines will operate on three flags: TVT, PRINTR, and USER. The TVT flag will indicate whether the screen is selected as an output device; the PRINTR flag will indicate whether the printer is selected as an output device; and the USER flag will indicate whether the user-provided subroutine is selected as an output device.

For convenience, we'll have a separate byte for each flag and define a flag as "off" when its value is zero, and "on" when its value is nonzero.

Using this definition of a flag, we can select a given device simply by storing a nonzero value in the flag for that device; we can deselect a device simply by storing a zero in the flag for that device.

The definitions for the flags and listings of the select and deselect subroutines follow:

Device Flags

	OFF = 0	When a device flag = zero, that device is not selected.
	ON = $FF	When a device flag = $FF, that device is selected.
TVT	.BYTE ON	This flag is zero if TVT is not selected; nonzero otherwise. Initially, the TVT is selected.

PRINTR	.BYTE OFF	This flag is zero if the PRINTR is not selected; nonzero otherwise. Initially, the printer is not selected.
USER	.BYTE OFF	This flag is zero if the user-provided output subroutine is not selected; nonzero otherwise. Initially, the user-provided function is deselected.

Select and Deselect Subroutines

TVT.ON	LDA #ON STA TVT RTS	Select TVT as an output device by setting the flag that indicates the "select" state of the TVT.
TVTOFF	LDA #OFF STA TVT RTS	Deselect TVT as an output device by clearing the flag that indicates the "select" state of the TVT.
PR.ON	LDA #ON STA PRINTR RTS	Select printer as an output device by setting the flag that indicates the "select" state of the printer.
PR.OFF	LDA #OFF STA PRINTR RTS	Deselect printer as an output device by clearing the flag that indicates the "select" state of the printer.
USR.ON	LDA #ON STA USER RTS	Select user-written subroutine as an output device by setting the flag that indicates the "select" state of the output routine provided by the user.
USROFF	LDA #OFF STA USER RTS	Deselect user-written subroutine as an output device by clearing the flag that indicates the "select" state of the output routine provided by the user.
ALL.ON	JSR TVT.ON JSR PR.ON JSR USR.ON RTS	Select all output devices by selecting each output device individually.
ALLOFF	JSR TVTOFF JSR PR.OFF JSR USROFF RTS	Deselect all output devices by deselecting each output device individually.

A General Character-Print Routine

Now that a calling routine can select or deselect any combination of output devices, we need a routine that will output a given character to all currently selected output devices. Let's call this routine PR.CHR, because it will *PR*int a *CH*a*R*acter.

All the software in this book that outputs characters will do so by calling PR.CHR; none of that software will call your system's character-output routines directly. That makes the software in this book much easier to maintain. If you ever replace your system's TVT output routine or its printer-output routine with one of your own, you won't have to change the rest of the software in this book. That software will continue to call PR.CHR. However, if many lines of code in many places called your system's character-output routines directly, then replacing a read-only memory output routine with one of your own would require you to change many operands in many places. Who needs to work that hard? Funneling all character output through one routine, PR.CHR, means we can improve our character output in the future without difficulty.

When it is called, PR.CHR will look at the TVT flag. If the TVT flag is set, it will call your system's TVT output routine. Then it will look at the PRINTR flag. If the PRINTR flag is set, it will call your system's routine that sends a character to the serial output port. Finally, it will look at the USER flag. If the USER flag is set, it will call the user-provided character-output routine. Having done all of this, PR.CHR can return. Figure 7.1 is a flowchart for PR.CHR.

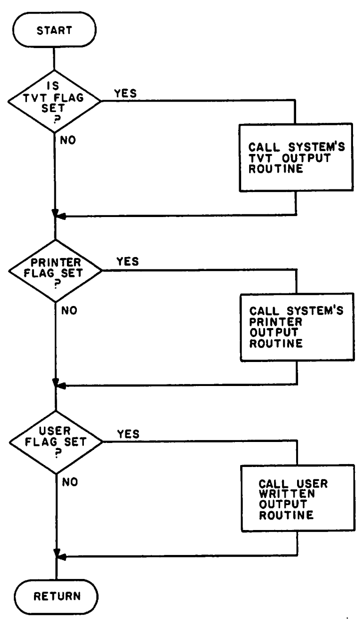

Figure 7.1: *To print a character to all currently selected output devices (PR.CHR, a general character-output routine).*

Output Vectors

If the character output routines are located at different addresses in different systems, how can PR.CHR know the addresses of the routines it must call? It can't. But it can call those subroutines indirectly, through pointers that you set.

You must set three pointers, or *output vectors*, so that they point to the character output routines in your system. A pointer called ROMTVT must point to your system's TVT output routine; a pointer called ROMPRT must point to your system's routine that sends a character to the serial output port; and a pointer called USROUT must point to your own, user-written, character-output routine. (If you have not written a special character-output subroutine, USROUT should point to a dummy routine which is nothing but an RTS instruction.) Then, if you ever relocate your TVT output routine, your printer-output routine, or your user-written output routine, you'll only have to change one output vector: ROMTVT, ROMPRT, or USROUT. Everything else in this book can remain the same.

ROMTVT, ROMPRT, and USROUT need not be located anywhere near PR.CHR. That means we can keep all the pointers and data specific to your system in one place. We can store the output vectors with the screen parameters, in a single block of memory called SYSTEM DATA. See Appendix B1 or B2 for your computer.

The source code of the PR.CHR routine follows:

PR.CHR

```
PR.CHR      STA CHAR       Save the character.
            BEQ EXIT       If it's a null, return without printing it.
            LDA TVT        Is TVT selected?
            BEQ IF.PR      If not, test next device.
            LDA CHAR       If so, send character indirectly to
            JSR SEND.1     system's TVT output routine.
IF.PR       LDA PRINTR     Is printer selected?
            BEQ IF.USR     If not, test next device.
            LDA CHAR       If so, send character indirectly
            JSR SEND.2     to system's printer driver.
IF.USR      LDA USER       Is user-written output subroutine
                           selected?
            BEQ EXIT       If not, test next device.
            LDA CHAR       If so, send character indirectly
            JSR SEND.3     to user-written output subroutine.
EXIT        RTS            Return to caller.
CHAR        .BYTE 0        This byte holds the last character passed
                           to PR.CHR.
```

Vectored Subroutine Calls

```
SEND.1      JMP (ROMTVT)
SEND.2      JMP (ROMPRT)
SEND.3      JMP (USROUT)
```

Specialized Character-Output Routines

Given PR.CHR, a general character-output routine, we can write specific character-output routines to perform several commonly required functions. For example, it's often necessary for a program to print a carriage return and a line feed, thus causing a new line, or to print a space, or to print a byte in hexadecimal format. Let's develop several dedicated subroutines to perform these functions. Since each of these subroutines will call PR.CHR, their output will be directed to all currently selected output devices.

Here are source listings for a few such subroutines: CR.LF, SPACE, and PR.BYT:

PRINT A CARRIAGE RETURN-LINE FEED

```
            CR = $0D        ASCII carriage return character.
            LF = $0A        ASCII line feed character.

CR.LF       LDA #CR         Send a carriage return and a
            JSR PR.CHR      line feed to the currently selected
            LDA #LF         device(s).
            JSR PR.CHR
            RTS             Return.
```

PRINT A SPACE

```
SPACE       LDA #$20        Load accumulator with ASCII space.
            JSR PR.CHR      Print it to all currently selected output
                            devices.
            RTS             Return.
```

PRINT BYTE

```
PR.BYT      PHA             Save byte.
            LSR A           Determine ASCII for the 4 MSB (most-
```

```
                  significant bits) in the
LSR A             byte:
LSR A
LSR A
JSR ASCII
JSR PR.CHR        Print that ASCII character to the current
                  device(s).
PLA               Determine ASCII for the 4 LSB (least-
                  significant bits) in the
JSR ASCII         byte that was passed to this subroutine.
JSR PR. CHR       Print that ASCII character to the current
                  device(s).
RTS               Return to caller.
```

Repetitive Character Output

Since some calling programs might need to output more than one space, a new line, or other character, why not have a few print utilities to perform such repetitive character outputs? In each case, the calling program need only load the X register with the desired repeat count. Then it would call SPACES to print X spaces, CR.LFS to print X new lines, or CHARS to print the character in the accumulator X times. Calling any of these routines with zero in the X register will cause no characters to be printed. To output seven spaces, a calling program would only have to include the following two lines of code:

```
LDX #7
JSR SPACES
```

To output four blank lines, a program would require these two lines of code:

```
LDX #4
JSR CR.LFS
```

To output ten asterisks, a program would need these three lines of code:

```
LDA #'*
LDX #10
JSR CHARS
```

In order to support these calling sequences, we'll need three small subroutines, SPACES, CR.LFS, and CHARS:

Print X Spaces; Print X Characters

```
SPACES    LDA #$20       Load accumulator with ASCII space.
CHARS     STX REPEAT     Initialize the repeat counter.
RPLOOP    PHA            Save character to be repeated.
          LDX REPEAT     Has repeat counter timed out yet?
          BEQ RPTEND     If so, exit. If not,
          DEC REPEAT     decrement repeat counter.
          JSR PR.CHR     Print character to all currently selected
                         output devices.
          PLA
          CLC            Loop back to repeat
          BCC RPLOOP     character, if necessary.
RPTEND    PLA            Clean up stack.
          RTS            Return to caller.
```

Print X New Lines

```
CR.LFS    STX REPEAT     Initialize repeat counter.
CRLOOP    LDX REPEAT     Exit if repeat counter has timed out.
          BEQ END.CR
          DEC REPEAT     Decrement repeat counter.
          JSR CR.LF      Print a carriage return and line feed.
          CLC            Loop back to see if done yet.
          RCC CRLOOP
END.CR    RTS            If done, return to caller.
REPEAT    .BYTE          This byte is used as a repeat counter by
                         SPACES, CHARS, and CR.LFS.
```

Print a Message

Some calling programs might need to output messages stored at arbitrary places in memory. So let's develop a subroutine, called PR.MSG, to perform this function. PR.MSG will print a message to all currently selected output devices. It must get characters from the message in a sequential manner and pass each character to PR.CHR, thus printing it on all currently selected output devices.

But how can PR.MSG know where the message starts and ends?

We could require that the message be placed in a known location, but then

PR.MSG would lose usefulness as it loses generality. We could require that a pointer in a known location be initialized so that it points to the start of the message. But that would still tie up the fixed 2 bytes occupied by that pointer. Or we could have a register specify the location of a pointer that actually points to the start of the message. Presumably a calling program can find some convenient 2 bytes in the zero page to use as a pointer, even if it must save them before it sets them. The calling program can set this zero-page pointer so that it points to the beginning of the message, and then set the X register so that it points to that zero-page pointer. Having done so, the calling program may call PR.MSG. Using the indexed indirect addressing mode, PR.MSG can then get characters from the message.

When PR.MSG has printed the entire message, it will return to its caller.

How will PR.MSG know when it has reached the end of the message? We can mark the end of each message with a special character: call it ETX, for *E*nd of *TeX*t. And for reasons which will become clear in Chapter 10, *A Disassembler*, we'll also start each message with another special character: TEX, for *TEX*t follows.

If we can develop PR.MSG to work from these inputs, then it won't be hard for a calling program to print any particular message in memory. Let's look at the required calling sequence.

A message, starting with a TEX and ending with an ETX, begins at some address. We'll call the high byte of that address MSG.HI and the low bye of that address MSG.LO. Thus, if the message starts at address $13A9, MSG.HI = $13 and MSG.LO = $A9.

MSGPTR is some zero-page pointer. It may be anywhere in the zero page. If the calling program does not have to preserve MSGPTR, it can print the message to the screen with the following code:

```
JSR TVT.ON        Select TVT as an output device. (Any other currently
                  selected output device will echo the screen output.)
LDA #MSG.LO       Set MSGPTR
STA MSGPTR        so it points
LDA #MSG.HI       to the start
STA MSGPTR+1      of the message.
LDX #MSGPTR       Set X register so it points to MSGPTR.
JSR PR.MSG        Print the message to all currently selected output
                  devices.
```

If the calling program must preserve MSGPTR, it will have to save MSGPTR and MSGPTR+1 before executing the above lines of code and restore MSGPTR and MSGPTR+1 after executing the above lines of code.

That looks like a reasonably convenient calling sequence. So now let's turn our attention to PR.MSG itself and develop it so it meets the demands of its callers.

Print a Message

```
PR.MSG      STX TEMP.X      Save X register, which specifies message
                            pointer.
            LDA 1,X         Save message pointer.
            PHA
            LDA 0,X
            PHA
LOOP        LDX TEMP.X      Restore original value of X, so it points
                            to message pointer.
            LDA (0,X)       Get next character from message.
            CMP #ETX        Is it the end of message indicator?
            BEQ MSGEND      If so, handle the end of the message...
            INC 0,X         If not, increment the message pointer
            BNE NEXT        so it points to the next
            INC 1,X         character in the message.
NEXT        JSR PR.CHR      Send the character to all currently
                            selected output devices.
            CLC             Get next character
            BCC LOOP        from message.
MSGEND      PLA             Restore message pointer.
            STA 0,X
            PLA
            STA 1,X
            RTS             Return to caller, with MSGPTR pre-
                            served.
TEMP.X      .BYTE 0         This data cell is used to preserve the ini-
                            tial value of X.
```

Print the Following Text

Even more convenient than PR.MSG would be a routine that doesn't require the caller to set any pointer or register in order to indicate the location of a message. But if no pointer or register indicates the start of the message, how can any subroutine know where the message starts?

It can look on the stack.

Why not have a subroutine, called Print-the-Following, which prints the message that follows the call to Print-the-Following. Since Print-the-Following is longer than six characters, let's shorten its name to "PRINT:", letting the colon in "PRINT:" suggest the phrase "the following." A calling program might then print "HELLO" with the following lines of code:

```
JSR TVT.ON          Select TVT as an output device. (Other currently
                    selected output devices will echo the screen output.)
JSR PRINT:
.BYTE TEX
.BYTE "HELLO"
.BYTE ETX
(6502 code follows the ETX)
.
.
.
```

Whenever the 6502 calls a subroutine, it pushes the address of the subroutine's caller onto the stack. This enables control to return to the caller when the subroutine ends with an RTS, because the 6502 knows it can find its return address on the stack. The subroutine PRINT: can take advantage of this fact by pulling its own return address off the stack, and using it as a pointer to the message that should be printed. When it reaches the end of the message, it can place a new return address on the stack, an address that points to the end of the message. Then PRINT: can execute an RTS. Control will then pass to the 6502 code immediately following the ETX at the end of the message. The source code for PRINT: follows:

```
PRINT:      PLA             Pull return address from
            TAX             stack and save it in
            PLA             registers X and Y.
            TAY
            JSR PUSHSL      Save the select pointer, because we're
                            going to use it as a text pointer.
            STX SELECT      Set SELECT = return address.
            STY SELECT+1
            JSR INC.SL      Increment SELECT pointer so it points
                            to TEX character.
LOOP        JSR INC.SL      Increment select pointer so it points to
                            the next character in the message.
            JSR GET.SL      Get character.
            CMP #ETX        Is it end of message indicator?
            BEQ ENDIT       If so, adjust return address and return.
            JSR PR.CHR      If not, print the character to all current-
                            ly selected devices.
            CLC             Then loop to get
            BCC LOOP        next character...
ENDIT       LDX SELECT
            LDY SELECT+1
```

```
         JSR POP.SL        Restore select pointer to its original
                           value.
         TYA               Push address
         PHA               of ETX
         TXA               onto the stack.
         PHA
         RTS               Return (to byte immediately following
                           ETX).
```

Saving and Restoring the SELECT Pointer

Now that a number of subroutines are accessing the contents of memory with the SELECT utilities (GET.SL, PUT.SL, INC.SL and DEC.SL) we should provide yet another pair of SELECT utilities to enable the subroutines to save and restore the SELECT pointer. With such save and restore functions, any subroutine can use the SELECT pointer to access memory, without interfering with the use of the SELECT pointer by other subroutines. PUSHSL will push the SELECT pointer onto the stack and POP.SL will pop the SELECT pointer off the stack. PUSHSL and POP.SL will each preserve X,Y, and the zero page.

Save Select Pointer
(Preserving X,Y, and the Zero Page)

```
PUSHSL   PLA               Pull return address from stack and
         STA RETURN        store it temporarily in RETURN.
         PLA
         STA RETURN+1
         LDA SELECT+1      Push select pointer onto stack.
         PHA
         LDA SELECT
         PHA
         LDA RETURN+1      Push return address back onto stack.
         PHA
         LDA RETURN
         PHA
         RTS               Return to caller. (Caller will find select
                           pointer on top of the stack.)
```

Restore Select Pointer (Preserving X,Y, and the Zero Page)

```
POP.SL      PLA             Save return address temporarily.
            STA RETURN
            PLA
            STA RETURN+1
            PLA             Restore select pointer from stack.
            STA SELECT
            PLA
            STA SELECT+1
            LDA RETURN+1    Place return address back on stack.
            PHA
            LDA RETURN
            PHA
            RTS             Return to caller.
RETURN      .WORD 0         This pointer is used by PUSHSL and
                            POP.SL to preserve their return ad-
                            dresses.
```

Conclusion

With the print utilities presented in this chapter, it should be easy to write the character-output portions of many programs, making it possible for calling programs to select any combination of output devices and to send individual characters, bytes, or complete messages to those devices. The calling programs will be completely insulated from the particular data representations used by the print utilities. The calling programs do not need to know the nature or location of the output-device flags or the addresses of the output vectors; they need only know the addresses of the print utilities.

Similarly, although the print utilities use subroutines that operate on the SELECT pointer, the print utilities themselves never access the SELECT pointer directly. They are completely insulated from the nature and location of the SELECT pointer. As long as they know the addresses of the SELECT utilities, the print utilities can get the currently selected byte, select the next or the previous byte, save the SELECT pointer onto the stack, and restore the SELECT pointer from the stack. If at some point we should implement a different representation of "the currently selected byte," we need only change the SELECT utilities; the print utilities, and all other programs which use the SELECT utilities need never change.

Insulating blocks of code from the internal representation of data in other blocks of code makes all the code much easier to maintain.The following print utilities are available to external callers:

CHARS	Send the character in the accumulator "X" times to all currently selected output devices.
CR.LF	Cause a new line on all currently selected devices.
CR.LFS	Cause "X" new lines on all currently selected devices.
PR.BYT	Print the byte in the accumulator, in hexadecimal representation.
PR.CHR	Print the character in the accumulator on all currently selected devices.
PR.MSG	Print the message pointed to by a zero-page pointer specified by X.
PRINT:	Print the message following the call to "PRINT:".
SPACE	Send a space to all currently selected output devices.
SPACES	Send "X" spaces to all currently selected output devices.

Exercises

1) Write a printer test program, which sends every possible character from $00 to $FF to the printer.

2) Rewrite the printer test program so that it prints just one character per line.

Chapter 8:

Two Hexdump Tools

The Visible Monitor allows you to examine memory, but only 1 byte at a time. You'll quickly feel the need for a software tool that will display or print out the contents of a whole block of memory. This is especially useful if you wish to debug a program. You can't debug a program if you're not sure what's in it. A hexdump tool will show you what you've actually entered into the computer, by displaying the contents of memory in hexadecimal form.

I've developed two kinds of hexdump programs, each for a different type of output device. When I'm working at the keyboard, I want a hexdump routine that dumps from memory to the *screen*, a line or a group of lines at a time. But for documentation and for program development or debugging away from the keyboard, I want a hexdump routine that dumps to a *printer*.

Most of the code required to dump from memory will be the same, whether we direct output to the screen or to the printer. However, there are enough differences between the two output devices that it is convenient to have two hexdump programs, one for the screen and one for the printer. Let's call them TVDUMP and PRDUMP.

TVDUMP

TVDUMP should be very responsive: when you are using the Visible Monitor, a single keystroke should cause one or more lines to be dumped to the screen. But how can TVDUMP know what lines you want to dump? Since the Visible Monitor allows you to select any address by rolling hexadecimal characters into the address field or by stepping forward and backward through memory, we might as well have

TVDUMP dump memory beginning with the currently selected address.

Since we're basing TVDUMP on the Visible Monitor's currently selected address, we can use some of the Visible Monitor's subroutines to operate on that address. GET.SL will get the currently selected byte, and INC.SL will increment the SELECT pointer, thereby selecting the next byte. The print utilities TVT.ON and PR.BYT will let us select the screen as an output device and print the accumulator in hexadecimal representation.

We ought to have TVDUMP provide two dumps that will be easily readable, even on the narrow confines of a twenty-two or forty-column display. That means we can't display a full hexadecimal line (16 bytes) on one screen line if we want to have a space between each byte. We can provide hexdumps that split each hexadecimal line into two or four screen lines. See outputs A and B in figure 8.1.

```
                Output A:

0200    HH  HH  HH  HH  HH  HH  HH  HH  HH
0208    HH  HH  HH  HH  HH  HH  HH  HH  HH

0210    HH  HH  HH  HH  HH  HH  HH  HH  HH
0218    HH  HH  HH  HH  HH  HH  HH  HH  HH

-------------------------------32 columns-------------------------------


            Output B:

0200  HH  HH  HH  HH
0204  HH  HH  HH  HH
0208  HH  HH  HH  HH
020C  HH  HH  HH  HH

0210  HH  HH  HH  HH
0214  HH  HH  HH  HH
0218  HH  HH  HH  HH
021C  HH  HH  HH  HH

-----------17 columns-----------
```

Figure 8.1: *Two TVDUMP formats.*

One way to provide such a hexdump is shown by the flowchart in figure 8.2. Using this flowchart as a guide, let's develop source code to perform the TVDUMP function:

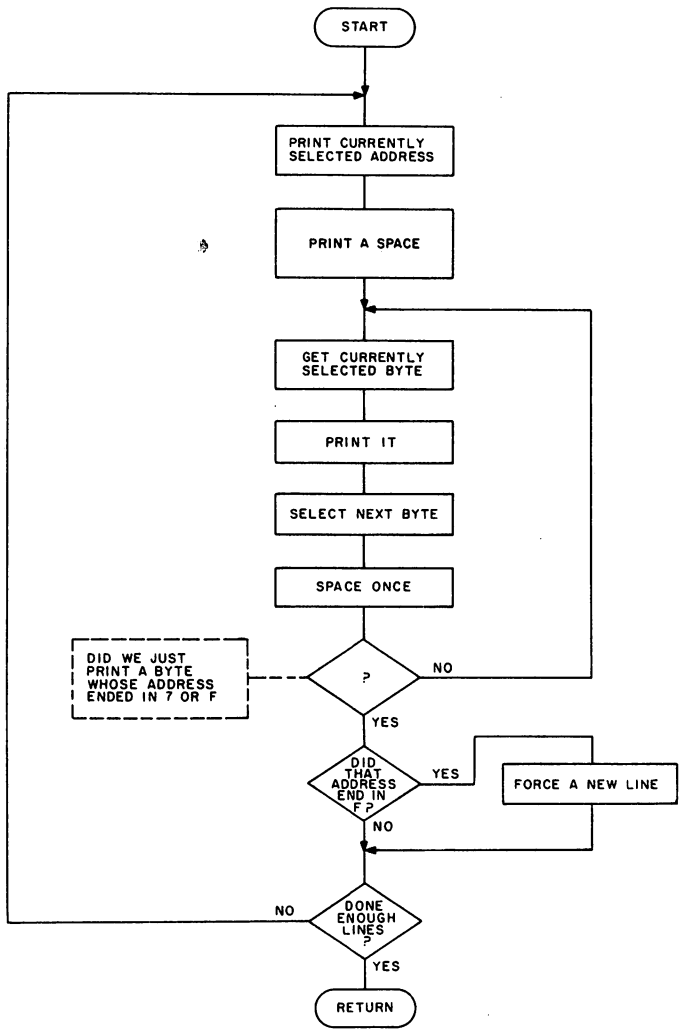

Figure 8.2: *Flowchart of the screen Hexdump Program.*

CONSTANTS

CR = $0D	Carriage return.
LF = $0A	Line feed.

REQUIRED SUBROUTINES

GET.SL	Get currently selected byte.
INC.SL	Increment the pointer that specifies the currently selected byte.
PR.BYT	Print the accumulator to currently selected devices, in hexadecimal representation.
SELECT	Pointer to currently selected address.

VARIABLES

COUNTR	.BYTE 0	This byte counts the number of lines dumped by TVDUMP.
MASK	.BYTE 7	For output A (suitable for C-64). Use ".BYTE 3" for output B (suitable for VIC-20).

TVDUMP

TVDUMP	JSR TVT.ON	Select TVT as an output device. (Other devices will echo the dump.)
	LDA #4	Set COUNTR to the number of lines
	STA COUNTR	to be dumped by TVDUMP.
	LDA SELECT	Set SELECT to beginning
	AND #$F0	of a hex line (16 bytes)
	STA SELECT	by zeroing 4 LSB in SELECT.
DUMPLN	JSR PR.ADR	Print the selected address.
	JSR SPACE	Print a space.
DMPBYT	JSR SPACE	Print a space.
	JSR DUMPSL	Dump currently selected byte.
	JSR INC.SL	Select next address by incrementing select pointer.

```
                LDA SELECT      Is it the beginning of a new
                AND MASK        screen line?
                BNE DMPBYT      If not, dump next byte...
                JSR CR.LF       If so, advance to a new line on the
                                screen.
                LDA SELECT      Does this address mark the beginning of
                                a new hexadecimal line?
                AND #$0F        (4 LSB of SELECT = 0?)
                BNE IFDONE
                JSR CR.LF       If so, skip a line on the screen.
IFDONE          DEC COUNTR      Dumped last line yet?
                BNE DUMPLN      If not, dump next line.
                JSR TVTOFF      Deselect TVT as an output device.
                RTS             Return to caller.
```

DUMP CURRENTLY SELECTED BYTE

This subroutine gets the currently selected byte (the byte pointed to by SELECT) and prints it in hexadecimal format on all selected devices.

```
DUMPSL          JSR GET.SL      Get currently selected byte.
                JSR PR.BYT      Print it in hexadecimal format.
                RTS             Return to caller.
```

PRINT ADDRESS

This subroutine prints, on all selected devices, the currently selected address (ie: the value of the SELECT pointer).

```
PR.ADR          LDA SELECT+1    Get the high byte of SELECT...
                JSR PR.BYT      ...and print it in hexadecimal format.
                LDA SELECT      Get the low byte of SELECT...
                JSR PR.BYT      ...and print it in hexadecimal format.
                RTS             Then return to caller.
```

PRDUMP

With the subroutine presented thus far in this chapter, we can dump to the screen just by calling TVDUMP. But what if we want to *print* a hexdump? Is a hexdump program that prints any different from one that dumps to the screen? Can we simply select the printer instead of the TVT and leave the rest of the code the same?

We could. But then we wouldn't be taking full advantage of the printer. TVDUMP produces an output that is easily read within the twenty-two or forty columns of a video display. Most printers can output sixty-four columns or more. We should take advantage of the extra width offered by a printer.

We should also recognize the difference in responsiveness between a screen and a hard-copy device. When I'm using a screen-based hexdump, I don't mind hitting a single key every time I want some lines dumped to the screen. But with a printing hexdump, I don't want to strike a key repeatedly to continue the dump. I don't mind striking a number of keys at the beginning in order to specify the memory to be dumped, but once I've done that I don't want to be bothered again. I want to set it and forget it.

When called, a printing hexdump program should announce itself by clearing the screen and displaying an appropriate title (eg: "PRINTING HEXDUMP"). Then it should ask you to specify the starting address and the ending address of the memory to be dumped.

Once it knows what you want to dump, PRDUMP should print a hexdump of the specified block of memory. For your convenience, PRDUMP should tell you what block of memory it will dump; then it should provide a header for each column of data and indicate the starting address of each line of data. (See the "D" appendices.)

Using the flowchart of figure 8.3 as a guide, we can write source code for the top level of the PRINTING HEXDUMP:

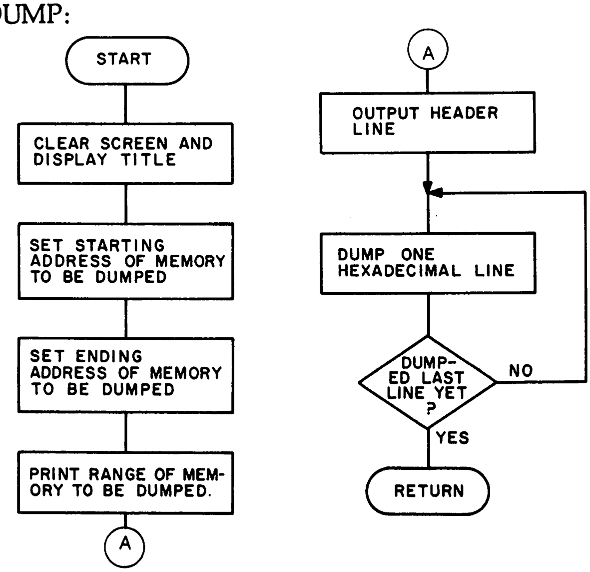

Figure 8.3: *To print a Hexdump.*

```
PRDUMP   JSR TITLE        Display the title.
         JSR SETADS       Let user set start address and end ad-
                          dress of memory to be dumped.
                          (SETADS returns with SELECT=EA,
                          the end address.)
         JSR GOTOSA       Set SELECT=SA, the starting address.
         JSR PR.ON        Select printer as a output device. (Other
                          selected devices will echo the dump.)
         JSR HEADER       Output hexdump header.
HXLOOP   JSR PRLINE       Dump one line. (PRLINE returns minus
                          if it dumped through ending address;
                          otherwise it returns PLUS.)
         BPL HXLOOP       Done yet? If not, dump next line.
         JSR CR.LF        If so, go to a new line.
         JSR PR.OFF       Deselect printer.
         RTS              Return to caller. Specified memory has
                          been dumped.
TITLE    JSR CLR.TV       Clear the screen.
         JSR TVT.ON       Select screen as an output device.
         JSR PRINT:       Display "Printing Hexdump" on all
                          selected output devices.
         .BYTE TEX        Text string must start with a TEX
                          character...
         .BYTE CR,'PRINTING '
         .BYTE 'HEXDUMP ',CR
         .BYTE LF,LF,
         .BYTE ETX        ...and end with an ETX character.
         RTS              Return to caller.
```

Get Starting, Ending Address

The printing hexdump program must secure from the user the starting address and the ending address of the memory to be dumped. The subroutine, SETADS, will perform these functions. It will place an appropriate prompt on the screen ("Set Starting Address" or "Set Ending Address") and then allow the user to specify an address.

Putting a prompt on the screen is easy: just select the TVT by calling TVT.ON, call "PRINT:" and follow this call with a TEX (start of text) character, the text of the prompt, and then an ETX (end of text) character. How can we allow the user to specify an address? We could make a subroutine, called GETADR, which gets an address by enabling the user to set some pointer. That sounds mighty familiar — that's what the Visible Monitor does. Conveniently, the Visible Monitor is a subroutine, which returns to its caller when the user presses Q for Quit. Therefore, after putting the appropriate prompt on the screen, SETADS will call the Visible Monitor. When the Visible Monitor returns, the SELECT pointer will specify the requested address.

SET STARTING ADDRESS, ENDING ADDRESS

```
SETADS    JSR TVT.ON        Select TVT as an output device. All
                            other selected output devices will echo
                            the screen output.
          JSR PRINT:        Put prompt on the screen:
          .BYTE TEX
          .BYTE CR,LF,LF
          .BYTE             'SET STARTING ADDRESS '
          .BYTE             'AND PRESS "Q".'
          .BYTE ETX
          JSR VISMON        Call the Visible Monitor, so user can
                            specify a given address.
          JSR SAHERE        Set starting address equal to address set
                            by the user.
SET.EA    JSR PRINT:        Put prompt on the screen:
          .BYTE TEX
          .BYTE CR,LF,LF
          .BYTE             'SET ENDING ADDRESS '
          .BYTE             'AND PRESS "Q".'
          .BYTE ETX
          JSR VISMON        Call the Visible Monitor, so user can
                            specify a given address.
          SEC               If user tried to set an
          LDA SELECT+1      ending address less than
          CMP SA+1          the starting address,
          BCC TOOLOW        make user do it over.
          BNE EAHERE        If SELECT is greater than SA, set
                            EA=SELECT. That will make EA
                            greater than SA.
          LDA SELECT
          CMP SA
          BCC TOOLOW
EAHERE    LDA SELECT+1      Set EA=SELECT...
          STA EA+1
          LDA SELECT
          STA EA
          RTS               ... and return.
SAHERE    LDA SELECT+1      Set SA=SELECT...
          STA SA+1
```

```
                LDA SELECT
                STA SA
                RTS               ...and return.
TOOLOW          JSR PRINT:        Since user set ending address
                .BYTE STX,        too low, print error message:
                .BYTE CR,LF,LR
                .BYTE             'ERROR! '
                .BYTE             'END ADDRESS LESS '
                .BYTE             'THAN START ADDRESS, '
                .BYTE             'WHICH IS '
                .BYTE ETX
                JSR PR.SA         Print starting address. ...and let the user
                                  set
                JMP SET.EA        the ending address again.
SA              .WORD 0           Pointer to starting address of memory to
                                  be dumped.
EA              .WORD $FFFF       Pointer to ending address of memory to
                                  be dumped.
```

Now that the user can set the starting address and the ending address for a hexdump (or for any other program that must operate on a contiguous block of memory), we should have utilities that print out the starting address, the ending address, or the range of addresses selected by the user. If the user set $D000 as the starting address and $D333 as the ending address, we should be able to call one subroutine that prints "$D000," another that prints "$D333," and a third that prints "$D000 — $D333."

Let's call these subroutines PR.SA, to print the starting address; PR.EA, to print the ending address; and RANGE, to print the range of addresses.

Print Starting Address

The following subroutine prints the value of SA, the starting address, in hexadecimal format:

```
PR.SA           LDA #'$           Print a dollar sign to
                JSR PR.CHR        indicate hexadecimal.
                LDA SA+1          Print high byte of starting address.
                JSR PR.BYT
                LDA SA            Print low byte of starting address.
                JSR PR.BYT
                RTS               Return to caller.
```

Print Ending Address

The following subroutine prints the value of EA, the ending address, in hexadecimal format:

```
PR.EA       LDA #'$         Print a dollar sign to
            JSR PR.CHR      indicate hexadecimal.
            LDA EA+1        Print high byte of ending address.
            JSR PR.BYT
            LDA EA          Print low byte of ending address.
            JSR PR.BYT
            RTS             Return to caller.
```

Print Range of Addresses

```
RANGE       JSR PR.SA       Print starting address.
            LDA #'-         Print a hyphen.
            JSR PR.CHR
            JSR PR.EA       Print ending address.
            RTS             Return to caller.
```

HEADER

We want a routine to print an appropriate header for the hexdump. It should accomplish two tasks: identify the block it will dump, and print a hexadecimal digit at the top of every column of hexdump output. Thus, HEADER should produce the output shown between the following lines:

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

```

Notice the blank line following the line of hexadecimal characters. This will insure a blank line between the header and the dump itself, making for a more readable output. (See the hexdumps in the D series of appendices which were produced with PRDUMP.)

Here are a few lines of code to print the first line of the header:

```
JSR PRINT:
.BYTE TEX,CR,LF
.BYTE 'DUMPING '
.BYTE ETX
JSR RANGE
JSR CR.LF
```

What about the rest of the header? Since all we want to do is print the hexadecimal digits 0 thru $F, with appropriate spacing between them, the rest of HEADER can just be some code to count from 0 to $F, convert to ASCII, and print:

PRINT HEXADECIMAL DIGITS (Version 1)

```
          LDX #7          Print seven spaces.
          JSR SPACES
          LDA #0          Initialize column counter
          STA COLUMN      to zero.
HXLOOP    LDA COLUMN      Convert column counter to
          JSR ASCII       an ASCII character and
          JSR PR.CHR      print it.
          LDX #2          Space twice after the character.
          JSR SPACES
          INC COLUMN      Increment the column counter.
          LDA COLUMN      Loop if counter not greater
          AND #$F0        than $0F.
          BEQ HXLOOP
          LDX #2          Otherwise, skip two lines
          JSR CR.LFS      after the header.
          RTS             Then return.
COLUMN    .BYTE 0         This 1-byte variable is used to count
                          from 00 to $0F.
```

Version 1 of PRINT HEXADECIMAL DIGITS will work, and in only 49 bytes. But that's 49 bytes of code, which among other things must count and branch, and if for some reason one of those bytes is wrong, Version 1 of PRINT HEXADECIMAL DIGITS will probably go directly into outer space. But we could write PRINT

HEXADECIMAL DIGITS in a much more straightforward manner, which, though somewhat more costly in terms of memory required, will be more readable and less likely to run amuck.

PRINT HEXADECIMAL DIGITS need only call "PRINT:", and follow this call with a text string consisting of the desired hexadecimal digits.

PRINT HEXADECIMAL DIGITS (Version 2)

```
JSR PRINT:
.BYTE TEX
.BYTE '          0    1    2    3    4    5    6    7    '
.BYTE           '8    9    A    B    C    D    E    F'
.BYTE CR,LF,LF
.BYTE ETX
RTS
```

Version 2 of PRINT HEXADECIMAL DIGITS requires 60 bytes. But it's more readable than Version 1 of PRINT HEXADECIMAL DIGITS, and it can be modified much more easily: just change the text in the message it prints. You don't have to calculate branch addresses or test the terminal condition in a loop. This is just one example of a programming problem that may be solved in a computation-intensive or a data-intensive manner.

Where other factors are about equal, I prefer data-intensive subroutines, because they're more readable and easier to change. Even in this case, I'm willing to pay the extra 11 bytes for a version of PRINT HEXADECIMAL DIGITS that I don't have to read twice. Hence, PRINT HEXADECIMAL DIGITS Version 2, and not Version 1, will appear in the assembler listing of HEADER in Appendix C5.

PRLINE

Clearly, most of the work of PRDUMP will be performed by the subroutine PRLINE, which dumps one line of memory to the printer. It will stop when it has dumped 16 bytes (one hexadecimal line) or has dumped through the ending address specified by the user.

As we did for TVDUMP, let's use SELECT as a pointer to the first byte that must be dumped by PRLINE. When PRLINE is called, it must see if the currently selected byte (the byte pointed to by SELECT) is at the start of a hexadecimal line. A byte is at the beginning of a hexadecimal line if the 4 LSB (least-significant bits) of its address are zero. Thus, $4ED8 is not the start of a hexadecimal line, but $4ED0 *is*.

If the currently selected byte is not the beginning of a hexadecimal line, PRLINE should space over to the appropriate column for that byte. If the currently selected byte is at the beginning of a hexadecimal line, PRLINE should print the address of the currently selected byte and space twice.

Once it has spaced over to the proper column, PRLINE need only get the currently selected byte, print it in hexadecimal format, space once, and then do the same for the next byte, until it has dumped the entire line or has dumped the last byte requested by the user.

Figure 8.4 gives a flowchart for the following routine:

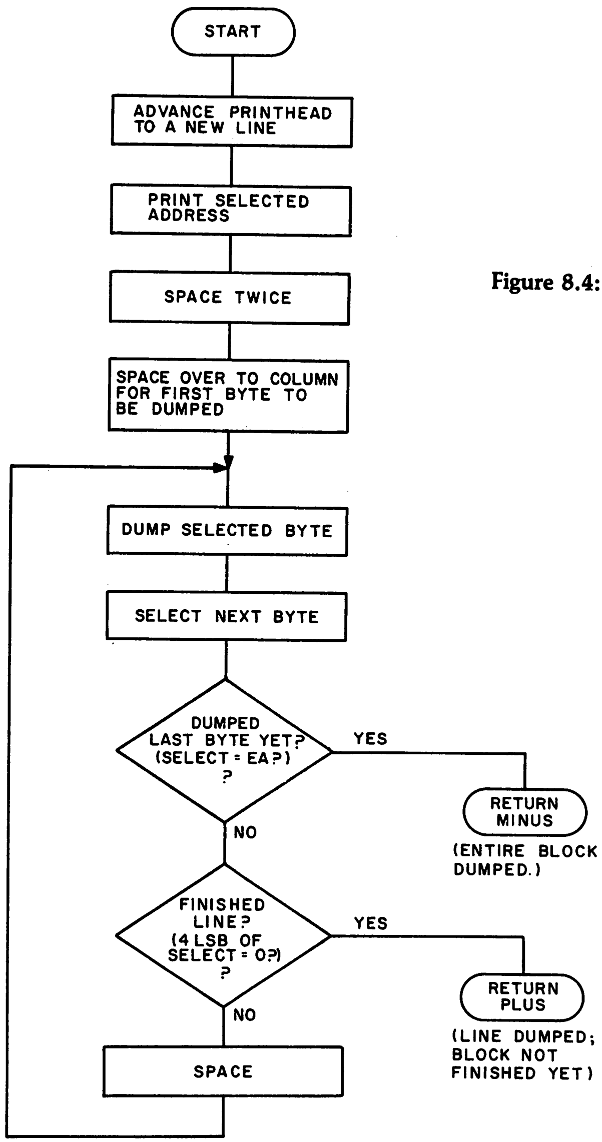

Figure 8.4: *Dump one line to the printer.*

PRLINE

```
PRLINE    JSR CR.LF       Advance printhead to a new line.
          LDA SELECT      Determine starting
          PHA             column
          AND #$0F        for this dump.
          STA COLUMN      Now COLUMN holds the number of the
                          column in which we will dump the first
                          byte.
          PLA             Set SELECT pointer to
          AND #$F0        beginning of a hexadecimal line.
          STA SELECT
          JSR PR.ADR      Print the selected address.
          LDX #3          Space three times — to the
          JSR SPACES      first column.
          LDA COLUMN      Do we dump from the first column?
          BEQ COL.OK      If so, we're at the correct column now.
LOOP      LDX #3          If not, space three
          JSR SPACES      times for each byte not
          JSR INC.SL      dumped.
          DEC COLUMN
          BNE LOOP
COL.OK    JSR DUMPSL      Dump the currently selected byte.
          JSR SPACE       Space once.
          JSR NEXTSL      Select the next byte in memory, unless
                          we've already dumped through the end
                          address.
          BMI EXIT        (MINUS means we've dumped through
                          the end address.)
NOT.EA    LDA SELECT      Dumped entire line?
          AND #$0F        (4 LSB of SELECT = 0?)
          CMP #0          If so, we've dumped the entire line. If
                          not,
          BNE COL.OK      select the next byte and dump it...
EXIT      RTS             PRLINE returns MINUS, with A=$FF,
                          if it dumped through ending address.
                          Otherwise it returns PLUS, with A=0.
```

Select Next Byte

NEXTSL tests to see if SELECT is less than the ending address. If so, it increments SELECT and returns PLUS (with zero in the accumulator). If not, it preserves SELECT and returns MINUS (with $FF in the accumulator).

NEXTSL

```
NEXTSL   SEC                Prepare to compare.
         LDA SELECT+1       Is high byte of SELECT less than
         CMP EA+1           high byte of end address (EA)?
         BCC SL.OK          If so, SELECT is less than EA, so it may
                            be incremented.
         BNE NO.INC         If SELECT is greater than EA, don't
                            increment SELECT.
                            SELECT is in the same page as EA,
         SEC                prepare to compare low bytes:
         LDA SELECT         Is low byte of SELECT less than
         CMP EA             low byte of EA?
         BCS NO.INC         If not, don't increment it.
SL.OK    JSR INC.SL         Since SELECT is less than EA, we may
                            increment it.
         LDA #0             Set "incremented" return code and
         RTS                return.
NO.INC   LDA #$FF           Set "not incremented" return code
         RTS                and return.
```

Go to Start of Block

GOTOSA sets SELECT = SA, thus selecting the first byte in the block defined by SA and EA:

```
GOTOSA   LDA SA             Set SELECT
         STA SELECT         equal to
         LDA SA+1           START ADDRESS
         STA SELECT+1       of block.
         RTS
```

Now the two hexdump tools are complete. You may invoke either tool directly from the Visible Monitor by displaying the start address of the given hexdump tool and pressing "G." This will work fine for PRDUMP: you'll get a chance to set the starting address and the ending address that you want to dump, and then you'll see the dump on both the printer and the screen. If you start TVDUMP with a "G" from the Visible Monitor, you'll only get a dump of TVDUMP itself. You won't be able to use TVDUMP to dump any other location in memory. Why? Because TVDUMP dumps from the displayed address, and to start any program with a "G" from the Visible Monitor, you must first display the starting address of that program. Probably you'd like to be able to use TVDUMP to dump other areas in memory. To do so, you must assign a Visible Monitor key (eg: "H") to the subroutine TVDUMP, so that the Visible Monitor will call TVDUMP whenever you press that key. See Chapter 12, *Extending the Visible Monitor.*

Chapter 9:

A Table-Driven Disassembler

With the Visible Monitor you can enter object code into your computer. With hexdump tools you can dump that object code to the screen or to a printer. However, you still can't be sure you've entered the instructions you intended to enter unless you refer back and forth from your hexdump to Appendix A4, *The 6502 Opcode List*. You must verify that every opcode you entered is for the instruction and the addressing mode that you had intended. You must count forward or backward in hexadecimal to make sure that the operands in your branch instructions are correct. If you entered one opcode or operand incorrectly, then even though your handwritten program may be correct, the version in your computer's memory will be wrong.

A disassembler (the opposite of an assembler) can make your life a lot easier by displaying or printing the mnemonics represented by the opcodes you entered into your computer, and by showing you the actual addresses and addressing modes represented by your operands. The disassembler can't know that address $FB has the label "TV.PTR," but it can let you know that a given instruction operates on address $FB.

A disassembled line includes the following fields:

Field Number	Field Description
1.	Mnemonic.
2.	Operand.
3.	Address of opcode.
4.	Opcode in hexadecimal.

5. First byte of operand (if present) in hexadecimal.
6. Second byte of operand (if present) in hexadecimal.

Here's a disassembled line, with each of the fields numbered:

1	*2*	*3*	*4*	*5*	*6*	(Field Numbers)
JSR	0400	08AC	20	00	04	(Disassembled Line)

As with hexdump tools, I find it convenient to have two disassemblers: one for the screen and one for the printer. The screen-oriented disassembler should direct a certain number of disassembled lines to the screen whenever it is called. On the other hand, the printing disassembler should get a starting address and an ending address from the user and print a continuous disassembly of that portion of memory. As before, when I direct output to a printer I want to set it and forget it.

Whether we disassemble to the screen or to a printer, we will disassemble one line at a time. How can a program disassemble a line? The same way a person does. You look at an opcode in memory and then consult a table such as Appendix A4 to determine the operation represented by that opcode. Each operation has two attributes, a mnemonic and an addressing mode. The procedure is simple. Write the mnemonic; then, from the addressing mode determine whether this opcode takes no operand, a 1-byte operand, or a 2-byte operand. If it takes an operand, look at the next byte or two in memory and then write the operand for the mnemonic.

Thus, if you wish to disassemble object code from some place in memory, and you find an \$8D at that location, you can determine from Appendix A6 that \$8D represents "store accumulator, absolute mode." Therefore, you'll write: "STA," which is the mnemonic for store the accumulator.

The absolute mode requires a 2-byte operand, so you'll look at the 2 bytes following the \$8D. If \$36 follows the \$8D and is itself followed by \$D0, then the disassembled line will look like this:

```
STA $D036
```

That's a lot easier to read than the original 3 bytes of object code:

```
8D 36 D0
```

DISASSEMBLY

JSR	0400	1E00	20	00	04
JSR	04A0	1E03	20	A0	04
LDA	(0021),Y	1E06	B1	21	
CLC		1E08	18		
BCC	1E00	1E09	90	F5	

HEXDUMP

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1E00	20	00	04	20	A0	04	B1	21	18	90	F5					

Figure 9.1: *Disassembly and hexdump of the same object code.*

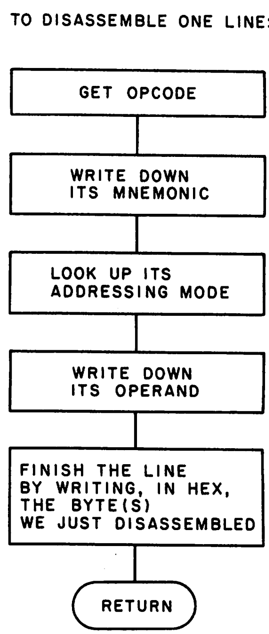

Figure 9.2: *Algorithm for disassembling one line of code.*

That looks pretty simple. We can use the SELECT pointer to indicate the current byte within memory, and we'll assume that lower-level subroutines exist or will exist to do the jobs required by DSLINE, which disassembles one line. With those assumptions, we can write source code for DSLINE:

DISASSEMBLE ONE LINE

DSLINE	JSR GET.SL	Get currently selected byte.
	PHA	Save it on stack.
	JSR MNEMON	Print the mnemonic represented by that opcode.
	JSR SPACE	Space once.
	PLA	Restore opcode to accumulator.
	JSR OPERND	Print the operand required by that opcode.
	JSR FINISH	Finish the line by printing fields 3 thru 6.
	JSR NEXTSL	Select next byte.
	RTS	Return to caller, with SELECT pointing at the last byte of the operand (or at the opcode, if it was a 1-byte instruction).

Print Mnemonic

We need a subroutine called MNEMON which prints the three-letter mnemonic for a given opcode. How can MNEMON do this? How do we do it? We look it up in a table such as Appendix A4. We could have a similar table in memory and then have MNEMON sequentially look up from the table the three characters comprising the desired mnemonic. That would require a 3-byte mnemonic for each of 256 possible opcodes: a 758-byte table. That's a lot of memory! Perhaps if we organize our data better we'll need less memory.

For example, why include the same mnemonic more than once in the table? Eight different opcodes use the mnemonic LDA; why should I use up 24 bytes to store "LDA" eight times? We could have a table of mnemonic names, which is nothing more than an alphabetical list of the three-letter mnemonics. There are only fifty-six different mnemonics; if we add one pseudo-mnemonic, "BAD," to mean that a given opcode is not valid, then we still have only fifty-seven mnemonics. The table of mnemonic names will therefore require only 171 bytes.

If you have a given opcode, how can you know which mnemonic in the table of mnemonic names corresponds to your opcode? A mnemonic *code* is some number that uniquely identifies a given mnemonic. Let's assume that we have a table of mnemonic codes which gives the mnemonic code for each possible opcode.

Now you can look up in the table of mnemonic codes the mnemonic code corresponding to a given opcode, and then use the mnemonic code as an index to the table of mnemonic *names*. The three sequential characters located in the table of mnemonic names will comprise the mnemonic for your original opcode.

This method requires not one but two tables. The two together, however, require considerably less memory than our first table did. The table of mnemonic codes will be 256-bytes long, since it must have an entry for every possible opcode, including invalid ones. The table of mnemonic names, on the other hand, will be only 171-bytes long, so the two tables together require only 427 bytes. That's 331 bytes or 43 percent less memory than our first table required.

Space saved in tables may not be worth it if large or complicated code is required as an index to those tables, but in this case the code is quite simple:

```
MNEMON      LDX #3            There are three letters in a mnemonic.
            STX LETTER        We'll keep track of the letters by count-
                              ing down to zero.
            TAX               Prepare to use the opcode as an index.
            LDA MCODES,X      Look up the mnemonic code for that op-
                              code. (MCODES is the table of
                              mnemonic codes.)
            TAX               Prepare to use that mnemonic code as
                              an index.
MNLOOP      LDA MNAMES,X      Get a mnemonic character. (MNAMES
                              is the list of mnemonic names.)
            STX TEMP.X        Save X register (since printing will
                              almost certainly change the X register).
            JSR PR.CHR        Print the character to all currently
                              selected devices.
            LDX TEMP.X        Restore X register to its previous value.
            INX               Adjust index for next letter.
            DEC LETTER        If three letters not yet printed,
            BNE MNLOOP        loop back to handle the next one.
            RTS               Otherwise, return to caller.
TEMP.X      .BYTE 0
LETTER      .BYTE 0
```

As you can see, MNEMON requires only 30 bytes of code in machine language: 2 bytes to hold variables and 427 bytes for the two tables (MNAMES and MCODES). The entire subroutine requires 459 bytes, but since most of those bytes are data in tables, comparatively little can go wrong with the program. If the wrong bytes are keyed into the table of mnemonic names, then the disassembler will print one or more incorrect characters in a mnemonic. But MNEMON won't crash! Bad data in means bad data out, but at least MNEMON will run, and a running program is a lot easier to correct than one that crashes and burns.

So again we have a data-intensive, rather than a computation-intensive, subroutine. The tables required by MNEMON are included in Appendix C8.

Print Operand

Now we come to the tricky part: printing the right operand given an opcode at some location in memory. When I disassemble object code by hand, I write the operand in two steps: first I determine the addressing mode of the given opcode, and then, if that addressing mode takes an operand, I write down the proper operand in the proper form. Proper form means including a comma and an X or a Y for every indexed instruction, including parentheses in the proper places for indirect instructions, and printing out all addresses *high* byte first, since that makes it easier to read an address.

OPERND (the subroutine that prints an operand for a given opcode in a given location in memory) will therefore determine the addressing mode for a given opcode, and then call an appropriate subroutine to handle that addressing mode:

OPERND

```
OPERND      TAX               Look up addressing mode code for
            LDA MODES,X       this opcode.
            TAX               X now indicates the addressing mode.
            JSR MODE.X        Call the subroutine that handles address-
                              ing mode "X."
            RTS               Return to caller.
```

MODES is a table giving the addressing mode for each opcode.

Note that OPERND can work only if we have a routine called MODE.X which somehow transfers control to the subroutine that handles addressing mode "X." How can MODE.X do this? One way is to have a table of pointers, in which the Xth pointer points to the subroutine that handles addressing mode "X." MODE.X must then transfer control to the Xth subroutine in this table. It would be nice if the 6502 offered an indexed JSR instruction, which would call the subroutine whose address is the Xth entry in the table. Unfortunately, the 6502 doesn't offer an indexed JSR instruction, so we'll have to simulate one in software.

Fortunately, the 6502 does offer an indirect JMP. If a pointer, called SUBPTR, can be made to point to a given subroutine, then the instruction JMP (SUBPTR) will transfer control to that subroutine. Therefore, MODE.X need only set SUBPTR equal to the Xth pointer in a table of subroutine pointers, and with the instruction

JMP (SUBPTR), it can transfer control to the Xth subroutine in the table.

HANDLE ADDRESSING MODE "X"

MODE.X	LDA SUBS,X	Get low byte of Xth pointer in the table of subroutine pointers.
	STA SUBPTR	Set low byte of subroutine pointer.
	INX	Adjust index to get next byte.
	LDA SUBS,X	Get high byte of Xth pointer in the table of subroutine pointers.
	STA SUBPTR+1	Set high byte of subroutine pointer.
	JMP (SUBPTR)	Jump to the subroutine specified by the subroutine pointer. That subroutine will then return to the *caller* of MODE.X, not to MODE.X itself.
SUBS		This is a table of pointers, in which the Xth pointer points to the subroutine that handles addressing mode X.

Disassembler Utilities

Given MODE.X, OPERND can call the right subroutine to handle any given addressing mode. Now all we need are thirteen different subroutines, one for each of the 6502's different addressing modes.

Before writing those subroutines, however, let's think for a moment about what they must do, and see if we can't write a few utility subroutines to perform those functions. With a proper set of utilities, the addressing mode subroutines themselves need only call the right utilities in the right order.

The following set of utilities seems reasonable:

- ONEBYT: Print a 1-byte operand.
- TWOBYT: Print a 2-byte operand.
- RPAREN: Print a right parenthesis.
- LPAREN: Print a left parenthesis.
- XINDEX: Print a comma and then the letter "X."
- YINDEX: Print a comma and then the letter "Y."

Print a 1-Byte Operand: ONEBYT

```
ONEBYT      JSR INC.SL       Advance to byte following opcode.
            JSR DUMPSL       Print it in hexadecimal.
            RTS              Return to caller.
```

Print a 2-Byte Operand: TWOBYT

A 2-byte operand always specifies an address with the low byte first. To print a 2-byte operand high byte first, we must first print the second byte in the operand and *then* print the first byte in the operand; each, of course, in hexadecimal format.

```
TWOBYT      JSR INC.SL       Advance to first byte of operand.
            LDA GET.SL       Load that byte into accumulator.
            PHA              Save it.
            JSR INC.SL       Advance to second byte of operand.
            JSR DUMPSL       Print it in hexadecimal format.
            PLA              Restore the operand's first byte to the
            JSR PR.BYT       accumulator, and print it in hexa-
                             decimal.
            RTS              Return to caller.
```

ONEBYT and TWOBYT each leave SELECT pointing at the last byte of the operand.

Print Right, Left Parenthesis: RPAREN, LPAREN

RPAREN prints a right parenthesis to all currently selected devices. LPAREN prints a left parenthesis to all currently selected devices.

```
RPAREN      LDA #')          Load accumulator with ASCII code for
                             right parenthesis.
            BNE SENDIT       Send it to all currently selected devices.
LPAREN      LDA #'(          Load accumulator with ASCII code for
                             left parenthesis.
SENDIT      JSR PR.CHR       Send it to all currently selected devices.
            RTS              Return to caller.
```

Index with Register X: XINDEX

XINDEX prints a comma and then the letter "X:"

XINDEX	LDA #',	Load accumulator with ASCII code for a comma; then print it to
	JSR PR.CHR	all currently selected devices.
	LDA #'X	Load accumulator with ASCII code for the letter "X;" then print it
	JSR PR.CHR	to all currently selected devices.
	RTS	Return to caller.

Index with Register Y: YINDEX

YINDEX prints a comma and then the letter "Y:"

YINDEX	LDA #',	Load accumulator with ASCII code for a comma; then print it to all
	JSR PR.CHR	currently selected devices.
	LDA #'Y	Load accumulator with ASCII code for the letter "Y;" then print it
	JSR PR.CHR	to all currently selected devices.
	RTS	Return to caller.

So much for the disassembler utilities. Now with a single subroutine call we can print a 1-byte or a 2-byte operand (and, of course, we can print a no-byte operand), and we can print any of the frequently used characters and character combinations. Okay, let's write some addressing mode subroutines:

Addressing Mode Subroutines

Because the 6502 has thirteen different addressing modes, we'll need thirteen different addressing mode subroutines:

Subroutine	Addressing Mode
ABSLUT	Absolute

ABS.X	Absolute,X
ABS.Y	Absolute,Y
ACC	Accumulator
IMPLID	Implied
IMMEDT	Immediate
INDRCT	Indirect
IND.X	Indirect,X
IND.Y	Indirect,Y
RELATV	Relative
ZEROPG	Zero Page
ZERO.X	Zero Page,X
ZERO.Y	Zero Page,Y

The main job for each subroutine will be to print the operand in the proper form. Although a given addressing mode will always have the same number of characters in its operand, unfortunately, different addressing modes may have operands of different lengths. For example, implied addressing mode has no characters in its operand, whereas indirect indexed addressing requires six characters in its operand.

But no matter how many characters appear in an operand, we want to make sure that field 3 (the address field) always begins at the same column. Therefore, every addressing-mode subroutine will return with A holding the number of characters in the operand, with X holding the number of bytes in the operand, and with SELECT pointing at the last byte in the operand (or at the opcode, if it was a 1-byte instruction). Then FINISH can print an appropriate number of spaces before printing fields 3 thru 6.

Absolute Mode: ABSLUT

To print the operand for an instruction in the absolute mode, we need only print a 2-byte operand. Thus, 8D B2 04 will disassemble as:

```
STA 04B2    8D B2 04
```

```
ABSLUT      JSR TWOBYT
            LDX #2          X holds number of bytes in operand.
            LDA #4          A holds number of characters in
                            operand.
            RTS
```

Absolute, X Mode: ABS.X

To print the operand for an instruction in the absolute, X mode, we must print a 2-byte operand, a comma, and then an "X:"

LDA D09A,X   BD 9A D0

```
ABS.X       JSR ABSLUT     Print the 2-byte operand.
            JSR XINDEX     Print the comma and the "X."
            LDX #2         X holds number of bytes in operand.
            LDA #6         A holds number of characters in
                           operand.
            RTS            Return to caller.
```

Abolute, Y Mode: ABS.Y

To print the operand for an instruction in the absolute, Y mode, we must print a 2-byte operand, a comma, and then a "Y:"

ORA 02FE,Y   19 FE 02

```
ABS.Y       JSR ABSLUT     Print the 2-byte operand.
            JSR YINDEX     Print the comma and the "Y."
            LDX #2         X holds number of bytes in operand.
            LDA #6         A holds number of characters in
                           operand.
            RTS            Return to caller.
```

Accumulator Mode: ACC

To print the operand for an instruction in the accumulator mode, we need only print the letter "A:"

ROR A   6A

```
ACC       LDA #'A          Load accumulator with ASCII code for
                           the letter A.
          JSR PR.CHR       Print it on all currently selected devices.
          LDX #0           X holds number of bytes in operand.
          LDA #1           A holds number of characters in
                           operand.
          RTS              Return to caller.
```

Implied Mode: IMPLID

Implied mode has no operand, so just return:

```
CLC   18
```

```
IMPLID    LDX #0           X holds number of bytes in operand.
          LDA #0           A holds number of characters in
                           operand.
          RTS
```

Immediate Mode: IMMEDT

Immediate mode requires a 1-byte operand, which we'll print in hexadecimal format. Thus, it should disassemble the two consecutive bytes "A9 41" as follows:

```
LDA #$41   A9 41
```

```
IMMEDT    LDA #'#          Print a '#' sign.
          JSR PR.CHR
          LDA #'$          Print a dollar sign.
          JSR PR.CHR
          JSR ONEBYT       Print 1-byte operand in hexadecimal for-
                           mat.
          LDX #1           X holds number of bytes in operand.
          LDA #4           A holds number of characters in
                           operand.
          RTS              Return to caller.
```

Indirect Mode: INDRCT

To print the operand for an instruction in the indirect mode, we need only print an absolute operand within parentheses. Thus, the three consecutive bytes "6C 00 04" will disassemble as:

JMP (0400) 6C 00 04

```
INDRCT      JSR LPAREN    Print left parenthesis.
            JSR ABSLUT    Print the 2-byte operand.
            JSR RPAREN    Print the right parenthesis.
            LDX #2        X holds number of bytes in operand.
            LDA #6        A holds number of characters in
                          operand.
            RTS           Return to caller.
```

Indirect, X Mode: IND.X

To print the operand for an instruction in the indirect, X addressing mode, we need to print a left parenthesis, a zero-page address, a comma, the letter "X," and then a right parenthesis. Thus, the two consecutive bytes "A1 3C" will disassemble as:

LDA (3C,X) A1 3C

```
IND.X       JSR LPAREN    Print a left parenthesis.
            JSR ZERO.X    Print a zero-page address, a comma, and
                          the letter "X."
            JSR RPAREN    Print a right parenthesis.
            LDX #1        X holds number of bytes in operand.
            LDA #6        A holds number of characters in
                          operand.
            RTS           Return to caller.
```

Indirect, Y Mode: IND.Y

To print the operand for an instruction in the indirect, Y mode, we must print a left parenthesis, a zero-page address, a right parenthesis, a comma, and then the letter "Y." Thus, the two consecutive bytes "B1 AF" will disassemble as:

```
LDA (AF),Y   B1 AF
```

```
IND.Y       JSR LPAREN      Print a left parenthesis.
            JSR ZEROPG      Print a zero-page address.
            JSR RPAREN      Print a right parenthesis.
            JSR YINDEX      Print a comma and then the letter "Y."
            LDX #1          X holds number of bytes in operand.
            LDA #6          A holds number of characters in
                            operand.
            RTS             Return to caller.
```

Relative Mode: RELATV

Relative mode can be tricky. A relative branch instruction specifies a forward branch if its operand is *plus* (in the range of 00 to $7F), but it specifies a backward branch if its operand is *minus* (in the range of $80 to $FF). Therefore, in order to determine the address specified by a relative branch instruction, we must first determine whether the operand is plus or minus, so we can determine whether we're branching forward or backward. Then we must add or subtract the least-significant 7 bits of the operand to or from the address immediately following the operand of the branch instruction; the result of that calculation will be the actual address specified by the branch instruction.

```
RELATV      JSR INC.SL      Select next byte in memory.
            JSR PUSHSL      Save SELECT pointer on stack.
            JSR GET.SL      Get operand byte.
            PHA             Save it on the stack.
            JSR INC.SL      Increment SELECT pointer so it points
                            to the opcode following the relative
                            branch instruction. (Relative branches
                            are relative to the next opcode.)
            PLA             Restore operand byte to accumulator.
            CMP #0          Is it plus or minus?
```

```
            BPL FORWRD       If plus, it means a forward branch.
                             Since operand byte is minus, we'll be
                             branching backward.
            DEC SELECT+1     Branching backward is like branching
                             forward from a location 256 bytes lower
                             in memory.
FORWRD      CLC              Add operand byte to the address
            ADC SELECT       of the opcode following the
            BCC RELEND       branch instruction.
            INC SELECT+1
RELEND      STA SELECT       Now SELECT points to the address
                             specified by the operand of the relative
                             branch instruction. Let's print it.
            JSR PR.ADR
            JSR POP.SL       Restore SELECT pointer.
            LDX #1           X holds number of bytes in operand.
            LDA #4           A holds number of characters in
                             operand.
            RTS              Return to caller, with SELECT pointer
                             once again pointing to the operand byte
                             of the relative branch instruction.
```

Zero-Page Mode: ZEROPG

To print the operand of an instruction that uses the zero-page addressing mode, we need only print a 1-byte operand. This will cause the bytes "85 2A" to be disassembled as:

```
STA 2A   85 2A
```

```
ZEROPG      LDA #0           Print two ASCII zeroes to all
            JSR PR.BYT       currently selected devices.
            JSR ONEBYT       Print the 1-byte operand.
            LDX #1           X holds number of bytes in operand.
            LDA #2           A holds number of characters in
                             operand.
            RTS              Return to caller.
```

Zero-Page Indexed Modes: ZERO.X, ZERO.Y

To print the operand of an instruction that uses the zero-page X or zero-page Y addressing mode, we need only print the zero-page address, a comma, and then an "X" or a "Y." Thus, "B5 6C" will disassemble as:

```
LDA 6C,X   B5 6C
```

and "B6 53" will disassemble as:

```
LDX 53,Y   B6 53
```

```
ZERO.X     JSR ZEROPG     Print the zero-page address.
           JSR XINDEX     Print a comma and the letter "X."
           LDX #1         X holds number of bytes in operand.
           LDA #2         A holds number of characters in
                          operand.
           RTS            Return to caller.
ZERO.Y     JSR ZEROPG     Print the zero-page address.
           JSR YINDEX     Print a comma and the letter "Y."
           LDX #1         X holds number of bytes in operand.
           LDA #2         A holds number of characters in
                          operand.
           RTS            Return to caller.
```

A Pseudo-Addressing Mode for Embedded Text

Now we have subroutines to disassemble machine code in any of the 6502's thirteen legal addressing modes. But what about text embedded in a machine-language program? We know that our programs already include text strings, where each text string begins with a TEX character ($7F) and ends with an ETX ($FF). The disassembler, however, doesn't know anything about embedded text. If we try to disassemble a machine-language program that includes embedded text, the disassembler will assume that the TEX character, and the text string itself, are 6502 opcodes and operands; because it doesn't know about text, it will misinterpret the text string.

Wouldn't it be nice if the disassembler could recognize the TEX character for what it is, and then print out the text string *as text*, rather than as opcodes and operands? When it has finished printing a text string, the disassembler could then resume treating the bytes following the ETX as conventional 6502 opcodes and operands.

Such behavior is not hard to implement. We need only define a pseudo-addressing mode, called TEXT mode, and say that the TEX character is the only opcode that has the TEXT addressing mode. Then we'll write a special addressing mode subroutine, called TXMODE, to print operands that are in the TEXT mode. TXMODE will print an operand in the TEXT mode by printing the text that follows the TEX character and ends with the first ETX character.

Here's some source code to implement such behavior:

```
TXMODE   PLA            Pop return address
         PLA            to OPERND.
         PLA            Pop return address
         PLA            to DSLINE.
TXLOOP   JSR NEXTSL     Advance past TEX pseudo-opcode.
         BMI TXEXIT     Return if reached EA.
         JSR GET.SL     Get the character.
         CMP #ETX       Is it the end of the text string?
         BEQ TXEXIT     If so, we've finished disassembling this
                        line.
         JSR PR.CHR     If not, print the character.
         CLC            Branch back to get
         BCC TXLOOP     the next character.
TXEXIT   JSR CR.LF      Advance to a new line.
         JSR NEXTSL     Advance to next opcode (if SELECT is
                        less than EA).
         RTS            Return to the caller of DSLINE, with
                        SELECT at the first opcode following
                        the text string.
```

Now that we have the desired addressing mode subroutines, we can make up the table of addressing mode subroutines:

```
SUBS        .WORD ABSLUT
            .WORD ABS.X
            .WORD ABS.Y
            .WORD ACC
            .WORD IMPLID
            .WORD IMMEDT
            .WORD INDRCT
```

```
.WORD IND.X
.WORD IND.Y
.WORD RELATV
.WORD ZEROPG
.WORD ZERO.X
.WORD ZERO.Y
```

Each addressing mode subroutine will return with SELECT pointing at the last byte in the instruction, with A holding the number of characters in the operand field, and with X holding the number of bytes in the operand (0, 1, or 2). Each addressing mode subroutine will return to OPERND, which will finish the line by calling FINISH.

Finishing the Line: FINISH

FINISH must space over to the proper column for field 3, which will hold the address of the opcode. Then it must print the address of the opcode and dump 1, 2 or 3 bytes, as necessary. FINISH will end by advancing the printhead to a new line and by advancing SELECT so that it points to the first byte following the disassembled line (unless it has disassembled through EA, the ending address, in which case it will return with SELECT = EA). FINISH returns PLUS if more bytes must be disassembled before EA is reached; it returns MINUS if it disassembled through EA.

```
FINISH      STA OPCHRS      Save the length of the operand,
            STX OPBYTS      in characters and in bytes.
            DEX             If necessary, decrement the
            BMI SEL.OK      SELECT pointer so it
LOOP.1      JSR DEC.SL      points to the opcode.
            DEX
            BPL LOOP.1
SEL.OK      SEC             Space over to the
            LDA ADRCOL      column for the address field:
            SBC #4          Operand field started in column 4...
            SBC OPCHRS      ... and includes OPCHRS characters.
            TAX             So now we need X spaces.
            JSR SPACES      Send enough spaces to reach address
                            column.
            JSR PR.ADR      Print address of opcode.
LOOP.2      JSR SPACE       Space once.
            JSR DUMPSL      Dump selected byte.
            JSR INC.SL      Select next byte.
```

```
            DEC OPBYTS     Completed last byte in instruction?
            BPL LOOP.2     If not, do next byte.
            JSR DEC.SL     Back up SELECT to last byte in
                           operand.
FINEND      JSR CR.LF      Advance to a new line.
            RTS            Return to caller.
OPBYTS      .BYTE          Number of bytes in operand.
OPCHRS      .BYTE 0        Number of characters in operand.
ADRCOL      .BYTE 16       Starting column for address field.
```

Now we can disassemble a line. So let's write the disassemblers, one for the printer and one for the screen. These routines will have much the same structure as TVDUMP and PRDUMP, which direct hexdumps to the printer or to the screen.

Disassemble to Screen: TV.DIS

```
TV.DIS      LDA DISLNS     Initialize line counter with
            STA LINUM      number of lines to be disassembled.
            LDA #$FF       Set end address to $FFFF,
            STA EA         so NEXTSL will always increment
            STA EA+1       the SELECT pointer.
            JSR TVT.ON     Select TVT as an output device. (Other
                           selected devices will echo the
                           disassembly.)
TVLOOP      JSR DSLINE     Disassemble one line.
            DEC LINUM      Completed last line yet?
            BNE TVLOOP     If not, disassemble next line.
            RTS            If so, return.
DISLNS      .BYTE 5        DISLNS holds number of lines to be
                           disassembled by TV.DIS. To disassem-
                           ble one line, set DISLNS=1.
LINUM       .BYTE 0        This variable keeps track of the number
                           of lines yet to be disassembled.
```

Printing Disassembler: PR.DIS

The printing disassembler (PR.DIS) will announce itself by displaying "PRINTING DISASSEMBLER" on the screen, but not on the printer. It will then let the user set the starting and ending addresses, in the same manner as PRDUMP. When the user has specified the block of memory to be disassembled, the PR.DIS will print a disassembly of the specified block of memory, echoing its output to the screen.

```
PR.DIS      JSR PR.OFF          Deselect printer.
            JSR TVT.ON          Select TVT.
            JSR PRINT:          Display title:
            .BYTE TEX
            .BYTE CR,LF
            .BYTE               'PRINTING DISASSEMBLER'
            .BYTE CR,LF,ETX
            JSR.SETADS          Let user set starting address
                                and end address.
            JSR GOTOSA          Set SELECT = Start address.
            JSR PR.ON           Select the printer.
PRLOOP      JSR DSLINE          Disassemble one line.
            BPL PRLOOP          If it wasn't the last line, disassemble the
                                next one.
            RTS                 Return to caller.
```

With PR.DIS and TV.DIS, you can disassemble any block of memory, directing the disassembly to the screen or to the printer. See Chapter 12 for guidance on mapping these two disassemblers to function keys in the Visible Monitor.

Chapter 10:

A General MOVE Utility

Many computer programs spend a lot of time moving things from one place to another. Such programs should be able to call a move utility for most of this work. A move utility should:

- Be general enough to move anything of any size from any place in memory to anywhere else.
- Not be upset when the origin block overlaps the destination.
- Have entry points with input configurations convenient to different callers.
- Preserve its inputs.
- Be *fast*.

This routine will be called often. A calling program doesn't want to spend all its time here. The cost of that speed is size, because we'll use straight-line, dedicated code to handle each of several special cases, but even so this move code will weigh in at less than 200 bytes. That's less than three percent of the memory available on a system with 8 K bytes of programmable memory.

Input Configurations

Different callers may find different input configurations convenient, so let's provide more than one entry point, each requiring different parameters to be set. The following two subroutine entry points are likely to meet the needs of most callers:

MOV.EA Move a block, defined by its starting address (SA), its ending address (EA), and its destination address (DEST).

MOVNUM — Move a block, defined by its starting address, the number of bytes in the block (NUM), and the destination of the block.

MOV.EA will simply be a "front end" for MOVNUM. It will set NUM = ending address − starting address of the source block.

Handling Overlap

There will be no problem with overlap if we always move from the leading edge of the source block — that is, copy *up* beginning with the highest byte to be moved, and copy *down* beginning with the lowest byte to be moved. This way, if a byte in the source block is overwritten it will already have been copied to its destination.

Going Up?

To avoid overlap, MOVNUM must determine whether it's copying up or down. Therefore, before moving anything it must see if the destination address is greater or lesser than the starting address. Then it can branch to MOVE-UP or MOVE-DOWN as appropriate.

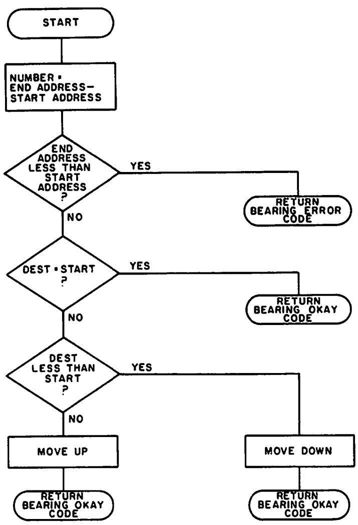

Figure 10.1: *Top level of block move. Flowchart of MOVE.EA and MOVNUM routines.*

Using the flowchart of figure 10.1 as a guide, let's write source code for the top level of MOV.EA and MOVNUM:

```
        GETPTR = 0              This is the input-page pointer.
        PUTPTR = GETPTR+2       This is the output-page pointer.
MOV.EA  SEC                     Set NUM = EA - SA
        LDX EA+1
        LDA EA
        SBC SA
        STA NUM
        BCS MOVE.1
        DEX
        SEC
MOVE.1  TXA
        SBC SA+1
        STA NUM+1
        BCS MOVNUM              Now NUM = EA - SA.
ER.RTN  LDA #ERROR              If EA less than SA,
        RTS                     return with error code.
MOVNUM  LDY #3                  Save the 4 zero-page
SAVE    LDA GETPTR,Y            bytes we'll use.
        PHA
        DEY
        BPL SAVE
        SEC                     Is DEST less than START?
        LDA SA+1
        CMP DEST+1
        BCC MOVEUP              If so, we'll move down.
        BNE MOVEDN              If not, we'll move up.
        LDA SA                  SA, destination are in the same
                                page.
        CMP DEST                If SA more than destination, we'll
        BCC MOVEUP              move down. If SA less than destina-
                                tion,
        BNE MOVEDN              we'll move up. If they are equal, we'll
                                return bearing okay code.
OK.RTN  LDY #0                  Restore 4 zero-page bytes that were
RESTOR  PLA                     used by the move code.
        STA GETPTR,Y
        INY
        CPY #4                  Restored last byte yet?
        BNE RESTOR              If not, restore next one. If so,
        RTS                     return, with move complete and zero
                                page preserved.
NUM     .WORD 0                 This 16-bit variable holds the number of
                                bytes to be moved.
```

Optimizing for Speed

Moving a page at a time is the fastest way to move data, and for large blocks we can move most of the bytes this way. Therefore, when moving data we'll move one page at a time until there is less than a page to move; then we'll move a byte at a time until the entire source block is moved. MOVE-UP and MOVE-DOWN must test to see if they have more or less than a page to move, and then branch to dedicated code that either moves a page or moves less than a page.

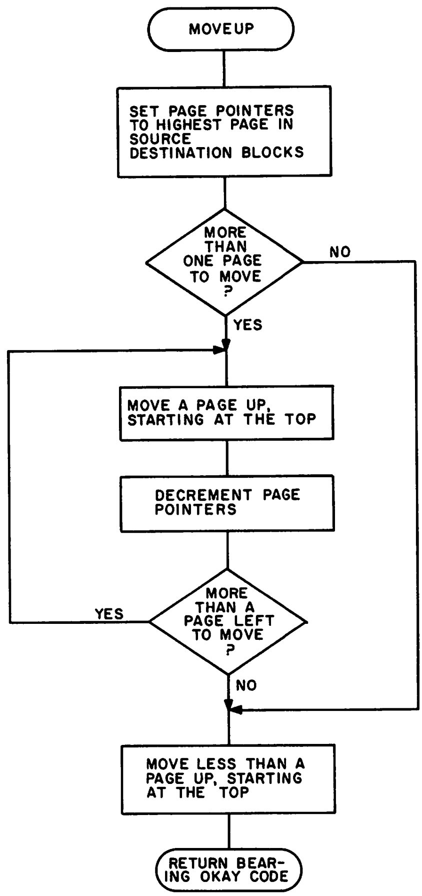

Figure 10.2: *Move a block up. Flowchart of the MOVEUP routine.*

MOVE-UP

Using figure 10.2 as a guide, we can write source code for MOVE-UP:

```
MOVEUP    LDA NUM+1          More than one page to move?
          BEQ LESSUP         If not, move less than a page up.
                             To move more than a page, set the page
                             pointers GETPTR and PUTPTR to the
                             highest pages in the source and destina-
                             tion blocks. To do this, treat X as the
                             high byte and Y as the low byte of a
                             pointer, which we'll call (X,Y). First set
                             (X,Y) = NUM − $FF, the relative ad-
                             dress of the highest page in the block.
          LDY NUM+1          Now Y is high byte of block size.
          LDA NUM            Now A is low byte of block size.
          SEC                Prepare to subtract.
          SBC #$FF.          Now A is a low byte of (block size −
                             $FF.)
          BCS NEXT.1
          DEY
NEXT.1    TAX                Now (X,Y) = NUM − $FF.
                             X is low byte, Y is high byte of NUM −
                             $FF.
          STY PUTPTR+1
          TXA
          CLC                Prepare to add.
          ADC SA
          STA GETPTR
          BCC NEXT.2
          INY
NEXT.2    TYA
          ADC SA+1
          STA GETPTR+1       Now GETPTR = SA + NUM − $FF
                             (the last page in the origin block).
          TXA
          CLC                Prepare to add.
          ADC DEST
          STA PUTPTR
          BCC NEXT.3
          INC PUTPTR+1
NEXT.3    LDA PUTPTR+1
          ADC DEST+1
          STA PUTPTR+1       Now PUTPTR = DEST + NUM − $FF
                             (the last page in the destination block).
                             Now the page pointers (GETPTR and
                             PUTPTR) point to the last page in, respec-
                             tively, the origin and destination blocks.
```

```
        LDX NUM+1         Load X with number of pages to move.
PAGEUP  LDY #$FF          Move a page up.
UPLOOP  LDA (GETPTR),Y    Get a byte from origin block.
        STA (PUTPTR),Y    Put it in destination block.
        DEY               Adjust index for next byte down.
        BNE UPLOOP        Loop if not the last byte.
        LDA (GETPTR),Y    Move last byte.
        STA (PUTPTR),Y
        DEC GETPTR+1      Decrement page pointers.
        DEC PUTPTR+1
        DEX               Still more than a page to move?
        BNE PAGEUP        If so, move up another page.
LESSUP  JSR LOPAGE        Set GETPTR, PUTPTR to bottom of
                          origin and destination blocks.
        LDY NUM           Set index to number of bytes to be
                          moved.
SOMEUP  LDA (GETPTR),Y    Move a byte.
        STA (PUTPTR),Y
        DEY               About to move last byte?
        CPY #$FF
        BNE SOMEUP        If not, move another.
        JMP OK.RTN        If so, return bearing "OK" code.
LOPAGE  LDA SA            Set page pointers to the bottom
        STA GETPTR        of the origin and destination
        LDA SA+1          blocks.
        STA GETPTR+1
        LDA DEST
        STA PUTPTR
        LDA DEST+1
        STA PUTPTR+1
        RTS               Return to caller.
```

Move-Down: MOVEDN

Figure 10.3 shows an algorithm for moving a block of data down through memory.

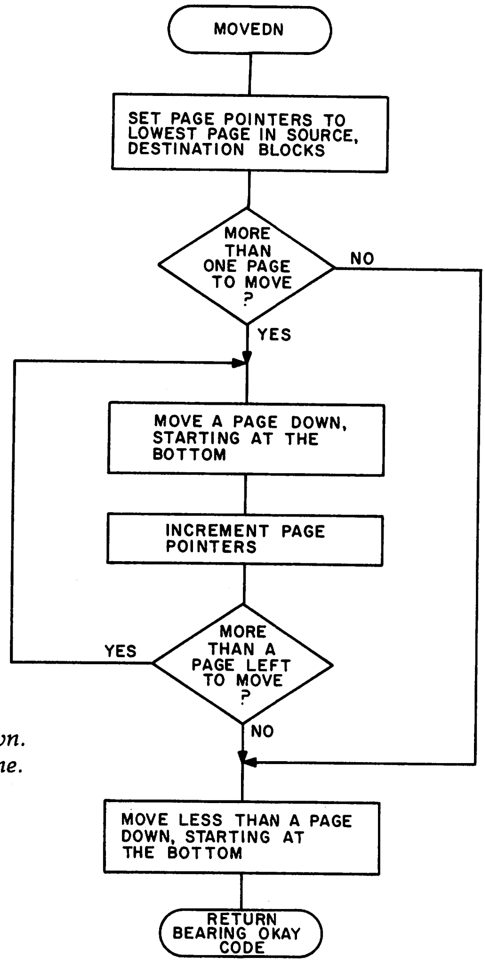

Figure 10.3: *Move a block down. Flowchart of the MOVEDN routine.*

Using figure 10.3 as a guide, we can write source code for the move-down routine:

```
MOVEDN   JSR LOPAGE        Set page pointers to bottom of origin
                           and destination blocks.
         LDY #0            Y must equal zero whether we move
                           more or less than a page.
         LDX NUM+1         More than one page to move?
         BEQ LESSDN        If not, move less than a page down.
                           Move a page down.
PAGEDN   LDA (GETPTR),Y    Get a byte from origin block
         STA (PUTPTR),Y    and put it in destination block.
         INY               Moved last byte in page?
```

```
            BNE PAGEDN
            INC GETPTR+1        Increment page pointers.
            INC PUTPTR+1
            DEX                 Still more than a page to move?
            BNE PAGEDN          If so, move another page down.
            LDY #0              Move less than a page down starting at
                                the bottom.
LESSDN      LDA (GETPTR),Y      Get a byte from origin...
            STA (PUTPTR),Y      and put it in destination block.
            INY                 Adjust index for next byte.
            SEC
            CPY NUM             Moved last byte yet?
            BCC LESSDN          If not, move another.
            JMP OK.RTN          If so, return to caller, bearing "OK"
                                code.
```

Speed

For large blocks of data, most bytes will be moved by the page-moving code: PAGE-UP and PAGE-DOWN. Since the processor spends most of its time in these loops, let's see how long they will take to move a byte. (Appendix A5, *Instruction Execution Times*, provides information on the number of cycles required for each 6502 operation.) Ordinarily I would not go into great detail concerning the speed of execution of a small block of code, but these two loops form the heart of the move utility, because they move most of the bytes in any large block. By making those two loops very efficient, we can make the move utility very fast. In fact, these loops will let us move blocks bigger than one page, at a rate approaching 16 cycles/byte moved. (By way of a benchmark, that's more than twice as fast as the time required to move large blocks with MOVIT, a smaller move program published in *The First Book of KIM.** MOVIT, made tiny [95 bytes] to use as little as possible of the KIM's limited programmable memory, requires at least 33 cycles/bytes moved.)

MOVE.EA and MOVNUM are move utilities because they have input configurations and performance suitable for many calling programs. But they are not very convenient to the human user who simply wants to move something. With the Visible Monitor and the move utility, you can move something from one place to

---

*Butterfield, et al, *The First Book of Kim*, Rochelle Park, NJ: Hayden Book Company, 1977.

another, but you have to know what addresses to set and you have to know the address of the move utility itself.

That's too much for me to remember. I want a *tool,* which will know the addresses and won't require me to remember them.

When I'm developing programs with the Visible Monitor and I want to move some data or code from one place to another, I'd like to be able to call up a move tool with a single keystroke — say "M." It's easier for me to remember " 'M' for Move" than it is to remember the address of the move utility and the addresses of its inputs.

Let's say I'm using the Visible Monitor and I press "M." This invokes the move tool. The first thing it should do is let me know that it's active. What if I hit the "M" key by mistake? The computer should let me know that I've invoked a new program.

It should put up a title: "MOVE TOOL." Then it should let me specify the start, end, and destination addresses of a given block in memory. When these addresses are set, the move tool can call MOV.EA, which will actually perform the move, based on the addresses set by the user.

The top level of the move tool is therefore quite simple. Figure 10.4 shows the flowchart for the following routine:

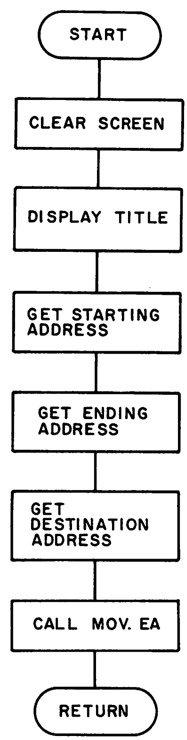

Figure 10.4: *A move tool. Flowchart of MOVER routine.*

MOVER

```
MOVER   JSR TVT.ON              Select screen as an output device.
        JSR PRINT:              Put a title on the screen.
        .BYTE TEX,CR
        .BYTE '  MOVE TOOL'
        .BYTE CR,LF,LF
        .BYTE ETX
        JSR SETADS              Get starting address,
                                ending address, and
        JSR SET.DA              destination address from user.
        JSR MOV.EA              Move the block specified by those
                                pointers.
        RTS                     Return to caller, with requested block
                                moved and with zero page preserved.
```

Of course, MOVER can work only if we have a routine that lets the user set the destination address. Let's write such a routine, and we'll be all set to move whatever we like, to wherever we want it.

Set Destination Address: SET.DA

```
SET.DA  JSR TVT.ON              Select TVT as an output device. All
                                other selected output devices will echo
                                the screen output.
        JSR PRINT:              Put prompt on the screen:
        .BYTE TEX
        .BYTE CR,LF,LF
        .BYTE                   "SET DESTINATION ADDRESS  "
        .BYTE                   "AND PRESS Q."
        .BYTE ETX
        JSR VISMON              Call the Visible Monitor, so user can
                                specify a given address.
DAHERE  LDA SELECT              Set destination address equal to
        STA DEST                address set by the user.
        LDA SELECT+1
        STA DEST+1
        RTS                     Return to caller.
DEST    .WORD 0                 Pointer to destination of block to be
                                moved.
```

See Chapter 12, *Extending the Visible Monitor,* to learn how to hook the move tool into the Visible Monitor by mapping it to a given key. Then to move anything in memory to anywhere else, you need only strike that key and the move tool will do the rest.

Chapter 11:

A Simple Text Editor

With the Visible Monitor you can enter ASCII text into memory by placing the arrow under field 2 and striking character keys. But you must strike two keys for every character in the message: first the character key, to enter the character into the displayed address, and then the space bar, to select the next address. Furthermore, if you want to enter an ASCII space or carriage return into memory, you'll have to place an arrow under field 1 and enter the hexadecimal representation of the desired character: $20 for a space; $0D for a carriage return. Then, of course, you'll have to hit the space bar to select the next address, and the RIGHT-ARROW key to move the arrow back underneath field 2, so that you can enter the next character into memory.

If you only need to enter up to a dozen ASCII characters at a time, then the Visible Monitor should meet your needs. When you need to enter longer messages into memory, you'll find yourself wanting a more suitable tool — a simple text editor.

Text editors come in many different shapes, sizes and formats. A line-oriented editor, suitable for creating and editing program source files, requires that you enter and edit text a line at a time. Usually each line must be numbered when it is entered; then, in order to edit a line, you must first specify it by its line number.

On the other hand, a character-oriented editor allows you to overstrike, insert, or delete characters anywhere in a given string of characters. Character-oriented editors are frequently found in word processors for office applications, but don't get your hopes up; this chapter will not present software nearly as sophisticated as that available in even the humblest of word processors. However, it will present a very simple character-oriented editor that will enable you to enter and edit text strings, such as prompts, anywhere in memory.

Structure

The text editor will have the three-part structure shown in figure 11.1. From this we can write source code for the top level of the text editor:

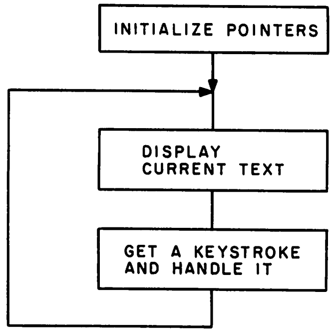

Figure 11.1: *Structure of simple text editor.*

```
EDITOR   JSR SETBUF      Initialize pointers and variables required
                         by the editor.
EDLOOP   JSR SHOWIT      Show the user a portion of the text
                         buffer.
         JSR EDITIT      Let the user edit the buffer or move
                         about within it.
         CLC
         BCC EDLOOP      Loop back to show the current text.
```

Look familiar? It should. This is essentially the same structure used in the Visible Monitor. It's a simple structure, well-suited to the needs of many interactive display programs.

SETBUF

The text editor will operate on text in a portion of memory called the *text buffer*. Because the editor must be able to change the contents of the text buffer, the buffer must occupy programmable memory and may not be used for any other purpose. This exemplifies a problem familiar to programmers: how to allocate memory in the most effective manner. Memory used to store a program cannot be used at the same time to store text; nor can memory allotted to the text buffer be used for storing programs or variables.

How do you get five pounds of tomatoes into a four-pound-capacity sack — without crushing the tomatoes or tearing the sack? You don't. If you want to store a lot of text in your computer's programmable memory, you might not have room for much of a text editor. On the other hand, an elaborate text editor, requiring a good deal of programmable memory for its own code, may not leave much room in your system for storing text.

Therefore, this text editor leaves the allocation of memory for the text buffer to the discretion of the user. A subroutine called SETBUF sets pointers to the starting and ending addresses of the text buffer. The rest of the editor then operates on the text buffer defined by those pointers.

SETBUF sets the starting and ending addresses of the edit buffer. If you always want to enter and edit text in the same buffer, then substitute your own subroutine to set the starting and ending addresses to the values you desire. Otherwise, use the following version of SETBUF, which lets the user define a new text buffer each time it is called.

For testing purposes, you might even want to set the text buffer completely inside screen memory. This allows you to *see* exactly what's happening inside the text buffer.

SETBUF

```
SETBUF   JSR TVT.ON                Select TVT.
         JSR PRINT:                Display "SET UP EDIT BUFFER."
         .BYTE TEX,CR,LF,LF
         .BYTE 'SET UP EDIT BUFFER'
         .BYTE CR,LF,LF,ETX
GETADS   JSR SETADS                Let user set starting address and end ad-
                                   dress of edit buffer.
         JSR GOTOSA                Now SELECT = starting address of edit
                                   buffer.
         RTS                       Return to caller.
```

This version of SETBUF allows the user to set the text buffer anywhere in memory, provided that the ending address is not lower in memory than the starting address. It returns with the SELECT pointer pointing at the starting address of the buffer.

SHOWIT

Now that SETBUF has set the pointers associated with the text buffer, let's figure out how to display part of that buffer.

Figure 11.2 shows the simple 3-line display to be used by the text editor. "X" marks the home position of the edit display. Everything in the edit display is relative to the home position. Thus, to move the edit display about on your screen (ie: from the top of the screen to the bottom of the screen), you need only change the home position, which is set by SHOWIT.

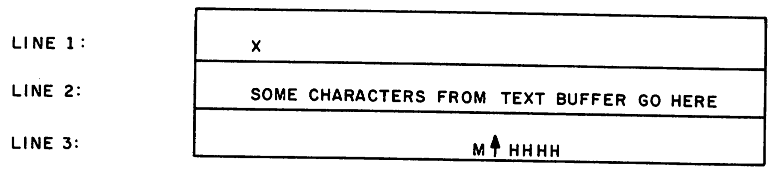

Figure 11.2: *Three-line display of simple text editor.*

Line 1 is entirely blank. Its only purpose is to separate the text displayed in line 2 from whatever you may have above it on your screen.

Line 2 displays a string of characters from the edit buffer. The central character in line 2 is the *current character*. The current character is indicated by an upward-pointing arrow as in line 3. The address of the current character is given by the four hexadecimal characters represented by "HHHH" in line 3.

The letter "M" in line 3 shows you where a graphic character will indicate the current mode of the editor.

Modes

This editor will have two modes: *overstrike mode* and *insert mode*. In overstrike mode you overstrike, or replace, the current character with the character from the keyboard. In insert mode, you insert the keyboard character into the text buffer just before the current character. How one sets these modes, a function for the subroutine EDITIT, will be discussed later. But SHOWIT must know the current mode in order to display the proper graphic in line 3 of the editor display.

Since we're going to have two modes, let's keep track of the current mode of the editor with a 1-byte variable called EDMODE. We'll assign the following values to EDMODE:

EDMODE = 0 when the editor is in overstrike mode.
EDMODE = 1 when the editor is in insert mode.

Any other value of EDMODE is undefined and therefore illegal. If SHOWIT should find that EDMODE has an illegal value, then it should set EDMODE to some legal default value — say, zero. That would make overstrike the default mode for the editor.

We'll also need two graphics characters, INSCHR and OVRCHR, to indicate insert and overstrike modes, respectively. In this chapter, the character to indicate a given edit mode will simply be the first initial of the mode name: "O" for overstrike mode, "I" for insert mode.

SHOWIT

```
SHOWIT   JSR TVPUSH      Save the zero-page bytes we'll use.
         JSR TVHOME      Set home position of the
                         edit display.
         LDX TVCOLS      Clear 3 rows for the
         LDY #3          edit display.
         JSR CLR.XY
         JSR TVHOME      Restore TV.PTR to home position of
                         edit display.
         JSR TVDOWN      Set TV.PTR to beginning of
         JSR TVPUSH      line 2 and save it.
         JSR LINE.2      Display text in line 2.
         JSR TV.POP      Set TV.PTR to beginning
         JSR TVDOWN      of line 3.
         JSR LINE.3      Display line 3.
         JSR TV.POP      Restore zero-page bytes used.
         RTS             Return to caller, with edit display on
                         screen, rest of screen unchanged, and
                         zero page preserved.
```

Of course, SHOWIT can work only if it can call a couple of routines (LINE.2 and LINE.3) to display lines 2 and 3 of the editor display, respectively. Let's write those routines.

Display Text Line

To display the text line, we simply need to copy a number of characters from the text buffer to the second line of the editor display. Since the screen is TVCOLS wide, we should display TVCOLS number of characters in such a way that the central character in the display is the currently selected character. We can do that if we decrement SELECT by TVCOLS/2 times, and then display TVCOLS number of characters:

LINE.2

```
LINE.2    JSR PUSHSL       Save SELECT pointer.
          LDA TVCOLS       Set X equal
          LSR A            to half the width
          TAX              of the screen.
          DEX
LOOP.1    JSR DEC.SL       Decrement SELECT X times.
          DEX
          BPL LOOP.1
          LDA TVCOLS       Initialize COUNTR. (We're
          STA COUNTR       going to display TVCOLS characters.)
LOOP.2    JSR GET.SL       Get a character from buffer.
          JSR TV.PUT       Put it on screen.
          JSR TVSKIP       Go to next screen position.
          JSR INC.SL       Advance to next byte in buffer.
          DEC COUNTR       Done last character in row?
          BPL LOOP.2       If not, do next character.
          JSR POP.SL       Restore SELECT from stack.
          RTS              Return to caller.
```

Display Status Line

Line 3 of the editor display provides status information: identifying the current mode of the editor, pointing at the current character in line 2 of the edit display, and providing the address of the current character.

LINE.3

```
LINE.3   LDA TVCOLS
         LSR A            A = TVCOLS/2
         SBC #2           A = (TVCOLS/2) - 2
         JSR TVPLUS       Now TV.PTR is pointing 2 characters to
                          the left of center of line 3 of the edit
                          display.
         LDA EDMODE       What is current mode?
         CMP #1           Is it insert mode?
         BNE OVMODE       If not, it must be overstrike mode.
         LDA #INSCHR      If so, load A with the insert graphic.
         CLC
         BCC TVMODE
OVMODE   LDA #OVRCHR      Load A with the overstrike graphic.
TVMODE   JSR TV.PUT       Put mode graphic on screen.
         LDA #2
         JSR TVPLUS       Now TVPTR is pointing at the center of
                          line 3 of the edit display.
         LDA ARROW        Display an up-arrow here,
         JSR TV.PUT       pointing up at the current character.
         LDA #2
         JSR TVPLUS       Now TV.PTR is pointing at the position
                          reserved for the address of the current
                          character.
         LDA SELECT+1     Display address of current
         JSR VUBYTE       character.
         LDA SELECT
         JSR VUBYTE
         RTS              Return to caller.
```

We've chosen to define the editor's current character as the character pointed to by SELECT. We've already developed some subroutines that operate on the SELECT pointer and on the currently selected byte, so we won't have to write many new editor utilities; instead, we can use many of the SELECT utilities presented in earlier chapters.

Edit Update

Now we can display the three lines of the edit display. What else must the editor do? Oh, yes: it must let us edit. Here's a reasonably useful, if small, set of editor functions:

- Allow the user to move forward through the message.
- Allow the user to move backward through the message.
- Allow the user to overstrike the current character.
- Allow the user to delete the current character.
- Allow the user to delete the entire message.
- Allow the user to insert a new character at the current character position.
- Allow the user to change modes from insert to overstrike and back again.
- Print the message.
- Allow the user to terminate editing, thus causing the editor to return to its caller.

What keys will perform these functions? I'll leave that up to you by treating the editor function keys as variables and keeping them in a table called EDKEYS (see Appendix C11). To assign a given function to a given key, store the character code generated by that key in the appropriate place in the table:

EDITIT

EDITIT	JSR GETKEY	Get a keystroke from the user.
	CMP QUITKY	Is it the "quit" key?
	BNE DO.KEY	If not, do what the key requires.
	PHA	Save the key on the stack. If the user gives us 2 "quit" keys in a row, we should exit the editor. So let's see if another QUITKY follows:
	JSR GETKEY	
	CMP QUITKY	Is this key a "quit" key?
	BNE NOTEND	If not, then this is not the end of the edit session, so we'd better handle both of those keys, and in their original order.
		End the edit session:
ENDEDT	PLA	Pop first "quit" key from stack.
	PLA	Pop from stack the return address to
	PLA	the editor's top level.
	RTS	Return to the editor's caller.
NOTEND	STA TEMPCH	Save the key that followed the "quit" key.
	PLA	Pop first "quit" key from stack.
	JSR DO.KEY	Handle it.
	LDA TEMPCH	Restore to the accumulator the key that followed the "quit" key.

```
                        "DO.KEY" does what the key in the ac-
                        cumulator requires:
DO.KEY   CMP MODEKY     Is it the "change mode" key?
         BNE IFNEXT     If not, perform the next test.
         DEC EDMODE     If so, change the editor's mode...
         BPL DO.END
         LDA #1
         STA EDMODE
DO.END   RTS            and return.
IFNEXT   CMP NEXTKY     Is it the "next" key?
         BNE IFPREV     If not, perform the next test.
         JSR NEXTCH     If so, advance the current position by
                        one character...
         RTS            and return.
IFPREV   CMP PREVKY     Is it the "previous" key?
         BNE IF.RUB     If not, perform the next test.
         JSR PREVCH     If so, back up the current position by
                        one character...
         RTS            and return.
IF.RUB   CMP RUBKEY     Is it the "delete" key?
         BNE IF.PRT     If not, perform the next test.
         JSR DELETE     If so, delete the current character...
         RTS            and return.
IF.PRT   CMP PRTKEY     Is it the "print" key?
         BNE IFFLSH     If not, perform the next test.
         JSR PRTBUF     If so, print the buffer...
         RTS            and return.
IFFLSH   CMP FLSHKY     Is it the "flush" key?
         BNE CHARKY     If not, perform the next test.
         JSR FLUSH      If so, flush all text in the edit buffer...
         RTS            and return.
                        OK. It's not an editor function key, so it
                        must be a regular character key. There-
                        fore, if we're in overstrike mode we'll
                        overstrike the current character with the
                        new character, and if we're in insert
                        mode we'll insert the new character at
                        the current character position.
CHARKY   LDX EDMODE     Are we in overstrike mode?
         BEQ STRIKE     If so, overstrike the character.
         JSR INSERT     If not, insert the character...
         RTS            and return.
STRIKE   JSR PUT.SL     Put the character into the currently
                        selected address, which is the address of
```

```
                                the current character.
         JSR NEXTSL             Advance to the next character position,
         RTS                    and return to caller.
INSERT   PHA                    Save the character to be inserted, while
                                we make space for it in the edit buffer...
         JSR PUSHSL             Push the address of the current character
                                onto the stack.
         LDA SA+1               Push starting address of the buffer
         PHA                    onto stack.
         LDA SA
         PHA
         LDA EA+1               Push ending address of the buffer
         PHA                    onto stack.
         LDA EA
         PHA
         JSR SAHERE             Set SA = SELECT, so current character
                                will be the start of the block we'll move.
         JSR NEXTSL             Advance to next character position in
                                the text buffer.
         BMI ENDINS             If we're at the end of the buffer, we'll
                                overstrike instead of inserting.
         JSR DAHERE             Set DEST = SELECT, so destination of
                                block move will be 1 byte above block's
                                start address (ie, we'll move a block up
                                by 1 byte).
         LDA EA                 Decrement end address
         BNE NEXT               so we won't move text
         DEC EA+1               beyond the end of
NEXT     DEC EA                 the text buffer.
                                Now the starting address is the current
                                character, the destination address is the
                                next character, and the ending address is
                                one character shy of the last character in
                                the buffer. We're ready now to move a
                                block.
OPENUP   JSR MOV.EA             Open up 1 byte of space at the current
                                character's location, by moving to DEST
                                the block specified by SA and EA.
ENDINS   PLA                    Restore EA so it points to the last byte
         STA EA                 in the edit buffer.
         PLA
         STA EA+1
         PLA                    Restore SA so it points to the first byte
         STA SA                 in the edit buffer.
```

PLA	
STA SA+1	
JSR POP.SL	Restore SELECT so it points to the current character.
PLA	Reload the accumulator with the character to be inserted. Since we've created a 1-byte space for this character, we need only overstrike it.
JSR STRIKE	
RTS	Return to caller.

EDITIT looks like it will do what we want it to do — provided that it may call the following (as yet unwritten) subroutines:

- NEXTCH — Select next character.
- PREVCH— Select previous character.
- FLUSH — Flush the buffer.
- PRTBUF — Print the buffer.

Let's write them.

Select Next Character

We want to be able to advance through the text buffer, but we don't want to be able to go beyond the end of the buffer or beyond the end of the message. The end of the message will be indicated by one or more ETX (end-of-text) characters. ETX characters will fill from the last character in the message to the end of the buffer. So if the current character is an ETX, we shouldn't be allowed to advance through memory. Or, if the current character is the last byte in the edit buffer, we shouldn't be allowed to advance through memory. But if we aren't at the end of our text for one reason or another, select the next character by calling the NEXTSL subroutine:

NEXTCH

NEXTCH	JSR GET.SL	Get currently selected character.
	CMP #ETX	Is it an ETX?
	BEQ AN.ETX	If so, return to caller, bearing a negative return code.

```
            JSR NEXTSL      If not, select next byte in the buffer, and
            RTS             return positive if we incremented
                            SELECT; negative if SELECT already
                            equaled EA.
AN.ETX      LDA #$FF        Since we are on an ETX, we won't incre-
                            ment
            RTS             SELECT; we'll just return with a
                            negative return code.
```

Select Previous Character

The PREVCH (select-previous-character routine) should work in a manner similar to that used by NEXTCH. NEXTCH increments the SELECT pointer and returns *plus*, unless SELECT is greater than or equal to EA, in which case NEXTCH preserves SELECT and returns *minus*. Conversely, PREVCH should decrement SELECT and return *plus*, unless SELECT is less than or equal to SA, in which case it should preserve SELECT and return *minus*:

PREVCH

```
PREVCH      SEC             Prepare to compare.
            LDA SA+1        Is SELECT in a higher page than SA?
            CMP SELECT+1
            BCC SL.OK       If so, SELECT may be decremented.
            BNE NOT.OK      If SELECT is in a lower page than SA,
                            then it's not okay. We'll have to fix it.
                            SELECT is in the same page as SA.
            LDA SA          Is SELECT greater than SA?
            CMP SELECT
            BEQ NO.DEC      If SELECT = SA, don't decrement it.
            BNE NOT.OK      If SELECT is less than SA, it's not okay,
                            so we'll have to fix it.
SL.OK       JSR DEC.SL      SELECT is OK, because it's greater than
                            SA. Thus, we may decrement it and it
                            will remain in the edit buffer.
            LDA #0          Set a positive return code...
            RTS             and return.
NOT.OK      LDA SA          Since SELECT is less than SA, it is
            STA SELECT      not even in the edit buffer. So give
            LDA SA+1        SELECT a legal value, by setting
                            it = SA.
```

```
         STA SELECT+1
         LDA #0           Set a positive return code...
         RTS.             and return.
NO.DEC   LDA #$FF         SELECT = SA, so change nothing. Set
         RTS              a negative return code and return.
```

Flush Buffer

To flush the buffer, we'll just fill the buffer with ETX characters:

FLUSH

```
FLUSH    JSR GOTOSA       Set SELECT to the first character posi-
                          tion in the buffer.
FLOOP    LDA #ETX         Load accumulator with an ETX
                          character...
         JSR PUT.SL       and put it into the buffer.
         JSR NEXTSL       Advance to next byte.
         BPL FLOOP        If we haven't reached the last byte in the
                          buffer, let's repeat the operation for this
                          byte.
         JSR GOTOSA       If we have reached the last byte in the
                          buffer, let's set SELECT to the beginning
                          of the buffer...
         JSR RTS          and return.
```

Print Buffer

To print the buffer, we must print the characters in the edit buffer up to, but not including, the first ETX. Even if there is no ETX in the buffer, we must not print characters from beyond the end of the buffer:

PRTBUF

```
PRTBUF   JSR GOTOSA       Set SELECT to the start of the buffer.
PRLOOP   JSR GET.SL       Get the currently selected character.
         CMP #ETX         Is it an ETX character?
         BEQ ENDPRT       If so, stop printing and return.
```

```
         JSR PR.CHR     If not, print it on all currently selected
                        devices.
         JSR NEXTCH     Advance SELECT by 1 byte within the
                        buffer.
         BPL PRLOOP     If we haven't reached the end of the buf-
                        fer, let's get the next character from the
                        buffer, and handle it.
ENDPRT   RTS            Since we reached the end of the buffer,
                        let's return.
                        When this routine returns, the current
                        character is at the end of the message.
```

Delete Current Character

To delete the current character, we'll take all the characters that follow it in the text buffer and move them to the left by 1 byte. Here's some code to implement such behavior:

```
DELETE   JSR PUSHSL     Save address of current character.
         LDA SA+1       Save buffer's start address.
         PHA
         LDA SA
         PHA
         JSR DAHERE     Set DEST = SELECT, because we'll
                        move a block of text down to here, to
                        close up the buffer at the current
                        character.
         JSR NEXTSL     Advance by 1 byte through text buffer,
                        if possible.
         JSR SAHERE     Set SA = SELECT, because the block
                        we'll move starts 1 byte above the cur-
                        rent character. (Note: the end address of
                        the block we'll move is the end address
                        of the text buffer.)
         JSR MOV.EA     Move block specified by SA, EA, and
                        DEST.
         PLA            Restore initial SA (which
         STA SA         is the start address of the
         PLA            text buffer, not of the block
         STA SA+1       we just moved).
```

JSR POP.SL	Restore SELECT = address of the current character.
RTS	Return to caller.

That's the last of the utilities we need. We now have enough code to comprise a simple text editor. Appendices C10 and C11 are listings of this text editor, showing key assignments that work on a Commodore 64 or VIC-20. If you prefer your editor functions mapped to different keys, simply change the values of the variables in the key table. If you don't want to have a given function, then for that function store a keycode of zero. You'll find this editor very handy for entering tables of ASCII characters into memory, and for entering, editing, and printing short text strings such as titles for your hexdumps and disassembler listings.

Chapter 12:

Extending the Visible Monitor

At this point you have the Visible Monitor, the print utilities, two hexdump tools, a table-driven disassembler, a move tool, and a simple text editor. Wouldn't it be nice if they were all combined into one interactive software package? Then you could call any tool or function with a single keystroke. Since the Visible Monitor already uses several keys (0 thru 9; A thru F; G; Space; Return; two arrow keys; and Clear-Screen), we'll have to map these new functions into unused keys.

Here's a list of keys and the functions they will have in the extended monitor:

H	Call a HEXDUMP tool (TVDUMP if the printer is not selected; PRDUMP if the printer is selected).
M	Call MOVER, the move tool.
P	Toggle the printer flag.
T	Call the text editor.
U	Toggle the user output flag.
?	Call the disassembler (TV.DIS if the printer is not selected; PR.DIS if the printer is selected).

With this assignment of keys to functions, we can select or deselect the *printer* at any time just by pressing "P," and likewise the *user*-driven output device just by pressing "U." We can print or display a *hexdump* just by pressing "H" and print or display a disassembly just by pressing "?" (which is almost mnemonic if we think of the disassembler as an answer to our question, "What's in the machine?"). We can move anything from anywhere to anywhere else by pressing "M" for *move*, and we can enter and edit text just by pressing "T" for *text editor.*

Here's some code to provide these features. Since we want to extend the monitor, this subroutine is called EXTEND:

EXTEND

```
                              When EXTEND is called by the Visible
                              Monitor's UPDATE routine, a character
                              from the keyboard is in the ac-
                              cumulator.
EXTEND   CMP #'P              Is it the "P" key?
         BNE IF.U             If not, perform the next test.
         LDA PRINTR           If so, toggle the
         EOR #$FF             printer flag...
         STA PRINTR
         RTS                  and return to caller.
IF.U     CMP #U               Is it the "U" key?
         BNE IF.H             If not, perform the next test.
         LDA USR.FN           If so,
         EOR #$FF             toggle the user-output
         STA USR.FN           flag...
         RTS                  and return.
IF.H     CMP #'H              Is it the "H" key?
         BNE IF.M             If not, perform the next test.
         LDA PRINTR           Is the printer selected?
         BNE NEXT.1           If so, print a hexdump.
         JSR TVDUMP           If not, dump to screen...
         RTS                  and return.
NEXT.1   JSR PRDUMP           Print a hexdump...
         RTS                  and return.
IF.M     CMP #'M              Is it the "M" key?
         BNE IF.DIS           If not, perform the next test.
         JSR MOVER            If so, call the move tool.
         RTS                  ...and return.
IF.DIS   CMP #'?              Is it the "?" key?
         BNE IF.T             If not, perform the next test.
         LDA PRINTR           Is the printer selected?
         BNE NEXT.2           If so, print a disassembly.
         JSR TV.DIS           If not, dump to screen...
         RTS                  and return.
NEXT.2   JSR PR.DIS           Print a disassembly...
         RTS                  and return.
IF.T     CMP #'T              Is it the "T" key?
         BNE EXIT             If not, return.
```

```
            JSR EDITOR      If so, call the text editor...
            RTS             and return.
EXIT        RTS             Extend this subroutine by adding more
                            test-and-branch code here.
```

The only remaining step is to modify the Visible Monitor's UPDATE routine so that it calls EXTEND, rather than DUMMY, before it returns. Currently, the Visible Monitor's UPDATE routine calls DUMMY just before it returns, with the bytes $20, $10, and $30 at addresses $33D1, $33D2, and $33D3, respectively. To make the Visible Monitor's UPDATE routine call EXTEND (instead of DUMMY), you must change $33D2 from $10 to $B0.

You can change this byte with the Visible Monitor itself, provided that you are very careful not to touch any key except the keys that are legal to the *un*extended Visible Monitor. Once you have changed $33D2, you may strike any key, but while you are changing $33D2, striking a key that is not legal within the unextended Visible Monitor will cause the Visible Monitor to crash. Be careful. Once you have changed $33D2, try out your new extensions of the Visible Monitor by pressing the now legal keys: "H," "M," "P," "U," "?," and "T."

Chapter 13:
Entering the Software into Your System

Chapters 5 thru 12 present software that will do useful work for you, but only if you can get it into your computer's memory. If your Commodore 64 or VIC-20 had a machine-language monitor, you could use it to enter the object code for the software in this book. (But if you already *had* such a monitor, you wouldn't have much need for the software in this book!)

Of course, your computer features a BASIC interpreter in ROM (read-only memory), but lacks a machine-language monitor. How can you enter hexadecimal object code into memory using only a BASIC interpreter? Perhaps more importantly, even if we manage to enter that object code into memory, how can we save that object code onto a cassette or disk? If all we have is a BASIC interpreter, the simplest solution is to make our object code look like a BASIC program.

That's not so hard. A BASIC program may contain DATA statements, so a simple BASIC program can contain a number of DATA statements, where the DATA statements actually represent, in decimal, the values of successive bytes in the object code. Then the BASIC program can READ those DATA statements and POKE the values it finds into the appropriate section of memory.

Using BASIC to Load Machine Language

The software in this book can be entered into your computer by RUNning just such a series of BASIC programs. Each of these programs consists of an OBJECT CODE LOADER followed by some number of DATA statements. The first two

DATA statements specify the range of DATA statements that follow. Each of the following DATA statements contains ten values: the first value is the start address at which object code from the line is to be loaded; the next eight values represent bytes to be loaded into memory, beginning at the specified address; and the tenth value is the checksum. The checksum is simply the total of the first nine values in the DATA statement. Of these ten values, the first and the tenth will always be greater than 4000, and the others will always be less than 256.

Appendices E1 through E11 contain this book's object code in the form of such DATA statements. You must type each of these DATA statements into your computer, but the BASIC OBJECT CODE LOADER is designed to let you know if you've made a mistake. It won't catch *any* possible error you might make while typing, but it will catch the most likely errors. How? The answer is in the checksum. If you make a mistake while typing in one of these DATA lines, the checksum will almost certainly fail to match the sum of the address and the 8 bytes in the line. Then, when you RUN the OBJECT CODE LOADER, it will identify the offending data statement by printing its line number as well as the address specified by the offending line.

The object code loader will use the following variables:

A	The address specified by a data line. Object code from that data line is to be loaded into memory beginning at that address.
BYTE	An array of DIMension 8, containing the values of 8 consecutive bytes of object code as specified by a data line.
CHECK	The checksum specified by a data line.
FIRST	The number of the first DATA statement containing object code.
LAST	The number of the last DATA statement containing object code.
LINE	A line counter, tracking the number of data lines of object code already loaded into memory.
SUM	The calculated sum of the 8 bytes of object code and the address specified by a given data line. If SUM equals the checksum specified by that data line, then the data is probably correct.
TEMP	A temporary variable.

NOTE: In the following listing, the REMarks are optional.

```
100 REM                    OBJECT CODE LOADER by Ken Skier
110 REM
120 DIM BYTE(8)            :REM Initialize BYTE array.
130 READ FIRST             :REM Get the line number of the first
140 REM                    DATA statement containing object code.
150 READ LAST              :REM Get the line number of the last
```

```
160 REM                          DATA statement containing object code.
170 FOR LINE=FIRST TO LAST       :REM Read the specified DATA lines.
180 GOSUB 300                    :REM Load next data line into memory.
190 NEXT LINE                    :REM If not done, read next DATA line.
200 PRINT "LOADED LINES",FIRST,"THROUGH",LAST,"SUCCESSFULLY."
210 END                          :REM If done, say so.
220 REM
230 REM                          Subroutine at 300 handles one
240 REM                          DATA statement.
300 READ A                       :REM Get address for object code.
310 SUM=A                        :REM Initialize calculated sum of data.
320 FOR J=1 TO 8                 :REM Get 8 bytes of object code from
321 REM                          data.
330 READ BYTE(J)                 :REM Put them in the byte array, and
340 SUM=SUM+BYTE(J)              :REM add them to the calculated sum of
341 REM                          data.
350 NEXT J                       :REM Now we have the 8 bytes, and we
360 REM                          have calculated the sum of the data.
370 READ CHECK                   :REM Get checksum from data line.
380 IF SUM < >CHECK THEN 500     :REM If checksum error, handle it.
390 FOR J=1 TO 8                 :REM Since there is no checksum error,
400 POKE A+J-1,BYTE(J)           :REM poke the data into the specified
410 NEXT J                       :REM portion of memory,
420 RETURN                       :REM and return to caller.
430 REM
440 REM                          Checksum error-handling code follows.
500 PRINT "CHECKSUM ERROR IN DATA LINE",LINE
510 PRINT "START ADDRESS GIVEN IN BAD DATA LINE IS", A
520 END
530 REM                          The next two DATA statements specify
540 REM                          the range of DATA statements that
550 REM                          contain object code.
570 REM
600 DATA ????                    :REM This should be the number of the
610 REM                          first DATA statement containing object
611 REM                          code.
612 REM
620 DATA ????                    :REM This should be the number of the
630 REM                          last DATA statement containing object
631 REM                          code.
```

Once you've entered the BASIC OBJECT CODE LOADER into your computer's memory, SAVE it on a cassette. Remember that by itself the BASIC

OBJECT CODE LOADER can do nothing; it needs DATA statements in the proper form to be a complete, useful program. When you're ready to create such a program, LOAD the BASIC OBJECT CODE LOADER from cassette back into memory. Now you're ready to append to it DATA statements from one of the E Appendices — for example, from Appendix E1. Do not append DATA statements from more than one appendix to the same BASIC program. Append as many DATA lines as you can, without using memory above $2FFF (decimal 12287). You can insure that you don't run over this limit by setting 12287 as the top of memory available to your system's BASIC interpreter. To do so, issue the following BASIC commands immediately after turning on your computer:

```
POKE 52,47 :  POKE 56,47
POKE 51,255:  POKE 55,255
```

That will keep BASIC from using memory above $2FFF.

Before you can append to the OBJECT CODE LOADER all of the DATA statements from a given E appendix, your BASIC interpreter may give you an OUT OF MEMORY error (MEMORY FULL). If that happens, delete the last DATA line you appended to the OBJECT CODE LOADER. Let's say you've appended DATA lines 1000 thru 1022 when you get an OUT OF MEMORY error. Delete DATA line 1022. Now enter the line numbers of the first and last of the object code DATA statements into DATA lines 600 and 620, like this:

```
600   DATA   1000
620   DATA   1021
```

DATA lines 600 and 620, the very first DATA lines in your program, tell the BASIC OBJECT CODE LOADER how many DATA lines of object code follow. Now the OBJECT CODE LOADER can "know" how many DATA lines to read, without reading too few or too many. In this case, DATA lines 600 and 620 tell the OBJECT CODE LOADER that the object code may be found in DATA lines 1000 thru 1021.

Note that DATA lines 600 and 620 each contain one value, whereas the remaining DATA lines each contain ten values.

Now you are ready to RUN the OBJECT CODE LOADER. Unless you're a better typist than I am, you probably made some mistakes while typing in the DATA lines from Appendix E1. Don't worry; the incorrect data will not be blindly loaded into memory. If the BASIC OBJECT CODE LOADER detects a checksum error, it will tell you so, like this:

```
CHECKSUM ERROR IN DATA STATEMENT            1012
START ADDRESS GIVEN IN BAD DATA LINE IS     12640
```

This means that data statement 1012 has a checksum error: ie, bad data. To help you double check, the second line of the error message specifies the start address given by the bad data line: this is the first number in the offending data line. These two items of information should make it easy for you to find the bad data line—just look for the DATA statement whose line number is 1012 and whose first value is 12640. That's the DATA statement you entered incorrectly. Now you need only eyeball the ten numbers in that line, comparing them to the corresponding DATA statement in Appendix E1, and you should quickly find the number or numbers you entered incorrectly. Fix that DATA statement, and RUN the LOADER again.

When you have entered all of the DATA statements correctly, RUNning the LOADER will load the object code they specify into memory. The OBJECT CODE LOADER will then print:

LOADED LINES aaaa THROUGH bbbb SUCCESSFULLY

where 'aaaa' is the number of the first DATA line of object code, and 'bbbb' is the number of the last DATA line of object code in the program. This message tells you that the BASIC OBJECT CODE LOADER has read and POKE'd the indicated range of DATA statements into memory.

When you see this message, you have verified the program, so SAVE it on a cassette. Then make up a new BASIC program, containing the OBJECT CODE LOADER and the next group of DATA statements from an E Appendix. (Remember not to append DATA lines from more than one E Appendix to the same BASIC program.) Store in lines 600 and 620 the line numbers of the first and last DATA statements you copied from the E Appendix. Verify and SAVE this program as well, and then continue in this manner until you have entered, verified, and SAVE'd BASIC programs containing all of the DATA statements in Appendices E1 thru E11, as well as the DATA statements in the E Appendix containing system data for your computer (E12 for the VIC; E13 for the C-64). RUNning all of those BASIC programs will then enter all of the software presented in this book into your computer's memory.

At this point, you should be ready to transfer control from your computer's BASIC interpreter to the VISIBLE MONITOR.

Activating the Visible Monitor

Once you have entered the object code for the Screen Utilities, the Visible Monitor, and the System Data Block into your system, you can activate the Visible Monitor by causing the 6502 in your computer to execute a JSR (jump to subroutine) to $308F, which is 12431 decimal.

You can invoke the Visible Monitor from BASIC in the immediate mode with the following BASIC command:

SYS (12431)

When you press (RETURN), you'll see the Visible Monitor display, because SYS (12431) causes BASIC to call the subroutine at address 12431 decimal, which is \$308F—the entry point for the Visible Monitor.

Once you have activated the Visible Monitor, you should see its display on the screen. If you don't see such a display, then the Visible Monitor has not been entered properly into your system's memory; perhaps you LOADed one of the BASIC programs whose DATA statements contain object code, but you forgot to RUN it.

If you do see the Visible Monitor display on the screen, press the space bar. The display should change — specifically, the displayed address should increment, and fields 1 and 2, immediately to the right of the displayed address, may also change.

If nothing changes when you press the space bar, then the display code probably works fine, but you failed to enter the UPDATE code properly.

If the space bar does change the display, then test out the other functions of the Visible Monitor: press RETURN to decrement the selected address; press hexadecimal keys to select a different address; then select an address somewhere in unused RAM (e.g., \$03FD) and place new data into that address.

If your Visible Monitor fails to perform properly, you may have entered it into memory incorrectly. Compare the DATA statements you appended to the OBJECT CODE LOADER with the DATA statements in the E Appendices. Remember: if even 1 byte is entered incorrectly, then in all likelihood the Visible Monitor will fail to function. Remember that you must LOAD and RUN BASIC programs containing, jointly *all* of the DATA statements in appendices E1 thru E11, plus all the DATA statements in appendix E12 (if you have a VIC) or appendix E13 (if you have a Commodore 64). Do not try to call the Visible Monitor until you have entered into memory all of the object code it uses—including the system data block designed for your system. Invoking the Visible Monitor too soon will surely cause it to crash.

To extend the Visible Monitor as described in Chapter 12, store a \$BO in address \$33D2. To disable the features described in Chapter 12, store a \$10 in address \$33D2. Now you're really getting your hands on the machine, reaching into memory and operating on the bytes, and with that kind of control, you can do almost anything.

Saving a Machine Language Program on Tape or Disk

With the Visible Monitor and its extensions, you can create ever more powerful machine language (ML) programs. Presumably, you will want to save them on tape or disk, so that you can load them again and run them whenever you wish. Fortunately, the Commodore 64 and VIC-20 computers contain a set of subroutines in ROM, called the KERNAL routines, which include the subroutines you need to save object code on tape and disk.

We'll assume that the Visible Monitor and its extensions is in memory, as well as some machine language program that you have entered. To save that program on tape or disk, run the following BASIC program:

BASIC PROGRAM TO SAVE ANY MACHINE LANGUAGE ON TAPE OR DISK

```
10   DEVICE =12364
20   LNGTH  =12365
30   NAME   =12366
40   MLSAV  =12386
50   SETADS =13795
60   :
100  PRINT "SAVE A MACHINE LANGUAGE PROGRAM"
110  PRINT
120  INPUT "FILE NAME";NAME$
125  IF LEN(NAME$)>19 THEN NAME$=LEFT$(NAME$,19)
130  POKE LNGTH,LEN(NAME$)
140  IF LEN(NAME$)=0 THEN 200
150  :
160  FOR J=1 TO LEN(NAME$)
170  :  POKE NAME+J-1,ASC(MID$(NAME$,J))
180  NEXT J
190  :
200  PRINT "SAVE TO (T)APE OR (D)ISK?"
210  GET A$:IF LEN(A$)=0 THEN 210
220  IF A$="T" THEN POKE DEVICE,1:GOTO 300
230  IF A$="D" THEN POKE DEVICE,8:GOTO 300
240  PRINT "KEYSTROKE IGNORED.":PRINT:GOTO 200
250  :
300  SYS (SETADS)  :  REM GET START, END ADDRESSES
310  SYS (MLSAV)   :  REM SAVE THE PROGRAM ON DISK OR TAPE.
```

The above BASIC program will ask you to specify a filename; then it will store that filename at an address in memory called NAME. It will ask you to specify the device you wish to use (tape or disk) and then it will ask you to specify the start and end addresses of the machine language program. When you have done so, it will create a file on the tape or disk, save the specified portion of memory in that file, and then return to BASIC. You'll know it's done when you see the "READY" prompt on the screen.

Once you have saved a machine language on tape or disk, you may load it in again at any time. If you have saved the file on tape, enter this BASIC command:

```
LOAD "program name",1,1
```

On the other hand, if you saved the machine language program on disk, enter this BASIC command:

```
LOAD "program name" ",8,1
```

After LOADing a machine language program, some pointers used by BASIC may be inaccurate, which will cause it to respond with "OUT OF MEMORY" to many perfectly legitimate commands. To fix these pointers, be sure to issue the following BASIC commands immediately after using the LOAD command to load a machine language program:

```
POKE 52,47 :  POKE 56,47
POKE 51,255:  POKE 55,255
NEW
```

Issuing these commands will delete the BASIC program in memory (if there is one); but it will not affect any machine language program in memory above $2FFF. In any case, it will correct the BASIC pointers that were incorrectly modified by LOADING the machine language program. You may then load a BASIC program, or enter one from the keyboard, without damaging the machine language program that you just loaded.

Appendix A1:

Hexadecimal Conversion Table

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	158	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

Appendix A2:

ASCII Character Codes

Code	Char	Code	Char	Code	Char	Code	Char
00	NUL	20	SP	40	@	60	`
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	–	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	¦
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

Appendix A3:

6502 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
BML	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 Register X
CPY	Compare Memory and Register Y
DEC	Decrement Memory
DEX	Decrement Register X
DEY	Decrement Register Y
EOR	"Exclusive Or" Memory with Accumulator
INC	Increment Memory
INX	Increment Register X
INY	Increment Register Y

JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Register X with Memory
LDY	Load Register 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 Register X in Memory
STY	Store Register Y in Memory
TAX	Transfer Accumulator to Register X
TAY	Transfer Accumulator to Register Y
TSX	Transfer Stack Pointer to Register X
TXA	Transfer Register X to Accumulator
TXS	Transfer Register X to Stack Pointer
TYA	Transfer Register Y to Accumulator

Appendix A4:

6502 Instruction Set — Opcode List

00 — BRK
01 — ORA — (Indirect,X)
02 — Future Expansion
03 — Future Expansion
04 — Future Expansion
05 — ORA — Zero Page
06 — ASL — Zero Page
07 — Future Expansion
08 — PHP
09 — ORA — Immediate
0A — ASL — Accumulator
0B — Future Expansion
0C — Future Expansion
0D — ORA — Absolute
0E — ASL — Absolute
0F — Future Expansion

10 — BPL
11 — ORA — (Indirect),Y
12 — Future Expansion
13 — Future Expansion
14 — Future Expansion
15 — ORA — Zero Page,X
16 — ASL — Zero Page,X
17 — Future Expansion
18 — CLC
19 — ORA — Absolute,Y
1A — Future Expansion
1B — Future Expansion
1C — Future Expansion
1D — ORA — Absolute, X
1E — Future Expansion
1F — Future Expansion

20 — JSR
21 — AND — (Indirect,X)
22 — Future Expansion
23 — Future Expansion
24 — Bit — Zero Page
25 — AND — Zero Page
26 — ROL — Zero Page
27 — Future Expansion
28 — PLP
29 — AND — Immediate
2A — ROL — Accumulator
2B — Future Expansion
2C — BIT — Absolute
2D — AND — Absolute
2E — ROL — Absolute
2F — Future Expansion

30 — BMI
31 — AND — (Indirect),Y
32 — Future Expansion
33 — Future Expansion
34 — Future Expansion
35 — AND — Zero Page,X
36 — ROL — Zero Page,X
37 — Future Expansion
38 — SEC
39 — AND — Absolute,Y
3A — Future Expansion
3B — Future Expansion
3C — Future Expansion
3D — AND — Absolute,X
3F — Future Expansion

40 — RTI
41 — EOR — (Indirect,X)
42 — Future Expansion
43 — Future Expansion
44 — Future Expansion
45 — EOR — Zero Page
46 — LSR — Zero Page
47 — Future Expansion
48 — PHA
49 — EOR — Immediate
4A — LSR — Accumulator
4B — Future Expansion
4C — JMP — Absolute
4D — EOR — Absolute
4E — LSR — Absolute
4F — Future Expansion

50 — BVC
51 — EOR — (Indirect),Y
52 — Future Expansion
53 — Future Expansion
54 — Future Expansion
55 — EOR — Zero Page,X
56 — Zero Page,X
57 — Future Expansion
58 — CLI
59 — EOR — Absolute,Y
5A — Future Expansion
5B — Future Expansion
5C — Future Expansion
5D — EOR — Absolute,X
5E — LSR — Absolute,X
5F — Future Expansion

60 — RTS
61 — ADC — (Indirect,X)
62 — Future Expansion
63 — Future Expansion
64 — Future Expansion
65 — ADC — Zero Page
66 — ROR — Zero Page
57 — Future Expansion
68 — PLA
69 — ADC — Immediate
6A — ROR — Accumulator
6B — Future Expansion
6C — JMP — Indirect
6D — ADC — Absolute
6E — ROR — Absolute
6F — Future Expansion

70 — BVS
71 — ADC — (Indirect),Y
72 — Future Expansion
73 — Future Expansion
74 — Future Expansion
75 — ADC — Zero Page,X
76 — ROR — Zero Page,X
77 — Future Expansion
78 — SEI
79 — ADC Absolute,Y
7A — Future Expansion
7B — Future Expansion
7C — Future Expansion
7D — ADC — Absolute,X
7E — ROR — Absolute,X
7F — Future Expansion

80 — Future Expansion
81 — STA — (Indirect,X)
82 — Future Expansion
83 — Future Expansion
84 — STY — Zero Page
85 — STA — Zero Page
86 — STX — Zero Page
87 — Future Expansion
88 — DEY
89 — Future Expansion
8A — TXA
8B — Future Expansion
8C — STY — Absolute
8D — STA — Absolute
8E — STX — Absolute
8F — Future Expansion

90 — BCC
91 — STA — (Indirect),Y
92 — Future Expansion
93 — Future Expansion
94 — STY — Zero Page,X
95 — STA — Zero Page,X
96 — STX — Zero Page,Y
97 — Future Expansion
98 — TYA
99 — STA — Absolute,Y
9A — TXS
9B — Future Expansion
9C — Future Expansion
9D — STA — Absolute,X
9E — Future Expansion
9F — Future Expansion

A0 — LDY — Immediate
A1 — LDA — (Indirect,X)
A2 — LDX — Immediate
A3 — Future Expansion
A4 — LDY — Zero Page
A5 — LDA — Zero Page
A6 — LDX — Zero Page
A7 — Future Expansion
A8 — TAY
A9 — LDA — Immediate
AA — TAX
AB — Future Expansion
AC — LDY — Absolute
AD — LDA — Absolute
AE — LDX — Absolute
AF — Future Expansion

B0 — BCS
B1 — LDA — (Indirect),Y
B2 — Future Expansion
B3 — Future Expansion
B4 — LDY — Zero Page,X
B5 — LDA — Zero Page,X
B6 — LDX — Zero Page,Y
B7 — Future Expansion
B8 — CLV
B9 — LDA — Absolute,Y
BA — TSX
BB — Future Expansion
BC — LDY — Absolute,X
BD — LDA — Absolute,X
BE — LDX — Absolute,Y
BF — Future Expansion

C0 — CPY — Immediate
C1 — CMP — (Indirect,X)
C2 — Future Expansion
C3 — Future Expansion
C4 — CPY — Zero Page
C5 — CMP — Zero Page
C6 — DEC — Zero Page
C7 — Future Expansion
C8 — INY
C9 — CMP — Immediate
CA — DEX
CB — Future Expansion
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 — SEC — (Indirect,X)
E2 — Future Expansion
E3 — Future Expansion
E4 — CPX — Zero Page
E5 — SBC — Zero Page
E6 — 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 A5:

Instruction Execution Times (in clock cycles)

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect), X	(Indirect), Y	Absolute Indirect
ADC	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
AND	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
ASL	2	.	5	6	.	6	7	.	.	.	.	.	.
BCC	.	.	.	.	.	.	.	.	.	2**	.	.	.
BCS	.	.	.	.	.	.	.	.	.	2**	.	.	.
BEQ	.	.	.	.	.	.	.	.	.	2**	.	.	.
BIT	.	.	3	.	.	4	.	.	.	.	.	.	.
BMI	.	.	.	.	.	.	.	.	.	2**	.	.	.
BNE	.	.	.	.	.	.	.	.	.	2**	.	.	.
BPL	.	.	.	.	.	.	.	.	.	2**	.	.	.
BRK	.	.	.	.	.	.	.	.	.	.	.	.	.
BVC	.	.	.	.	.	.	.	.	.	2**	.	.	.
BVS	.	.	.	.	.	.	.	.	.	2**	.	.	.
CLC	.	.	.	.	.	.	.	.	2	.	.	.	.
CLD	.	.	.	.	.	.	.	.	2	.	.	.	.
CLI	.	.	.	.	.	.	.	.	2	.	.	.	.
CLV	.	.	.	.	.	.	.	.	2	.	.	.	.
CMP	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
CPX	.	2	3	.	.	4	.	.	.	.	.	.	.
CPY	.	2	3	.	.	4	.	.	.	.	.	.	.
DEC	.	.	5	6	.	6	7	.	.	.	.	.	.
DEX	.	.	.	.	.	.	.	.	2	.	.	.	.
DEY	.	.	.	.	.	.	.	.	2	.	.	.	.
EOR	.	2	3	4	.	4	4*	4*	.	.	6	5	.

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect), X	(Indirect), Y	Absolute Indirect
INC	.	.	5	6	.	6	7	.	.	.	.	.	.
INX	.	.	.	.	.	.	.	.	2	.	.	.	.
INY	.	.	.	.	.	.	.	.	2	.	.	.	.
JMP	.	.	.	.	.	3	.	.	.	.	.	.	5
JSR	.	.	.	.	.	6	.	.	.	.	.	.	.
LDA	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
LDX	.	2	3	.	4	4	.	4*	.	.	.	.	.
LDY	.	2	3	4	.	4	4*	.	.	.	.	.	.
LSR	2	.	5	6	.	6	7	.	.	.	.	.	.
NOP	.	.	.	.	.	.	.	.	2	.	.	.	.
ORA	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
PHA	.	.	.	.	.	.	.	.	3	.	.	.	.
PHP	.	.	.	.	.	.	.	.	3	.	.	.	.
PLA	.	.	.	.	.	.	.	.	4	.	.	.	.
PLP	.	.	.	.	.	.	.	.	4	.	.	.	.
ROL	2	.	5	6	.	6	7	.	.	.	.	.	.
ROR	2	.	5	6	.	6	7	.	.	.	.	.	.
RTI	.	.	.	.	.	.	.	.	6	.	.	.	.
RTS	.	.	.	.	.	.	.	.	6	.	.	.	.
SBC	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
SEC	.	.	.	.	.	.	.	.	2	.	.	.	.
SED	.	.	.	.	.	.	.	.	2	.	.	.	.
SEI	.	.	.	.	.	.	.	.	.	.	.	.	.
STA	.	.	3	4	.	4	5	5	.	.	6	6	.
STX*	.	.	3	.	4	4	.	.	.	.	.	.	.
STY**	.	.	3	4	.	4	.	.	.	.	.	.	.
TAX	.	.	.	.	.	.	.	.	2	.	.	.	.
TAY	.	.	.	.	.	.	.	.	2	.	.	.	.
TSX	.	.	.	.	.	.	.	.	2	.	.	.	.
TXA	.	.	.	.	.	.	.	.	2	.	.	.	.
TXS	.	.	.	.	.	.	.	.	2	.	.	.	.
TYA	.	.	.	.	.	.	.	.	2	.	.	.	.

* Add one cycle if indexing across page boundary

** Add one cycle if branch is taken, Add one additional if branching operation crosses page boundary

Appendix A6:

6502 Opcodes by Mnemonic and Addressing Mode

Mnemonics	Addressing Modes												
	ABSOLUTE	ABSOLUTE,X	ABSOLUTE,Y	ACCUMULATOR	IMMEDIATE	IMPLIED	INDIRECT	INDIRECT,X	INDIRECT,Y	RELATIVE	ZERO PAGE	ZERO PAGE,X	ZERO PAGE,Y
ADC	6D	7D	79	.	69	.	.	61	71	.	65	75	.
AND	2D	3D	39	.	29	.	.	21	31	.	25	35	.
ASL	0E	1E	.	0A	.	.	.	.	.	.	06	16	.
BCC	.	.	.	.	.	.	.	.	.	90	.	.	.
BCS	.	.	.	.	.	.	.	.	.	B0	.	.	.
BEQ	.	.	.	.	.	.	.	.	.	F0	.	.	.
BIT	2C	.	.	.	.	.	.	.	.	.	24	.	.
BMI	.	.	.	.	.	.	.	.	.	30	.	.	.
BNE	.	.	.	.	.	.	.	.	.	D0	.	.	.
BPL	.	.	.	.	.	.	.	.	.	10	.	.	.
BRK	.	.	.	.	.	00	.	.	.	.	.	.	.
BVC	.	.	.	.	.	.	.	.	.	50	.	.	.
BVS	.	.	.	.	.	.	.	.	.	70	.	.	.
CLC	.	.	.	.	.	18	.	.	.	.	.	.	.
CLD	.	.	.	.	.	D8	.	.	.	.	.	.	.
CLI	.	.	.	.	.	58	.	.	.	.	.	.	.

Mnemonics	ABSOLUTE	ABSOLUTE,X	ABSOLUTE,Y	ACCUMULATOR	IMMEDIATE	IMPLIED	INDIRECT	INDIRECT,X	INDIRECT,Y	RELATIVE	ZERO PAGE	ZERO PAGE,X	ZERO PAGE,Y
CLV	.	.	.	.	.	B8	.	.	.	.	.	.	.
CMP	CD	DD	D9	.	C9	.	.	C1	D1	.	C5	D5	.
CPX	EC	.	.	.	E0	.	.	.	.	.	E4	.	.
CPY	CC	.	.	.	C0	.	.	.	.	.	C4	.	.
DEC	CE	DE	.	.	.	.	.	.	.	.	C6	D6	.
DEX	.	.	.	.	.	CA	.	.	.	.	.	.	.
DEY	.	.	.	.	.	88	.	.	.	.	.	.	.
EOR	4D	5D	59	.	49	.	.	41	51	.	45	55	.
INC	EE	FE	.	.	.	.	.	.	.	.	E6	F6	.
INX	.	.	.	.	.	E8	.	.	.	.	.	.	.
INY	.	.	.	.	.	C8	.	.	.	.	.	.	.
JMP	4C	.	.	.	.	.	6C	.	.	.	.	.	.
JSR	20	.	.	.	.	.	.	.	.	.	.	.	.
LDA	AD	BD	B9	.	A9	.	.	A1	B1	.	A5	B5	.
LDX	AE	.	BE	.	A2	.	.	.	.	.	A6	.	.
LDY	AC	BC	.	.	A0	.	.	.	.	.	A4	B4	.
LSR	4E	5E	.	4A	.	.	.	.	.	.	46	56	.
NOP	.	.	.	.	.	EA	.	.	.	.	.	.	.
ORA	0D	1D	19	.	09	.	.	01	11	.	05	15	.
PHA	.	.	.	.	.	48	.	.	.	.	.	.	.
PHP	.	.	.	.	.	08	.	.	.	.	.	.	.
PLA	.	.	.	.	.	68	.	.	.	.	.	.	.
PLP	.	.	.	.	.	28	.	.	.	.	.	.	.
ROL	2E	3E	.	2A	.	.	.	.	.	.	26	36	.
ROR	6E	7E	.	6A	.	.	.	.	.	.	66	76	.
RTI	.	.	.	.	.	40	.	.	.	.	.	.	.

	Addressing Modes												
	ABSOLUTE	ABSOLUTE,X	ABSOLUTE,Y	ACCUMULATOR	IMMEDIATE	IMPLIED	INDIRECT	INDIRECT,X	INDIRECT,Y	RELATIVE	ZERO PAGE	ZERO PAGE,X	ZERO PAGE,Y
Mnemonics													
RTS	.	.	.	.	.	60	.	.	.	.	.	.	.
SBC	ED	FD	F9	.	E9	.	.	E1	F1	.	E5	F5	.
SEC	.	.	.	.	.	38	.	.	.	.	.	.	.
SED	.	.	.	.	.	F8	.	.	.	.	.	.	.
SEI	.	.	.	.	.	78	.	.	.	.	.	.	.
STA	8D	9D	99	.	.	.	.	81	91	.	85	95	.
STX	8E	.	.	.	.	.	.	.	.	.	86	.	.
STY	8C	.	.	.	.	.	.	.	.	.	84	94	.
TAX	.	.	.	.	.	AA	.	.	.	.	.	.	.
TAY	.	.	.	.	.	A8	.	.	.	.	.	.	.
TSX	.	.	.	.	.	BA	.	.	.	.	.	.	.
TXA	.	.	.	.	.	8A	.	.	.	.	.	.	.
TXS	.	.	.	.	.	9A	.	.	.	.	.	.	.
TYA	.	.	.	.	.	98	.	.	.	.	.	.	.

Appendix B1:

The VIC-20

The Commodore VIC-20 is a very sophisticated, yet inexpensive, small computer. Unfortunately, the least expensive version of the VIC-20 has too little RAM to run the software presented in this book. The unexpanded VIC-20 contains only 3 K of RAM. To run the software in this book, your VIC must have at least 8 K of expansion RAM, beginning at $2000. You may add this extra RAM to your VIC by installing the Commodore VIC-1110 8 K Memory Expander or the VIC-1111 16 K memory expander. Other manufacturers may also provide appropriate expansion RAM.

If your VIC has at least 8 K of RAM, beginning at $2000, then its screen memory is located at $1000. (By contrast, the screen on an unexpanded VIC is located at $1E00.) The screen contains 25 rows, each consisting of 22 characters. The address of each screen location is 22 ($16) greater than the address of the location directly above it. Thus, the screen parameters for the VIC-20 are:

```
HOME      .WORD $1000
ROWINC    .BYTE  22      Address difference from
                         one row to the next.
TVCOLS    .BYTE  21      (We count columns from zero.)
TVROWS    .BYTE  24      (We count rows from zero.)
```

Is this all we need to know about the VIC's screen? Nope. With some computers, you can display any desired character on the screen by simply storing its ASCII code in a given screen location. But displaying ASCII characters on the VIC screen is a little more difficult. First, there is the matter of *color memory*. Then there's the problem of determining the proper VIC *screen code* for the character you wish to display. So let's examine each of these issues.

Color Memory

In addition to its screen memory, the VIC contains something called color memory. Every byte in screen memory has a corresponding byte in color memory. As you know, screen memory tells the VIC's display circuitry what characters to display on screen. But what does color memory do? Not surprisingly, it specifies the *colors* for those characters.

In a VIC with at least 8 K of expansion RAM, color memory is located at $9400, which is $8400 above screen memory. The n'th byte in color memory specifies the color for the n'th character in screen memory.

Note that if a portion of color memory contains the code for the screen's *background* color, then no characters will be visible in the corresponding portion of the screen—even if the corresponding portion of screen memory contains text. So before we try to display a character on the screen, we must store an appropriate (non-background) color code in the proper byte of color memory.

What color code will we use? We could select an arbitrary color—for example, black—but it makes more sense to use the color already selected by the user. Address $286 (646 decimal) contains the color code selected by the user, which is used by the VIC when it prints on the screen. So before we store any character in screen memory, we'll get the color code in address $286 and store it in the proper byte of color memory. That byte will be exactly $8400 above the byte pointed to by TV.PTR.

So now we know how to set color memory when we wish to display text on the screen. But how do we determine the proper *screen code* to use?

VIC Screen Codes

To display a given character on the screen, we must store the appropriate screen code in screen memory. Table B1.1 shows the VIC screen codes.

In Table B1.1, special graphic characters are indicated by an underline. To see those special graphics in all their glorious detail, enter the following BASIC program into your VIC and run it:

```
100 REM   DISPLAY VIC SCREEN CODES
110 REM   IN 16 BY 16 MATRIX
120 REM
130 PRINT CHR$(147)  :  REM CLEAR SCREEN
140 SCREEN=4096   :  REM SCREEN MEMORY
150 CMEM=37888    :  REM COLOR MEMORY
160 C=PEEK(646)   :  REM CURRENT COLOR
170 REM
```

```
180 FOR ROW=0 TO 15
190 FOR COL=0 TO 15
200 POKE CMEM+COL+22*ROW,C
210 POKE SCREEN+COL+22*ROW,COL+16*ROW
220 NEXT COL,ROW
230 REM
240 GOTO 240
```

This program clears the screen, pokes the current color code into the appropriate bytes of color memory, and pokes all 256 screen codes into a 16 by 16 grid of screen memory. When it has done so, it will sit in an endless loop in line 240. To break it out of this loop, press the RUN/STOP key.

FIXCHR

As you can see, the VIC's screen codes require a translation from ASCII. However, FIXCHR must do more than simply convert an ASCII code to a VIC screen code. It must also store the current color code in the appropriate byte of color memory. The following source code for FIXCHR will accomplish both of these tasks:

Table B1.1: *The VIC character set.*

LEFT NYBBLE OF CHARACTER / RIGHT NYBBLE OF CHARACTER	–0	–1	–2	–3	–4	–5	–6	–7	–8	–9	–A	–B	–C	–D	–E	–F
0–	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1–	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
2–		!	"	#	$	%	&	'	(	)	*	+	'	—	.	/
3–	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4–	—	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
5–	p	q	r	s	t	u	v	w	x	y	z	—	—	—	—	—
6–	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
7–	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
8–	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9–	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
A–		!	"	#	$	%	&	'	(	)	*	+	'	—	.	/
B–	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
C–	—	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
D–	p	q	r	s	t	u	v	w	x	y	z	—	—	—	—	—
E–	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
F–	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

These characters are in reverse video. (rows 8– to D–)

```
FIXCHR                        A character is in A. We
                              must convert it to proper
                              VIC screen code.

                              But first, put a color code
                              in appropriate byte of
                              color memory. (Otherwise,
                              that byte in color memory
                              might hold the background
                              color code, rendering the
                              character invisible.)
          PHA                 Save character to be
                              displayed.
          LDA TV.PTR+1        Save high byte...
          PHA                 ...of TV.PTR.
          CLC                 Make TV.PTR point
          ADC #$84            to appropriate byte
          STA TV.PTR+1        of color memory.
          LDY #0
          LDA $286            Get current color code.
                              Store it in appropriate
                              byte of color memory:
          STA (TV.PTR),Y
          PLA                 Restore high byte of TV.PR
          STA TV.PTR+1        to its original value.

          PLA                 Retrieve character to be
                              displayed.
          SEC                 Prepare to compare.
          CMP #$40            Is it less than $40? (Is
                              it a number or a punctuation
                              mark?)
          BCC FIXEND          If so, no conversion needed.
          CMP #$60            Is it in the range $40...$5F?
          BCC SUB.40          If so, subtract $40 to
                              convert from ASCII to
                              VIC screen code.
                              OK. It's greater than $5F.
          SBC #$20            Subtract $20 to convert
                              lower case ASCII to VIC.
          RTS                 and return.

SUB.40    SEC                 Prepare to subtract.
          SBC #$40            Subtract $40 to convert ASCII
```

```
                                 uppercase char to VIC code.
FIXEND         RTS               Return, with A holding
                                 VIC screen code for ASCII
                                 char originally in A.
```

VIC Keyboard Input Routine

To get an ASCII character from the VIC keyboard, call the following subroutine:

```
VICKEY         JSR $FFE4         Call VIC ROM key scan routine.
               CMP #0            Zero means no key.
               BEQ VICKEY        If no key, scan again.
               RTS               Return with ASCII character
                                 from the keyboard.
```

This subroutine yields the uppercase ASCII code for any letter key that you depress, and the proper ASCII code for any digit key or punctuation key.

VIC TVT Routine

To print an ASCII character to the screen, call $FFD2, a VIC ROM routine I will refer to as VICTVT.

Any printable ASCII character passed to $FFD2 will be printed properly to the screen at the VIC's current TVT screen location. You may change the VIC's current TVT screen location (which is *not* the same as the current location used by the screen utilities in Chapter 5) by calling VICTVT with the accumulator holding any of the control codes from Table B1.2.

These control codes may be passed directly to VICTVT, or they may be included within a string of characters to be printed by "PRINT:" or "PR.MSG." For example, if you wish to clear the screen before printing a message, just put the CLEAR character ($93) at the beginning of your message string, immediately following the STX. The message-printing subroutine will get the CLEAR character and pass it to PR.CHR, which, in turn, will pass it through the ROMTVT vector on to

Table B1.2: *Control codes that affect the next character to be printed by VICTVT.*

Character Name	Code	Function
CURSOR NORTH	$91	Move current location up by one row.
CURSOR EAST	$1D	Move current location one column to the right.
CURSOR SOUTH	$11	Move current location down by one row.
CURSOR WEST	$9D	Move current location left by one column.
INSERT	$94	Move current character, and all characters to its right, one column to the right.
DELETE	$14	Move current character, and all characters to its right, one column to the left.
HOME	$13	Set current location to upper left of screen.
CLEAR	$93	Set current location to the upper left corner and clear the screen.
REVERSE	$12	Select reverse video for following characters.
REVERSE-OFF	$92	Select normal video mode for following characters.

the VICTVT routine. The VICTVT routine will then clear the screen and set the current location to the upper left corner of the screen.

The next character in the string will then be printed in the upper left corner of a clear screen. If, instead of printing your message at the top row of a clear screen, you'd prefer to print it in the fifth row of a clear screen, just follow the CLEAR character with four CURSOR-SOUTH characters ($11, $11, $11, $11), and follow the four cursor-south characters with the text of your message. Following the text of your message, of course, you must include an ETX ($FF).

You might never use the VICTVT control codes, but it's good to know they're available, should you ever want your VIC's display screen to perform as something more than a glass teletype.

Setting the Top of Memory

Before you can load the Visible Monitor or its extensions into your VIC, you must ensure that your VIC's BASIC interpreter won't use memory above $2FFF. To do so, type the following lines into your VIC *immediately after you have turned it on*:

```
POKE 52,47 :POKE 56,47
POKE 51,255:POKE 55,255
NEW
```

(Be sure to press RETURN after typing each line.)

Now you may enter and run BASIC programs, without disturbing the memory used by the software in this book. Remember: you *must* set the top of memory, as shown above, each time you load the Visible Monitor and its extensions into your VIC. (See chapter 13.)

Invoking the Visible Monitor from BASIC

Once you have loaded the Visible Monitor (using the BASIC Object Code Loader and the "E" series of appendices), you can activate the Visible Monitor from BASIC with the following BASIC command:

```
SYS 12431
```

You may then return from the Visible Monitor to BASIC by pressing "Q" ("Q" for Quit).

Getting Hard Copy

The Printing Hexdump program, the Printing Disassembler, and the Simple Text Editor all direct their output to a printer. Actually, that's not quite true; they direct their output to any device you designate as logical file #2, be it a printer, a disk file, the modem, or whatever.

If you wish to output text and data from the software in this book to any device, you must first OPEN that device as *logical file* #2. The easiest way to do that is in a BASIC program, prior to activating the Visible Monitor. For example, here's a BASIC program that opens a 1200-baud channel on the RS-232 port, and then transfers control to the Visible Monitor:

```
100 OPEN 2,2,0,CHR$(8)
110 SYS 12431
120 CLOSE 2
```

Line 100 opens the RS-232 port as logical file #2 (configuring it to operate at 1200 baud). Then line 110 passes control to the BASIC ENTRY point of the Visible Monitor. (See appendix C13.) From the Visible Monitor, you may then select any of the software that prints, and it will direct its output to the RS-232 port. If you have a printer connected to that port, you will then get a hard copy of the hexdump, disassembly, or text.

If you replace line 100 in the above BASIC program with a line that opens a DISK file as logical file #2, then you'll direct the hexdump or disassembly to a disk file. Or replace line 100 with a line that opens the Commodore printer as device #2, and you'll send the hexdump or disassembly to the Commodore printer. Thus, you can send the hardcopy output to any device you desire, simply by opening that device as logical file #2 before invoking the Visible Monitor through its BASIC entry point (at 12431).

NOTE: If you don't open a device or file as logical device #2, then attempting to use any of the printing software in this book will cause it to print every character *twice* on the screen. So if you notice that your VIC has developed a ssttuutteerr when you try to print a hexdump or disassembly, the cure is simple: just exit to BASIC and open the desired output device as logical file #2 before re-entering the Visible Monitor.

Appendix B2:

The Commodore 64

The Commodore 64 is a very sophisticated, yet inexpensive, small computer. Unlike many other computers in its price range, it features 64 K of RAM, so it has more than enough RAM to run the software presented in this book.

The C-64's screen memory is normally located at $400. (As the *Commodore 64 Programmer's Reference Guide* shows, it is possible to locate the screen elsewhere in memory, but we will assume that you have not done so.) The screen contains 25 rows, each consisting of 40 characters. The address of each screen location is 40 ($28) greater than the address of the location directly above it. Thus, the screen parameters for the C-64 are:

```
HOME      .WORD $400
ROWINC    .BYTE  40      Address difference from
                         one row to the next.
TVCOLS    .BYTE  39      (We count columns from zero.)
TVROWS    .BYTE  24      (We count rows from zero.)
```

Is this all we need to know about the C-64's screen? Nope. With some computers, you can display any desired character on the screen by simply storing its ASCII code in a given screen location. But displaying ASCII characters on the C-64 screen is a little more difficult. First, there is the matter of *color memory*. Then there's the problem of determining the proper C-64 *screen code* for the character you wish to display. So let's examine each of these issues.

Color Memory

In addition to its screen memory, the C-64 contains something called color memory. Every byte in screen memory has a corresponding byte in color memory. As you know, screen memory tells the C-64's display circuitry what characters to display on screen. But what does color memory do? Not surprisingly, it specifies the *colors* for those characters.

The C-64's color memory is located at $D800, which is $D400 above screen memory. The n'th byte in color memory specifies the color for the n'th character in screen memory.

Note that if a portion of color memory contains the code for the screen's *background* color, then no characters will be visible in the corresponding portion of the screen—even if the corresponding portion of screen memory contains text. So before we try to display a character on the screen, we must store an appropriate (non-background) color code in the proper byte of color memory.

What color code will we use? We could select an arbitrary color—for example, black—but it makes more sense to use the color already selected by the user. Address $286 (646 decimal) contains the color code selected by the user, which is used by the C-64 when it prints on the screen. So before we store any character in screen memory, we'll get the color code in address $286 and store it in the proper byte of color memory. That byte will be exactly $D400 above the byte pointed to by TV.PTR.

So now we know how to set color memory when we wish to display text on the screen. But how do we determine the proper *screen code* to use?

C-64 Screen Codes

To display a given character on the screen, we must store the appropriate screen code in screen memory. Table B2.1 shows the C-64 screen codes.

Table B2.1: *The C-64 character set.*

	RIGHT NYBBLE OF CHARACTER															
LEFT NYBBLE OF CHARACTER	−0	−1	−2	−3	−4	−5	−6	−7	−8	−9	−A	−B	−C	−D	−E	−F
0−	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1−	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
2−		!	"	#	$	%	&	'	(	)	*	+	,	−	.	/
3−	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4−	—	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
5−	p	q	r	s	t	u	v	w	x	y	z	—	—	—	—	—
6−	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
7−	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
8−	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9−	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
A−		!	"	#	$	%	&	'	(	)	*	+	,	−	.	/
B−	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
C−	—	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
D−	p	q	r	s	t	u	v	w	x	y	z	—	—	—	—	—
E−	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
F−	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

These characters are in reverse video.

In Table B2.1, special graphic characters are indicated by an underline. To see those special graphics in all their glorious detail, enter the following BASIC program into your C-64 and run it:

```
100 REM   DISPLAY C-64 SCREEN CODES
110 REM   IN 16 BY 16 MATRIX
120 REM
130 PRINT CHR$(147)  :  REM CLEAR SCREEN
140 SCREEN=1024   :  REM SCREEN MEMORY
150 CMEM=55296    :  REM COLOR MEMORY
160 C=PEEK(646)   :  REM CURRENT COLOR
170 REM
180 FOR ROW=0 TO 15
190 FOR COL=0 TO 15
200 POKE CMEM+COL+40*ROW,C
210 POKE SCREEN+COL+40*ROW,COL+16*ROW
220 NEXT COL,ROW
230 REM
240 GOTO 240
```

This program clears the screen, pokes the current color code into the appropriate bytes of color memory, and pokes all 256 screen codes into a 16 by 16 grid of screen memory. When it has done so, it will sit in an endless loop in line 240. To break it out of this loop, press the RUN/STOP key.

FIXCHR

As you can see, the C-64's screen codes require a translation from ASCII. However, FIXCHR must do more than simply convert an ASCII code to a C-64 screen code. It must also store the current color code in the appropriate byte of color memory. The following source code for FIXCHR will accomplish both of these tasks:

FIXCHR	A character is in A. We must convert it to proper C-64 screen code.
	But first, put a color code in appropriate byte of color memory. (Otherwise, that byte in color memory

```
                                  might hold the background
                                  color code, rendering the
                                  character invisible.)
              PHA                 Save character to be
                                  displayed.
              LDA TV.PTR+1        Save high byte...
              PHA                 ...of TV.PTR.
              CLC                 Make TV.PTR point
              ADC #$D4            to appropriate byte
              STA TV.PTR+1        of color memory.
              LDY #0
              LDA $286            Get current color code.
                                  Store it in appropriate
                                  byte of color memory:
              STA TV.PTR+1        to its original value.

              PLA                 Retrieve character to be
                                  displayed.
              SEC                 Prepare to compare.
              CMP #$40            Is it less than $40? (Is
                                  it a number or a punctuation
                                  mark?)
              BCC FIXEND          If so, no conversion needed.
              CMP #$60            Is it in the range $40...$5F?
              BCC SUB.40          If so, subtract $40 to
                                  convert from ASCII to
                                  C-64 screen code.
                                  OK. It's greater than $5F.
              SBC #$20            Subtract $20 to convert
                                  lower case ASCII to C-64.
              RTS                 and return.

SUB.40        SEC                 Prepare to subtract.
              SBC #$40            Subtract $40 to convert ASCII
                                  uppercase char to C-64 code.
FIXEND        RTS                 Return, with A holding
                                  C-64 screen code for ASCII
                                  char originally in A.
```

C-64 Keyboard Input Routine

To get an ASCII character from the C-64 keyboard, call the following subroutine:

```
C64KEY      JSR $FFE4       Call C-64 ROM key scan routine.
            CMP #0          Zero means no key.
            BEQ C64KEY      If no key, scan again.
            RTS             Return with ASCII character
                            from the keyboard.
```

This subroutine yields the uppercase ASCII code for any letter key that you depress, and the proper ASCII code for any digit key or punctuation key.

C-64 TVT Routine

To print an ASCII character to the screen, call $FFD2, a C-64 ROM routine I will refer to as C-64TVT.

Any printable ASCII character passed to $FFD2 will be printed properly to the screen at the C-64's current TVT screen location. You may change the C-64's current TVT screen location (which is *not* the same as the current location used by the screen utilities in Chapter 5) by calling C-64TVT with the accumulator holding any of the control codes from Table B2.2.

These control codes may be passed directly to C-64TVT, or they may be included with a string of characters to be printed by "PRINT:" or "PR.MSG." For example, if you wish to clear the screen before printing a message, just put the

Table B2.2: *Control codes that affect the next character to be printed by C-64TVT.*

Character Name	Code	Function
CURSOR NORTH	$91	Move current location up by one row.
CURSOR EAST	$1D	Move current location one column to the right.
CURSOR SOUTH	$11	Move current location down by one row.
CURSOR WEST	$9D	Move current location left by one column.
INSERT	$94	Move current character, and all characters to its right, one column to the right.
DELETE	$14	Move current character, and all characters to its right, one column to the left.
HOME	$13	Set current location to upper left of screen.
CLEAR	$93	Set current location to the upper left corner and clear the screen.
REVERSE	$12	Select reverse video for following characters.
REVERSE-OFF	$92	Select normal video mode for following characters.

CLEAR character ($93) at the beginning of your message string, immediately following the STX. The message-printing subroutine will get the CLEAR character and pass it to PR.CHR, which, in turn, will pass it through the ROMTVT vector on to the C-64TVT routine. The C-64TVT routine will then clear the screen and set the current location to the upper left corner of the screen.

The next character in the string will then be printed in the upper left corner of a clear screen. If, instead of printing your message at the top row of a clear screen, you'd prefer to print it in the fifth row of a clear screen, just follow the CLEAR character with four CURSOR-SOUTH characters ($11, $11, $11, $11), and follow the four cursor-south characters with the text of your message. Following the text of your message, of course, you must include an ETX ($FF).

You might never use the C-64TVT control codes, but it's good to know they're available, should you ever want your C-64's display screen to perform as something more than a glass teletype.

Setting the Top of Memory

Before you can load the Visible Monitor (or the Extended Visible Monitor) into your C-64, you must insure that your C-64's BASIC interpreter won't use memory above $2FFF. To do so, type the following lines into your C-64 *immediately after you have turned it on*:

```
POKE 52,47 :POKE 56,47
POKE 51,255:POKE 51,255
NEW
```

(Be sure to press RETURN after typing each line.)

Now you may enter and run BASIC programs, without disturbing the memory used by the software in this book. Remember: you *must* set the top of memory, as shown above, each time you load the Visible Monitor and its extensions into your C-64. (See chapter 13.)

Invoking the Visible Monitor from BASIC

Once you have loaded the Visible Monitor (using the BASIC Object Code Loader and the "E" series of appendices), you can activate the Visible Monitor from BASIC with the following BASIC command:

```
SYS 12431
```

You may then return from the Visible Monitor to BASIC by pressing "Q" ("Q" for Quit).

Getting Hard Copy

The Printing Hexdump program, the Printing Disassembler, and the Simple Text Editor all direct their output to a printer. Actually, that's not quite true; they direct their output to any device you designate as logical file #2, be it a printer, a disk file, the modem, or whatever.

If you wish to output text and data from the software in this book to any device, you must first OPEN that device as *logical file* #2. The easiest way to do that is in a BASIC program, prior to activating the Visible Monitor. For example, here's a BASIC program that opens a 1200-baud channel on the RS-232 port, and then transfers control to the Visible Monitor:

```
100 OPEN 2,2,0,CHR$(8)
110 SYS 12431
120 CLOSE 2
```

Line 100 opens the RS-232 port as logical file #2 (configuring it to operate at 1200 baud). Then line 110 passes control to the BASIC ENTRY point of the Visible Monitor. (See appendix C13.) From the Visible Monitor, you may then select any of the software that prints, and it will direct its output to the RS-232 port. If you have a printer connected to that port, you will then get a hard copy of the hexdump, disassembly, or text.

If you replace line 100 in the above BASIC program with a line that opens a DISK file as logical file #2, then you'll direct the hexdump or disassembly to a disk file. Or replace line 100 with a line that opens the Commodore printer as device #2, and you'll send the hexdump or disassembly to the Commodore printer. Thus, you can send the hardcopy output to any device you desire, simply by opening that device as logical file #2 before invoking the Visible Monitor through its BASIC entry point (at 12431).

NOTE: If you don't open a device or file as logical device #2, then attempting to use any of the printing software in this book will cause it to print every character *twice* on the screen. So if you notice that your C-64 has developed a ssttuutteerr when you try to print a hexdump or disassembly, the cure is simple: just exit to BASIC and open the desired output device as logical file #2 before re-entering the Visible Monitor.

Appendix C1:

Screen Utilities

```
;      APPENDIX C1: ASSEMBLER LISTING OF
;                   SCREEN UTILITIES
;
;
;
;  SEE CHAPTER 5 OF TOP-DOWN ASSEMBLY LANGUAGE
;  PROGRAMMING FOR YOUR COMMODORE 64 AND VIC-20
;
;
;                  BY KEN SKIER
;
;
;          COPYRIGHT (C) 1984 BY KENNETH SKIER
;                LEXINGTON, MASSACHUSETTS
;
;
;
;
;
;
;
;
;
;
;
; *******************************************
;
;              ZERO PAGE BYTES
;
; *****************************************
;
;
;
;
;
      TV.PTR = $FB    THIS POINTER HOLDS THE
;                     ADDRESS OF THE CURRENT
;                     SCREEN LOCATION.
;
;
;
;
;
;
;
;
; *******************************************
;
;          SCREEN PARAMETERS
;
; *******************************************
;
;
;
```

```
                         PARAMS = $3000
                    ;             THE FOLLOWING ADDRESSES
                    ;             MUST BE INITIALIZED TO HOLD
                    ;             DATA DESCRIBING THE SCREEN
                    ;             ON YOUR SYSTEM.
                    ;
                    ;
                    ;
                    ;
                         HOME   = PARAMS
                    ;             HOME IS A POINTER TO CHARACTER
                    ;             POSITION IN UPPER LEFT CORNER.
                    ;
                         ROWINC = PARAMS+2
                    ;             ROWINC IS A BYTE GIVING
                    ;             ADDRESS DIFFERENCE FROM ONE
                    ;             ROW TO THE NEXT.
                    ;
                         TVCOLS = PARAMS+3
                    ;             TVCOLS IS A BYTE GIVING
                    ;             NUMBER OF COLUMNS ON SCREEN.
                    ;             (COUNTING FROM ZERO.)
                    ;
                         TVROWS = PARAMS+4
                    ;             TVROWS IS A BYTE GIVING
                    ;             NUMBER OF ROWS ON SCREEN,
                    ;             (COUNTING FROM ZERO.)
                    ;
                         HIPAGE = PARAMS+5
                    ;             HIPAGE IS THE HIGH BYTE OF
                    ;             THE HIGHEST ADDRESS ON SCREEN.
                    ;
                    ;
                         BLANK  = PARAMS+6
                    ;             YOUR SYSTEM'S CHARACTER
                    ;             CODE FOR A BLANK.
                    ;
                         ARROW  = PARAMS+7
                    ;             YOUR SYSTEM'S CHARACTER
                    ;             FOR AN UP-ARROW.
                    ;
                         FIXCHR = PARAMS+$11
                    ;             FIXCHR IS A SUBROUTINE THAT
                    ;             RETURNS YOUR SYSTEM'S
                    ;             DISPLAY CODE FOR ASCII.
                    ;             CODE.
                    ;
                    ;
                    ;
                    ;
                    ;
0000    = 3100           * =  $3100
                    ;
                    ;
```

```
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;  *******************************************
                          ;
                          ;                 CLEAR SCREEN
                          ;
                          ;  *******************************************
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;        CLEAR SCREEN, PRESERVING THE ZERO PAGE.
                          ;
                          ;
                          ;
                          ;
3100  20C431          CLR.TV JSR TVPUSH      SAVE ZERO PAGE BYTES THAT
                          ;                  WILL BE CHANGED.
3103  202B31                 JSR TVHOME      SET SCREEN LOCATION TO UPPER
                          ;                  LEFT CORNER OF THE SCREEN.
3106  AE0330                 LDX TVCOLS      LOAD X,Y REGISTERS WITH
3109  AC0430                 LDY TVROWS      X,Y DIMENSIONS OF SCREEN.
310C  201331                 JSR CLR.XY      CLEAR X COLUMNS, Y ROWS
                          ;                  FROM CURRENT SCREEN LOCATION.
310F  20D331                 JSR TV.POP      RESTORE ZERO PAGE BYTES THAT
                          ;                  WERE CHANGED.
3112  60                     RTS             RETURN TO CALLER, WITH ZERO
                          ;                  PAGE PRESERVED.
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;  *******************************************
                          ;
                          ;             CLEAR PORTION OF SCREEN
                          ;
                          ;  *******************************************
                          ;
                          ;
                          ;
                          ;
```

```
                          ;               CLEAR X COLUMNS, Y ROWS
                          ;               FROM CURRENT SCREEN LOCATION.
                          ;               MOVES TV.PTR DOWN BY Y ROWS.
                          ;
                          ;
                          ;
3113  8E2A31     CLR.XY STX COLS          SET THE NUMBER OF COLUMNS
                          ;               TO BE CLEARED.
3116  98                TYA
3117  AA                TAX               NOW X HOLDS NUMBER OF ROWS
                          ;               TO BE CLEARED.
                          ;
3118  AD0630     CLRROW LDA BLANK         WE'LL CLEAR THEM BY
                          ;               WRITING BLANKS TO THE
                          ;               SCREEN.
311B  AC2A31            LDY COLS          LOAD Y WITH NUMBER OF
                          ;               COLUMNS TO BE CLEARED.
311E  91FB       CLRPOS STA (TV.PTR),Y    CLEAR A POSITION BY
                          ;               WRITING A BLANK INTO IT.
                          ;
3120  88                DEY               ADJUST INDEX FOR NEXT
                          ;               POSITION ON THE ROW.
                          ;
3121  10FB              BPL CLRPOS        IF NOT DONE WITH ROW,
                          ;               CLEAR NEXT POSITION...
                          ;
3123  207631            JSR TVDOWN        IF DONE WITH ROW, MOVE
                          ;               CURRENT SCREEN LOCATION
                          ;               DOWN BY ONE ROW.
                          ;
3126  CA                DEX               DONE LAST ROW YET?
3127  10EF              BPL CLRROW        IF NOT, CLEAR NEXT ROW...
3129  60                RTS               IF SO, RETURN TO CALLER.
                          ;
312A  00         COLS   .BYTE 0           DATA CELL: HOLDS NUMBER OF
                          ;               COLUMNS TO BE CLEARED.
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ; *******************************************
                          ;
                          ;               TVHOME
                          ;
                          ; *******************************************
                          ;
                          ;
                          ;
```

```
                    ;
                    ;
312B  A200          TVHOME LDX #0       SET TV.PTR TO UPPER LEFT
312D  A000                 LDY #0       CORNER OF SCREEN, BY
                    ;                   ZEROING X AND Y AND THEN
312F  18                   CLC          GOING TO X,Y COORDINATES:
3130  900A                 BCC TVTOXY
                    ;
                    ;
                    ;
                    ;
                    ; ********************************************
                    ;
                    ;                    CENTER
                    ;
                    ; ********************************************
                    ;
                    ;
                    ;
                    ;
                    ;                   SET TV.PTR TO SCREEN'S
                    ;                   CENTER:
                    ;
                    ;
                    ;
                    ;
3132  AD0430        CENTER LDA TVROWS   LOAD A WITH TOTAL ROWS.
3135  4A                   LSR A        DIVIDE IT BY TWO.
3136  A8                   TAY          Y NOW HOLDS THE NUMBER OF
                    ;                   THE SCREEN'S CENTRAL ROW.
                    ;
3137  AD0330               LDA TVCOLS   LOAD A WITH TOTAL COLUMNS.
313A  4A                   LSR A        DIVIDE IT BY TWO.
313B  AA                   TAX          X NOW HOLDS THE NUMBER OF
                    ;                   THE SCREEN'S CENTRAL COLUMN.
                    ;
                    ;
                    ;                   X AND Y REGISTERS NOW HOLD
                    ;                   X,Y COORDINATES OF CENTER
                    ;                   OF SCREEN.
                    ;
                    ;                   SO NOW LET'S SET THE SCREEN
                    ;                   LOCATION TO THOSE X,Y
                    ;                   COORDINATES:
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ; ********************************************
```

```
                   ;
                   ;                        TVTOXY
                   ;
                   ; *******************************************
                   ;
                   ;
                   ;
                   ;
                   ;
313C  38           TVTOXY SEC                   SET CURRENT SCREEN LOCATION
                   ;                            TO COORDINATES GIVEN BY
                   ;                            THE X AND Y REGISTERS.
                   ;
313D  EC0330              CPX TVCOLS            IS X OUT OF RANGE?
3140  9003                BCC X.OK              IF NOT, LEAVE IT ALONE.
                   ;                            IF X IS OUT OF RANGE, GIVE
3142  AE0330              LDX TVCOLS            IT ITS HIGHEST LEGAL VALUE.
                   ;                            NOW X IS LEGAL.
                   ;
3145  38           X.OK   SEC                   IS Y OUT OF RANGE?
3146  CC0430              CPY TVROWS
3149  9003                BCC Y.OK              IF NOT, LEAVE IT ALONE.
                   ;
                   ;                            IF Y IS OUT OF RANGE, GIVE
314B  AC0430              LDY TVROWS            Y ITS HIGHEST LEGAL VALUE.
                   ;                            NOW Y IS LEGAL.
                   ;
                   ;
314E  AD0030       Y.OK   LDA HOME              SET TV.PTR EQUAL TO LOWEST
3151  85FB                STA TV.PTR            SCREEN ADDRESS.
3153  AD0130              LDA HOME+1
3156  85FC                STA TV.PTR+1
                   ;
3158  08                  PHP                   SAVE CALLER'S DECIMAL FLAG.
3159  D8                  CLD                   CLEAR DECIMAL FOR BINARY
                   ;                            ADDITION.
                   ;
315A  8A                  TXA                   ADD X TO TV.PTR
315B  18                  CLC
315C  65FB                ADC TV.PTR
315E  9003                BCC COLSET
3160  E6FC                INC TV.PTR+1
3162  18                  CLC
                   ;
                   ;
3163  C000         COLSET CPY #0                ADD Y*ROWINC TO TV.PTR:
3165  F00B                BEQ TV.SET
3167  18           ADDROW CLC
3168  6D0230              ADC ROWINC
316B  9002                BCC *+4
316D  E6FC                INC TV.PTR+1
316F  88                  DEY
3170  D0F5                BNE ADDROW
                   ;
```

```
                        ;
3172  85FB        TV.SET STA TV.PTR
3174  28                 PLP              RESTORE CALLER'S DECIMAL FLAG
3175  60                 RTS              RETURN TO CALLER
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ; ******************************************
                        ;
                        ;          TVDOWN, TVSKIP, AND TVPLUS
                        ;
                        ; ******************************************
                        ;
                        ;
                        ;
                        ;
                        ;
3176  AD0230      TVDOWN LDA ROWINC       MOVE TV.PTR DOWN BY ONE ROW.
3179  18                 CLC
317A  9005               BCC TVPLUS
                        ;
317C  209B31      VUCHAR JSR TV.PUT       PUT CHARACTER ON SCREEN
                        ;                 AND THEN
                        ;
317F  A901        TVSKIP LDA #1           SKIP ONE SCREEN LOCATION
                        ;                 BY INCREMENTING TV.PTR
                        ;
                        ;
3181  08          TVPLUS PHP              TVPLUS ADDS ACCUMULATOR
3182  D8                 CLD              TO TV.PTR, KEEPING TV.PTR
3183  18                 CLC              WITHIN SCREEN MEMORY.
3184  65FB               ADC TV.PTR
3186  9002               BCC *+4
3188  E6FC               INC TV.PTR+1
318A  85FB               STA TV.PTR
318C  38                 SEC              IS CURRENT SCREEN LOCATION
318D  AD0530             LDA HIPAGE       OUTSIDE OF SCREEN MEMORY?
3190  C5FC               CMP TV.PTR+1
3192  B005               BCS TV.OK
                        ;
3194  AD0130             LDA HOME+1       IF SO, WRAP AROUND FROM
3197  85FC               STA TV.PTR+1     BOTTOM TO TOP OF SCREEN.
                        ;
3199  28          TV.OK  PLP              RESTORE ORIGINAL DECIMAL
319A  60                 RTS              FLAG AND RETURN TO CALLER.
                        ;
                        ;
```

```
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ; ********************************************
                        ;
                        ;                 TV.PUT
                        ;
                        ; ********************************************
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
319B  201130           TV.PUT JSR FIXCHR     CONVERT ASCII CHARACTER
                        ;                    TO YOUR SYSTEM'S DISPLAY
                        ;                    CODE.
                        ;
319E  A000                    LDY #0         PUT CHARACTER AT CURRENT
31A0  91FB                    STA (TV.PTR),Y SCREEN LOCATION.
31A2  60                      RTS            THEN RETURN.
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ; ********************************************
                        ;
                        ;         DISPLAY A BYTE IN HEX FORMAT
                        ;
                        ; ********************************************
                        ;
                        ;
                        ;
                        ;
                        ;
31A3  48               VUBYTE PHA            SAVE BYTE TO BE DISPLAYED.
31A4  4A                      LSR A          MOVE 4 MOST SIGNIFICANT
31A5  4A                      LSR A          BITS INTO POSITIONS
31A6  4A                      LSR A          FORMERLY OCCUPIED BY 4
31A7  4A                      LSR A          LEAST SIGNIFICANT BITS.
                        ;
31A8  20B631                  JSR ASCII      DETERMINE ASCII CHAR FOR
                        ;                    HEX DIGIT IN A'S 4 LSB.
                        ;
```

```
31AB  207C31          JSR VUCHAR      DISPLAY THAT ASCII CHAR ON
                 ;                    SCREN AND ADVANCE TO NEXT
                 ;                    SCREEN LOCATION.
                 ;
31AE  68              PLA             RESTORE ORIGINAL BYTE TO A.
31AF  20B631          JSR ASCII       DETERMINE ASCII CHAR FOR
                 ;                    A'S 4 LSB.
                 ;
31B2  207C31          JSR VUCHAR      STORE THIS ASCII CHAR JUST
                 ;                    TO THE RIGHT OF THE OTHER
                 ;                    ASCII CHAR, AND ADVANCE TO
                 ;                    NEXT SCREEN POSITION.
                 ;
                 ;
31B5  60              RTS             RETURN TO CALLER.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; ****************************************
                 ;
                 ;                 HEX-TO-ASCII
                 ;
                 ; ****************************************
                 ;
                 ;
                 ;
                 ;
                 ;
31B6  08         ASCII  PHP           THIS ROUTINE RETURNS ASCII
31B7  D8                CLD           FOR 4 LSB IN ACCUMULATOR.
31B8  290F              AND #$0F      CLEAR HIGH 4 BITS IN A.
31BA  C90A              CMP #$0A      IS ACCUMULATOR GREATER
                 ;                    THAN 9?
31BC  3002              BMI DECIML    IF NOT, IT MUST BE 0-9.
                 ;
31BE  6906              ADC #6        IF SO, IT MUST BE A-F.
                 ;                    ADD 36 HEX TO CONVERT IT.
                 ;                    TO CORRESPONDING ASCII CHAR.
31C0  6930       DECIML ADC #$30      IF A IS 0-9, ADD 30 HEX
                 ;                    TO CONVERT IT TO
                 ;                    CORRESPONDING ASCII CHAR.
                 ;
31C2  28                PLP           RESTORE ORIGINAL DECIMAL
                 ;                    FLAG, AND
31C3  60                RTS           RETURN TO CALLER
                 ;
                 ;
```

```
5860                      ;
5870                      ;
5880                      ;
5890                      ;
5900                      ;
5910                      ;
5920                      ;
5930                      ;
5940                      ;
5950                      ;
5960                      ;
5970                      ; ********************************************
5980                      ;
5990                      ;                    TVPUSH
6000                      ;
6010                      ; ********************************************
6020                      ;
6030                      ;
6040                      ;
6050                      ;                    SAVE CURRENT SCREEN LOCATION
6060                      ;                    ON STACK, FOR CALLER.
6070                      ;
6080                      ;
6090                      ;
6100                      ;
6110                      ;
6120  31C4   68           TVPUSH PLA           PULL RETURN ADDRESS FROM
6130  31C5   AA                  TAX           STACK AND SAVE IT IN X AND
6140  31C6   68                  PLA           Y REGISTERS.
6150  31C7   A8                  TAY
6160                      ;
6170                      ;
6180  31C8   A5FC                LDA TV.PTR+1  GET TV.PTR AND
6190  31CA   48                  PHA
6200  31CB   A5FB                LDA TV.PTR    PUSH IT ONTO THE STACK.
6210  31CD   48                  PHA
6220                      ;
6230                      ;
6240  31CE   98                  TYA           PLACE RETURN ADDRESS
6250  31CF   48                  PHA
6260  31D0   8A                  TXA           BACK ON STACK.
6270  31D1   48                  PHA
6280                      ;
6290                      ;
6300  31D2   60                  RTS           THEN RETURN TO CALLER.
6310                      ;                    CALLER WILL FIND TV.PTR ON
6320                      ;                    STACK, LOW BYTE ON TOP.
6330                      ;
6340                      ;
6350                      ;
6360                      ;
6370                      ;
6380                      ;
6390                      ;
```

```
                          ;
                          ;
                          ;
                          ; ******************************************
                          ;
                          ;                 TV.POP
                          ;
                          ; ******************************************
                          ;
                          ;
                          ;
                          ;                  RESTORE SCREEN LOCATION
                          ;                  PREVIOUSLY SAVED ON STACK.
                          ;
                          ;
                          ;
31D3   68                TV.POP PLA          PULL RETURN ADDRESS FROM
31D4   AA                       TAX          STACK, SAVING IT IN X...
31D5   68                       PLA
31D6   A8                       TAY          ...AND IN Y
                          ;
                          ;
31D7   68                       PLA          RESTORE...
31D8   85FB                     STA TV.PTR   ...TV.PTR
31DA   68                       PLA             ...FROM
31DB   85FC                     STA TV.PTR+1       ...STACK.
                          ;
                          ;
31DD   98                       TYA          PLACE RETURN ADDRESS
31DE   48                       PHA          BACK ...
31DF   8A                       TXA
31E0   48                       PHA          ...ON STACK.
                          ;
                          ;
31E1   60                       RTS          RETURN TO CALLER.
```

CROSS REFERENCE LISTING:

ADDROW	3167	ARROW	3007	ASCII	31B6	BLANK	3006
CENTER	3132	CLR.TV	3100	CLR.XY	3113	CLRPOS	311E
CLRROW	3118	COLS	312A	COLSET	3163	DECIML	31C0
FIXCHR	3011	HIPAGE	3005	HOME	3000	PARAMS	3000
ROWINC	3002	TV.OK	3199	TV.POP	31D3	TV.PTR	00FB
TV.PUT	319B	TV.SET	3172	TVCOLS	3003	TVDOWN	3176
TVHOME	312B	TVPLUS	3181	TVPUSH	31C4	TVROWS	3004
TVSKIP	317F	TVTOXY	313C	VUBYTE	31A3	VUCHAR	317C
X.OK	3145	Y.OK	314E				

Appendix C2:

Visible Monitor (Top Level and Display Subroutines)

```
                ;       APPENDIX C2: ASSEMBLER LISTING OF
                ;                   THE VISIBLE MONITOR
                ;
                ;       TOP LEVEL AND DISPLAY SUBROUTINES
                ;
                ;
                ;
                ;

                ;  SEE CHAPTER 6 OF TOP-DOWN ASSEMBLY LANGUAGE
                ;  PROGRAMMING FOR YOUR COMMODORE 64 AND VIC
                ;
                ;                 BY KEN SKIER
                ;
                ;
                ;            COPYRIGHT (C) 1984 BY KENNETH SKIER
                ;                  LEXINGTON, MASSACHUSETTS
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; ********************************************
                ;
                ;                     EQUATES
                ;
                ; ********************************************
                ;
                ;
                ;
                ;
                        TV.PTR = $FB
                ;
                        GETPTR = $FB
                ;
                ;
                        PARAMS = $3000  ADDRESS OF SYSTEM DATA
                ;                       BLOCK.
                ;
                        TVCOLS = PARAMS+3
                ;                       TVCOLS IS A BYTE GIVING
                ;                       NUMBER OF COLUMNS ON SCREEN.
                ;                       (COUNTING FROM ZERO.)
```

```
                ;
                ;
                     ARROW  = PARAMS+7
                ;                      THIS DATA BYTE HOLDS YOUR
                ;                      SYSTEM'S CHARACTER CODE
                ;                      FOR AN UP-ARROW.
                ;
                     ROMKEY = PARAMS+8
                ;                      ROMKEY IS A POINTER TO
                ;                      YOUR SYSTEM'S SUBROUTINE
                ;                      TO GET AN ASCII CHARACTER
                ;                      FROM THE KEYBOARD.
                ;
                     SPACE  = $20
                ;
                     RUBOUT = $7F
                ;
                     CR     = $0D      ASCII FOR CARRIAGE RETURN.
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; ********************************************
                ;
                ;          REQUIRED SUBROUTINES
                ;
                ; ********************************************
                ;
                ;
                ;
                     TVSUBS = $3100
                     CLR.TV = TVSUBS
                     CLR.XY = TVSUBS+$13
                     TVHOME = TVSUBS+$2B
                     TVTOXY = TVSUBS+$3C
                     TVDOWN = TVSUBS+$76
                     VUCHAR = TVSUBS+$7C
                     TVSKIP = TVSUBS+$7F
                     TVPLUS = TVSUBS+$81
                     VUBYTE = TVSUBS+$A3
                     ASCII  = TVSUBS+$B6
                     TVPUSH = TVSUBS+$C4
                     TV.POP = TVSUBS+$D3
                ;
                ;
                ;
```

```
0000  = 3200           * = $3200
                 ;
                 ;
                 ;
                       UPDATE = *+$E3
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;  ******************************************
                 ;
                 ;           USER-MODIFIABLE DATA
                 ;
                 ;  ******************************************
                 ;
                 ;
                 ;
                 ;
                 ;
3200  00         FIELD  .BYTE 0        NUMBER OF CURRENT FIELD.
                 ;                     (MUST BE 0-6.)
                 ;
3201  00         REG.A  .BYTE 0        IMAGE OF ACCUMULATOR.
                 ;
3202  00         REG.X  .BYTE 0        IMAGE OF X-REGISTER.
                 ;
3203  00         REG.Y  .BYTE 0        IMAGE OF Y-REGISTER.
                 ;
3204  00         REG.P  .BYTE 0        IMAGE OF PROCESSOR STATUS
                 ;                     REGISTER.
                 ;
                        REGS   = REG.A
                 ;
3205  0000       SELECT .WORD 0        POINTER TO CURRENTLY-
                 ;                     SELECTED ADDRESS.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;  ******************************************
                 ;
```

```
                          ;          THE VISIBLE MONITOR
                          ;
                          ; ****************************************
                          ;
                          ;
                          ;
                          ;
3207   08         VISMON PHP               SAVE CALLER'S STATUS FLAGS.
3208   D8                CLD               CLEAR DECIMAL MODE, SINCE
                          ;                ARITHMETIC OPERATIONS IN THIS
                          ;                BOOK ARE ALWAYS BINARY.
                          ;
3209   201232            JSR DSPLAY        PUT MONITOR DISPLAY ON
                          ;                SCREEN.
                          ;
320C   20E332            JSR UPDATE        GET USER REQUEST AND
                          ;                HANDLE IT.
320F   18                CLC
3210   90F6              BCC VISMON+1      LOOP BACK TO DISPLAY...
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ; ****************************************
                          ;
                          ;             MONITOR-DISPLAY
                          ;
                          ; ****************************************
                          ;
                          ;
                          ;
                          ;
                          ;
3212   20C431     DSPLAY JSR TVPUSH        SAVE ZERO PAGE BYTES THAT
                          ;                WILL BE MODIFIED.
                          ;
3215   202532            JSR CLRMON        CLEAR A PORTION OF SCREEN.
3218   203532            JSR LINE.1        DISPLAY LABEL LINE.
321B   205D32            JSR LINE.2        DISPLAY DATA LINE.
321E   20B032            JSR LINE.3        DISPLAY ARROW LINE.
                          ;
3221   20D331            JSR TV.POP        RESTORE ZERO PAGE BYTES
                          ;                THAT WERE SAVED ABOVE.
                          ;
3224   60                RTS               RETURN TO CALLER.
                          ;
                          ;
                          ;
                          ;
```

```
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ; *****************************************
                        ;
                        ;         CLEAR PORTION OF SCREEN
                        ;
                        ; *****************************************
                        ;
                        ;
                        ;
                        ;
                        ;
3225  A200              CLRMON LDX #0          SET TV.PTR TO COLUMN 0,
3227  A000                     LDY #0          ROW 0.
3229  203C31                   JSR TVTOXY
                        ;
322C  AE0330                   LDX TVCOLS      LOAD X WITH NUMBER OF
                        ;                      COLUMNS TO BE CLEARED.
                        ;
322F  A003                     LDY #3          LOAD Y WITH NUMBER OF
                        ;                      ROWS (3) TO BE CLEARED.
                        ;
3231  201331                   JSR CLR.XY      CLEAR X COLUMNS, Y ROWS.
                        ;
3234  60                       RTS             RETURN TO CALLER.
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ; *****************************************
                        ;
                        ;           DISPLAY LABEL LINE
                        ;
                        ; *****************************************
                        ;
                        ;
                        ;
                        ;
                        ;
3235  A20B              LINE.1 LDX #11         X-COORDINATE OF LABEL "A".
3237  A000                     LDY #0          Y-COORDINATE OF LABEL "A".
3239  203C31                   JSR TVTOXY      SET TV.PTR TO POINT TO
                        ;                      SCREEN LOCATION OF LABEL "A".
                        ;
```

```
3230  A000                    LDY #0          PUT LABELS ON SCREEN:
323E  8C5232                  STY LBLCOL      INITIALIZE LABEL COLUMN
                    ;                         COUNTER.
                    ;
3241  B95332        LBLOOP    LDA LABELS,Y    GET A CHARACTER AND
3244  207C31                  JSR VUCHAR      PUT IT ON THE SCREEN.
3247  EE5232                  INC LBLCOL      PREPARE FOR NEXT CHARACTER.
324A  AC5232                  LDY LBLCOL      DONE LAST CHARACTER?
324D  C00A                    CPY #10
324F  D0F0                    BNE LBLOOP      IF NOT, DO NEXT CHARACTER.
                    ;
3251  60                      RTS             RETURN TO CALLER.
3252  00            LBLCOL    .BYTE 0         DATA CELL: HOLDS COLUMN
                    ;                         OF CHARACTER TO BE COPIED.
                    ;
                    ;
                    ;
                    ;
3253  4120205820    LABELS    .BYTE 'A   X   Y   P'     ;
      2059202050
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ; ******************************************
                    ;
                    ;              DISPLAY DATA LINE
                    ;
                    ; ******************************************
                    ;
                    ;
                    ;
                    ;
                    ;
325D  A200          LINE.2    LDX #0          LOAD X WITH STARTING
                    ;                         COLUMN OF DATA LINE.
                    ;
325F  A001                    LDY #1          LOAD Y WITH ROW NUMBER
                    ;                         OF DATA LINE.
                    ;
3261  203C31                  JSR TVTOXY      SET TV.PTR TO POINT TO
                    ;                         THE START OF THE DATA LINE.
                    ;
3264  AD0632                  LDA SELECT+1    DISPLAY HIGH BYTE OF
3267  20A331                  JSR VUBYTE      CURRENTLY-SELECTED ADDRESS.
326A  AD0532                  LDA SELECT      DISPLAY LOW BYTE OF
326D  20A331                  JSR VUBYTE      CURRENTLY-SELECTED ADDRESS.
                    ;
```

```
3270  207F31          JSR TVSKIP     SKIP ONE SPACE AFTER
                 ;                   ADDRESS FIELD.
                 ;
3273  209532          JSR GET.SL     GET CURRENTLY-SELECTED
                 ;                   BYTE.
                 ;
3276  48              PHA            SAVE IT.
                 ;
3277  20A331          JSR VUBYTE     DISPLAY IT, IN HEX FORMAT,
                 ;                   IN FIELD 1.
                 ;
327A  207F31          JSR TVSKIP     SKIP ONE SPACE AFTER FIELD
                 ;                   1.
                 ;
327D  68              PLA            RESTORE CURRENTLY-SELECTED
                 ;                   BYTE TO ACCUMULATOR.
                 ;
327E  207C31          JSR VUCHAR     DISPLAY IT IN CHARACTER
                 ;                   FORMAT, IN FIELD 2.
                 ;
3281  207F31          JSR TVSKIP     SKIP ONE SPACE AFTER FIELD 2.
                 ;
                 ;
                 ;                   DISPLAY 6502 REGISTER
                 ;                   IMAGES IN FIELDS 3-6:
                 ;
3284  A200            LDX #0         START WITH ACCUMULATOR
                 ;                   IMAGE.
3286  BD0132     VUREGS LDA REGS,X   LOOK UP THE REGISTER IMAGE.
3289  20A331          JSR VUBYTE     DISPLAY IT IN HEX FORMAT.
328C  207F31          JSR TVSKIP     SKIP ONE SPACE AFTER HEX
                 ;                   FIELD.
                 ;
328F  E8              INX            GET READY FOR NEXT REGISTER...
3290  E004            CPX #4         DONE FOUR REGISTERS YET?
3292  D0F2            BNE VUREGS     IF NOT, DO NEXT ONE...
                 ;
3294  60              RTS            IF ALL REGISTERS DISPLAYED,
                 ;                   RETURN.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; *******************************************
                 ;
                 ;          GET SELECTED BYTE
                 ;
```

```
                      ; *******************************************
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
3295   A5FB         GET.SL LDA GETPTR     GET BYTE POINTED TO BY
3297   48                  PHA            THE SELECT POINTER
3298   A6FC                LDX GETPTR+1   (PRESERVING THE ZERO PAGE).
                      ;
329A   AD0532              LDA SELECT
329D   85FB                STA GETPTR
329F   AD0632              LDA SELECT+1
32A2   85FC                STA GETPTR+1
                      ;
32A4   A000                LDY #0
32A6   B1FB                LDA (GETPTR),Y
32A8   A8                  TAY
32A9   68                  PLA
32AA   85FB                STA GETPTR
32AC   86FC                STX GETPTR+1
32AE   98                  TYA
32AF   60                  RTS            RETURN TO CALLER.
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ; *******************************************
                      ;
                      ;           DISPLAY ARROW LINE
                      ;
                      ; *******************************************
                      ;
                      ;
                      ;
                      ;
                      ;
32B0   AC0032       LINE.3 LDY FIELD      LOOK UP CURRENT FIELD.
32B3   38                  SEC
32B4   C007                CPY #7
32B6   9005                BCC FLD.OK
32B8   A000                LDY #0
32BA   8C0032              STY FIELD
32BD   B9CD32       FLD.OK LDA FIELDS,Y   LOOK UP COLUMN NUMBER FOR
                      ;                   CURRENT FIELD.
32C0   AA                  TAX            THAT WILL BE THE ARROW'S
                      ;                   X-COORDINATE.
```

```
32C1   A002                LDY #2          SET ARROW'S Y-COORDINATE.
32C3   203C31              JSR TVTOXY      MAKE TV.PTR POINT TO ARROW
                  ;                        LOCATION.
                  ;
32C6   AD0730              LDA ARROW       PLACE AN UP-ARROW IN
32C9   207C31              JSR VUCHAR      THAT LOCATION
32CC   60                  RTS             AND RETURN TO CALLER.
                  ;
                  ;
32CD   030608     FIELDS   .BYTE 3,6,8     THIS DATA AREA SHOWS WHICH
32D0   0B0E                .BYTE $0B,$0E   COLUMN SHOULD GET AN UP-
32D2   1114                .BYTE $11,$14   ARROW TO INDICATE ANY ONE
                  ;                        OF FIELDS 0-6.  CHANGING
                  ;                        ONE OF THESE VALUES WILL
                  ;                        CAUSE THE UP-ARROW TO APPEAR
                  ;                        IN A DIFFERENT COLUMN WHEN
                  ;                        INDICATING A GIVEN FIELD.
                  ;
                  ;
```

CROSS REFERENCE LISTING:

ARROW	3007	ASCII	31B6	CLR.TV	3100	CLR.XY	3113
CLRMON	3225	CR	000D	DSPLAY	3212	FIELD	3200
FIELDS	32CD	FLD.OK	32BD	GET.SL	3295	GETPTR	00FB
LABELS	3253	LBLCOL	3252	LBLOOP	3241	LINE.1	3235
LINE.2	325D	LINE.3	32B0	PARAMS	3000	REG.A	3201
REG.P	3204	REG.X	3202	REG.Y	3203	REGS	3201
ROMKEY	3008	RUBOUT	007F	SELECT	3205	SPACE	0020
TV.POP	31D3	TV.PTR	00FB	TVCOLS	3003	TVDOWN	3176
TVHOME	312B	TVPLUS	3181	TVPUSH	31C4	TVSKIP	317F
TVSUBS	3100	TVTOXY	313C	UPDATE	32E3	VISMON	3207
VUBYTE	31A3	VUCHAR	317C	VUREGS	3286		

Appendix C3:

Visible Monitor (Update Subroutine)

```
;       APPENDIX C3: ASSEMBLER LISTING OF
;                    THE VISIBLE MONITOR
;
;
;                    UPDATE SUBROUTINE
;
;
;
; SEE CHAPTER 6 OF TOP-DOWN ASSEMBLY-LANGUAGE
; PROGRAMMING FOR YOUR COMMODORE 64 AND VIC-20
;
;                    BY KEN SKIER
;
;
;             COPYRIGHT (C) 1984 BY KENNETH SKIER
;                   LEXINGTON, MASSACHUSETTS
;
;
;
;
;
;
;
;
;
;
; ******************************************
;
;                   EQUATES
;
; ******************************************
;
;
;
;
;
;
;
        TV.PTR = $FB
;
;
        PARAMS = $3000 ADDRESS OF SYSTEM DATA
;                      BLOCK.
;
;
        ARROW  = PARAMS+7
;                      THIS DATA BYTE HOLDS YOUR
;                      SYSTEM'S CHARACTER CODE
;                      FOR AN UP-ARROW.
;
        ROMKEY = PARAMS+8
;                      ROMKEY IS A POINTER TO
;                      YOUR SYSTEM'S SUBROUTINE
```

```
;                   TO GET AN ASCII CHARACTER
;                   FROM THE KEYBOARD.
;
        DUMMY  = PARAMS+$10
;                   DUMMY RETURNS WITHOUT DOING
;                   ANYTHING.
;
;
        SPACE  = $20
;
        CLRKEY = 147    CLEAR SCREEN KEY
;
        CR     = $0D    ASCII FOR CARRIAGE RETURN.
;
;
;
;
;
;
;
;
;
;
;
;
; ******************************************
;
;          REQUIRED SUBROUTINES
;
; ******************************************
;
;
;
;
;
        TVSUBS = $3100

        CLR.TV = TVSUBS
;                   CLR.TV CLEARS THE SCREEN.
;
;
        VMSUBS = $3200
;                   STARTING PAGE OF VISIBLE
;                   MONITOR CODE.
;
        GET.SL = VMSUBS+$95
;                   GET.SL GETS THE CURRENTLY-
;                   SELECTED BYTE.
;
;
;
;
;
;
```

```
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;  *******************************************
                         ;
                         ;          USER-MODIFIABLE DATA
                         ;
                         ;  *******************************************
                         ;
                         ;
                         ;
                         ;
                         ;
                              FIELD   = VMSUBS       NUMBER OF CURRENT FIELD.
                                                     (MUST BE 0-6.)
                         ;
                         ;
                              REG.A   = VMSUBS+1     IMAGE OF ACCUMULATOR.
                         ;
                              REG.X   = VMSUBS+2     IMAGE OF X-REGISTER.
                         ;
                              REG.Y   = VMSUBS+3     IMAGE OF Y-REGISTER.
                         ;
                              REG.P   = VMSUBS+4     IMAGE OF PROCESSOR STATUS
                                                     REGISTER.
                         ;
                         ;
                              REGS = REG.A
                         ;
                              SELECT  = VMSUBS+5     POINTER TO CURRENTLY-
                                                     SELECTED ADDRESS.
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;  *******************************************
                         ;
                         ;          KEYBOARD INPUT ROUTINE
                         ;
                         ;  *******************************************
                         ;
                         ;
0000   = 32E0                 * = VMSUBS+$E0
                         ;
                         ;
32E0   6C0830           GETKEY JMP (ROMKEY)   JSR GETKEY CALLS THE
```

```
                      ;                    COMMODORE KEYBOARD INPUT
                      ;                    ROUTINE INDIRECTLY.
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ; ******************************************
                      ;
                      ;             MONITOR-UPDATE
                      ;
                      ; ******************************************
                      ;
                      ;
                      ;
                      ;
                      ;
32E3  20E032        UPDATE JSR GETKEY      GET A CHARACTER FROM THE
                      ;                    KEYBOARD.
                      ;
32E6  C91D                 CMP #$1D        IS IT THE RIGHT-ARROW KEY?
32E8  D010                 BNE IF.LFT      IF NOT, PERFORM NEXT TEST.
                      ;
32EA  EE0032        NEXT.F INC FIELD       IF SO, SELECT NEXT FIELD.
32ED  AD0032               LDA FIELD
32F0  C907                 CMP #7          IF ARROW WAS UNDER RIGHT-
32F2  D005                 BNE UP.EX1      MOST FIELD, PLACE IT UNDER
32F4  A900                 LDA #0          LEFT-MOST FIELD.
32F6  8D0032               STA FIELD
32F9  60            UP.EX1 RTS             THEN RETURN TO CALLER.
                      ;
                      ;
32FA  C99D          IF.LFT CMP #$9D        IS IT THE LEFT-ARROW KEY?
32FC  D00B                 BNE IF.SP       IF NOT, PERFORM NEXT TEST.
                      ;
32FE  CE0032        PREV.F DEC FIELD       IF SO, SELECT PREVIOUS
3301  1005                 BPL UP.EX2      FIELD: THE FIELD TO THE
3303  A906                 LDA #6          LEFT OF THE CURRENT FIELD.
3305  8D0032               STA FIELD
3308  60            UP.EX2 RTS             THEN RETURN
                      ;
                      ;
3309  C920          IF.SP CMP #SPACE       IS IT THE SPACE BAR?
330B  D009                 BNE IF.CR       IF NOT, PERFORM NEXT TEST.
                      ;
330D  EE0532        INC.SL INC SELECT      IF SO, STEP FORWARD THROUGH
3310  D003                 BNE *+5         MEMORY BY INCREMENTING
3312  EE0632               INC SELECT+1    THE POINTER THAT SELECTS
```

```
                          ;                     THE ADDRESS TO BE DISPLAYED.
      3315  60                   RTS            THEN RETURN TO CALLER.
                          ;
                          ;
      3316  C90D          IF.CR  CMP #CR        IS IT THE CARRIAGE RETURN?
      3318  D00C                 BNE IFCHAR     IF NOT, PERFORM NEXT TEST.
                          ;
      331A  AD0532        DEC.SL LDA SELECT     IF SO, STEP BACKWARD THROUGH
      331D  D003                 BNE *+5        MEMORY BY DECREMENTING THE
      331F  CE0632               DEC SELECT+1   POINTER THAT SELECTS THE
      3322  CE0532               DEC SELECT     ADDRESS TO BE DISPLAYED.
      3325  60                   RTS            THEN RETURN.
                          ;
                          ;
      3326  AE0032        IFCHAR LDX FIELD      IS ARROW UNDER CHARACTER
      3329  E002                 CPX #2         FIELD (FIELD 2)?
      332B  D01B                 BNE IF.GO      IF NOT, PERFORM NEXT TEST.
                          ;                     IF SO,
      332D  A8            PUT.SL TAY            STORE THE
      332E  A5FB                 LDA TV.PTR     CHARACTER IN THE CURRENTLY-
      3330  48                   PHA            SELECTED ADDRESS.
      3331  A6FC                 LDX TV.PTR+1   (PRESERVING THE ZERO PAGE.)
      3333  AD0532               LDA SELECT
      3336  85FB                 STA TV.PTR
      3338  AD0632               LDA SELECT+1
      333B  85FC                 STA TV.PTR+1
      333D  98                   TYA
      333E  A000                 LDY #0
      3340  91FB                 STA (TV.PTR),Y
      3342  86FC                 STX TV.PTR+1
      3344  68                   PLA
      3345  85FB                 STA TV.PTR
      3347  60                   RTS            THEN RETURN.
                          ;
                          ;
      3348  C947          IF.GO  CMP #'G'       IS IT 'G' FOR GO?
      334A  D023                 BNE IF.HEX     IF NOT, PERFORM NEXT TEST.
                          ;
      334C  AC0332        GO     LDY REG.Y      IF SO, LOAD REGISTERS
      334F  AE0232               LDX REG.X      FROM REGISTER IMAGES...
      3352  AD0432               LDA REG.P
      3355  48                   PHA
      3356  AD0132               LDA REG.A
      3359  28                   PLP
      335A  206C33               JSR CALLIT     AND CALL SELECTED ADDRESS.
      335D  08                   PHP            WHEN THE SUBROUTINE RETURNS,
      335E  8D0132               STA REG.A      SAVE REGISTER VALUES IN
      3361  8E0232               STX REG.X      REGISTER IMAGES.
      3364  8C0332               STY REG.Y
      3367  68                   PLA
      3368  8D0432               STA REG.P
      336B  60                   RTS            THEN RETURN TO CALLER.
                          ;
                          ;
```

```
336C  6C0532     CALLIT JMP (SELECT)    JSR CALLIT CALLS THE
                  ;                     CURRENTLY-SELECTED ADDRESS,
                  ;                     INDIRECTLY.
                  ;
                  ;
336F  48         IF.HEX PHA             SAVE KEYBOARD CHARACTER.
3370  20D533            JSR BINARY      IS IT ASCII CHAR FOR 0-9 OR
                  ;                     A-F?  IF SO, CONVERT TO BINARY.
                  ;
                  ;
3373  304B              BMI IF.CLR      IF KEYBOARD CHAR WAS N
                  ;                     0-9 OR A-F, PERFORM NEXT
                  ;                     TEST.
                  ;
3375  A8                TAY             PULL KEYBOARD CHARACTER
3376  68                PLA             FROM STACK, WHILE SAVING
3377  98                TYA             BINARY EQUIVALENT IN A AND Y.
                  ;
3378  AE0032            LDX FIELD       IS ARROW UNDER ADDRESS
337B  D014              BNE NOTADR      FIELD (FIELD 0)?
                  ;
337D  A203       ADRFLD LDX #3          SINCE ARROW IS UNDER ADDRESS
337F  18         ADLOOP CLC             FIELD, ROLL HEX DIGIT INTO
3380  0E0532            ASL SELECT      ADDRESS FIELD BY ROLLING IT
3383  2E0632            ROL SELECT+1    IT INTO THE POINTER THAT
3386  CA                DEX             SELECTS THE DISPLAYED
3387  10F6              BPL ADLOOP      ADDRESS.
3389  98                TYA
338A  0D0532            ORA SELECT
338D  8D0532            STA SELECT
3390  60                RTS             THEN RETURN.
                  ;
                  ;
3391  E001       NOTADR CPX #1          IS ARROW UNDER FIELD 1?
3393  D018              BNE REGFLD      IF NOT, IT MUST BE UNDER
                  ;                     A REGISTER FIELD.
                  ;
3395  290F       ROL.SL AND #$0F        ROLL 4 LSB IN A INTO
3397  48                PHA             CURRENTLY-SELECTED BYTE.
3398  209532            JSR GET.SL      GET THE CURRENTLY-SELECTED
339B  0A                ASL A           BYTE AND SHIFT LEFT 4 TIMES...
339C  0A                ASL A
339D  0A                ASL A
339E  0A                ASL A
339F  29F0              AND #$F0
33A1  8DAC33            STA TEMP
33A4  68                PLA
33A5  0DAC33            ORA TEMP
33A8  202D33            JSR PUT.SL      PUT IT IN CURRENTLY-SELECTED
33AB  60                RTS             ADDRESS AND RETURN.
                  ;
33AC  00         TEMP   .BYTE 0
                  ;
                  ;
```

```
                      ;
33AD   CA             REGFLD DEX           THE ARROW MUST BE UNDER A
33AE   CA                    DEX           REGISTER IMAGE: FIELD 3,
33AF   CA                    DEX           4, 5, OR 6.
33B0   A003                  LDY #3
                      ;
33B2   18             RGLOOP CLC           ROLL HEX DIGIT INTO
33B3   1E0132                ASL REGS,X    APPROPRIATE REGISTER IMAGE.
33B6   88                    DEY
33B7   10F9                  BPL RGLOOP
33B9   1D0132                ORA REGS,X
33BC   9D0132                STA REGS,X
33BF   60                    RTS
                      ;
                      ;
33C0   68             IF.CLR PLA           RESTORE KEYBOARD CHARACTER.
33C1   C993                  CMP #CLRKEY   IS IT THE CLEAR SCREEN KEY?
                      ;
33C3   D004                  BNE NOTCLR    IF NOT, PERFORM NEXT TEST.
                      ;
33C5   200031                JSR CLR.TV    IF IT IS, THEN CLEAR THE
33C8   60                    RTS           SCREEN AND RETURN.
                      ;
                      ;
33C9   C951           NOTCLR CMP #'Q'      IS IT 'Q' FOR QUIT?
33CB   D004                  BNE OTHER     IF NOT, PERFORM NEXT TEST.
                      ;
                      ;                    IT IS 'Q' FOR QUIT.  THE
                      ;                    USER WANTS TO RETURN TO THE
                      ;                    CALLER OF THE VISIBLE
                      ;                    MONITOR.  SO LET'S DO THAT:
33CD   68                    PLA           POP UPDATE'S RETURN ADDRESS.
33CE   68                    PLA
                      ;
33CF   28                    PLP           RESTORE INITIAL 6502 FLAGS.
                      ;                    VISMON'S RETURN ADDRESS IS
                      ;                    NOW ON THE STACK.
33D0   60                    RTS           SO RETURN TO CALLER OF
                      ;                    VISMON.  IN THIS WAY,
                      ;                    VISMON CAN BE USED BY ANY
                      ;                    CALLER TO GET AN ADDRESS
                      ;
                      ;                    FROM THE USER.
                      ;
33D1   201030         OTHER  JSR DUMMY     REPLACE THIS CALL TO
                      ;                    DUMMY WITH A CALL TO ANY
                      ;                    SUBROUTINE THAT EXTENDS
                      ;                    FUNCTIONALITY OF THE
                      ;                    VISIBLE MONITOR.
33D4   60                    RTS           THEN RETURN.
                      ;
                      ;
                      ;
                      ;
```

```
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;  ******************************************
                         ;
                         ;          ASCII TO BINARY
                         ;
                         ;  ******************************************
                         ;
                         ;
                         ;
                         ;                    IF ACCUMULATOR HOLDS ASCII
                         ;                    0-9 OR A-F, THIS ROUTINE
                         ;                    RETURNS BINARY EQUIVALENT--
                         ;                    OTHERWISE, IT RETURNS $FF.
                         ;
                         ;
33D5   38               BINARY SEC
33D6   E930                    SBC #$30
33D8   900F                    BCC BAD
33DA   C90A                    CMP #$0A
33DC   900E                    BCC GOOD
33DE   E907                    SBC #7
33E0   C910                    CMP #$10
33E2   B005                    BCS BAD
33E4   38                      SEC
33E5   C90A                    CMP #$0A
33E7   B003                    BCS GOOD
33E9   A9FF             BAD    LDA #$FF
33EB   60                      RTS
                         ;
33EC   A200             GOOD   LDX #0
33EE   60                      RTS
```

```
CROSS REFERENCE LISTING:

ADLOOP  337F      ADRFLD  337D      ARROW   3007      BAD     33E9
BINARY  33D5      CALLIT  336C      CLR.TV  3100      CLRKEY  0093
CR      000D      DEC.SL  331A      DUMMY   3010      FIELD   3200
GET.SL  3295      GETKEY  32E0      GO      334C      GOOD    33EC
IF.CLR  33C0      IF.CR   3316      IF.GO   3348      IF.HEX  336F
IF.LFT  32FA      IF.SP   3309      IFCHAR  3326      INC.SL  330D
NEXT.F  32EA      NOTADR  3391      NOTCLR  33C9      OTHER   33D1
PARAMS  3000      PREV.F  32FE      PUT.SL  332D      REG.A   3201
REG.P   3204      REG.X   3202      REG.Y   3203      REGFLD  33AD
REGS    3201      RGLOOP  33B2      ROL.SL  3395      ROMKEY  3008
SELECT  3205      SPACE   0020      TEMP    33AC      TV.PTR  00FB
TVSUBS  3100      UP.EX1  32F9      UP.EX2  3308      UPDATE  32E3
VMSUBS  3200
```

Appendix C4:

Print Utilities

```
;        APPENDIX C4: ASSEMBLER LISTING OF
;                     PRINT UTILITIES
;
;
;
;
;  SEE CHAPTER 7 OF TOP-DOWN ASSEMBLY-LANGUAGE
;  PROGRAMMING FOR YOUR COMMODORE 64 AND VIC-20
;
;                    BY KEN SKIER
;
;
;          COPYRIGHT (C) 1984 BY KENNETH SKIER
;                 LEXINGTON, MASSACHUSETTS
;
;
;
;
;
;
;
;
;
;
; ******************************************
;
;                  CONSTANTS
;
; ******************************************
;
;
;
;
        CR      = $0D    CARRIAGE RETURN.
;
        ETX     = $FF    THIS CHARACTER MUST
                         TERMINATE ANY MESSAGE STRING.
;
;
        LF      = $0A    LINE FEED.
;
        OFF     = 0
;
        ON      = $FF
;
;
;
;
;
;
;
;
;
;
```

```
;  *****************************************
;
;                EXTERNAL ADDRESSES
;
;  *****************************************
;
;
;
;
;
;
;
;
;
;
        PARAMS = $3000
;                      ADDRESS OF SYSTEM DATA BLOCK.
;
;
;
        ROMPRT = PARAMS+$0C
;                      POINTER TO ROM ROUTINE THAT
;                      SENDS CHAR TO SERIAL OUTPUT.
;
;
;
        ROMTVT = PARAMS+$0A
;                      POINTER TO ROM ROUTINE THAT
;                      PRINTS A CHAR TO THE SCREEN.
;
;
;
        USROUT = PARAMS+$0E
;                      POINTER TO USER-WRITTEN
;                      CHARACTER OUTPUT ROUTINE.
;
;
;
;
        TVSUBS = $3100
        ASCII  = TVSUBS+$B6
;
;
;
;
        VMPAGE = $3200
;                      VISIBLE MONITOR STARTING
;                      PAGE
;
        SELECT = VMPAGE+5
        GET.SL = VMPAGE+$95
        INC.SL = VMPAGE+$10D
;
;
```

```
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;  *******************************************
                        ;
                        ;                  VARIABLES
                        ;
                        ;  *******************************************
                        ;
                        ;
                        ;
                        ;
                        ;
0000    = 3400                   * = $3400
                        ;
3400    00              PRINTR .BYTE OFF       PRINTER OUTPUT FLAG.
                        ;
3401    FF              TVT    .BYTE ON        TVT OUTPUT FLAG.
                        ;
                        ;
3402    00              USER   .BYTE OFF       OUTPUT FLAG FOR USER-
                        ;                      PROVIDED OUTPUT SUBROUTINE.
                        ;
3403    00              CHAR   .BYTE 0         CHARACTER MOST RECENTLY
                        ;                      PRINTED BY PR.CHR.
                        ;                      CHAR=00 MEANS PR.CHR HAS
                        ;                      NEVER PRINTED A CHARACTER.
                        ;
                        ;
3404    00              REPEAT .BYTE 0         THIS BYTE IS USED AS A
                        ;                      COUNTER BY SPACES, CHARS,
                        ;                      AND CR.LFS.
                        ;
                        ;
3405    00              TEMP.X .BYTE 0         DATA CELL: USED BY PR.MSG.
                        ;
                        ;
3406    0000            RETURN .WORD 0         THIS POINTER IS USED BY
                        ;                      PUSHSL AND POP.SL.
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;  *******************************************
                        ;
                        ;          DEVICE SELECT SUBROUTINES
                        ;
                        ;  *******************************************
                        ;
```

```
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
3408  A9FF        TVT.ON LDA #ON      SELECT SCREEN FOR OUTPUT
340A  8D0134             STA TVT      BY SETTING ITS DEVICE FLAG.
340D  60                 RTS
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
340E  A900        TVTOFF LDA #OFF     DE-SELECT SCREEN FOR
3410  8D0134             STA TVT      OUTPUT BY CLEARING ITS
3413  60                 RTS          DEVICE FLAG.
                  ;
                  ;
                  ;
                  ;
                  ;
3414  A9FF        PR.ON  LDA #ON      SELECT PRINTER FOR OUTPUT
3416  8D0034             STA PRINTR   BY SETTING ITS DEVICE FLAG.
3419  60                 RTS
                  ;
                  ;
                  ;
                  ;
                  ;
341A  A900        PR.OFF LDA #OFF     DE-SELECT PRINTER FOR OUTPUT
341C  8D0034             STA PRINTR   BY CLEARING ITS DEVICE FLAG.
341F  60                 RTS
                  ;
                  ;
                  ;
                  ;
                  ;
3420  A9FF        USR.ON LDA #ON      SELECT USER-WRITTEN
3422  8D0234             STA USER     SUBROUTINE BY SETTING
3425  60                 RTS          USER'S DEVICE FLAG.
                  ;
                  ;
                  ;
                  ;
                  ;
3426  A900        USROFF LDA #OFF     DE-SELECT USER-WRITTEN
3428  8D0234             STA USER     OUTPUT SUBROUTINE BY
342B  60                 RTS          CLEARING ITS DEVICE FLAG.
                  ;
                  ;
                  ;
                  ;
```

```
                ;
342C  200834    ALL.ON JSR TVT.ON     SELECT ALL OUTPUT DEVICES
342F  201434           JSR PR.ON      BY SELECTING EACH OUTPUT
3432  202034           JSR USR.ON     DEVICE INDIVIDUALLY.
3435  60               RTS
                ;
                ;
                ;
                ;
                ;
3436  200E34    ALLOFF JSR TVTOFF     DE-SELECT ALL OUTPUT DEVICES
3439  201A34           JSR PR.OFF     BY DE-SELECTING EACH ONE
343C  202634           JSR USROFF     INDIVIDUALLY.
343F  60               RTS
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;  *******************************************
                ;
                ;        A GENERAL CHARACTER PRINT ROUTINE
                ;
                ;  *******************************************
                ;
                ;
                ;
                ;
                ;
                ;      PRINT CHARACTER IN ACCUMULATOR ON
                ;      ALL CURRENTLY-SELECTED OUTPUT DEVICES.
                ;
                ;
                ;
3440  C900      PR.CHR CMP #0         TEST CHARACTER.
3442  F024             BEQ EXIT       IF IT'S A NULL, RETURN
                ;                     WITHOUT PRINTING IT.
3444  8D0334           STA CHAR       SAVE CHARACTER.
                ;
3447  AD0134           LDA TVT        IS SCREEN SELECTED?
344A  F006             BEQ IF.PR      IF NOT, TEST NEXT DEVICE.
                ;
344C  AD0334           LDA CHAR       IF SO, SEND CHARACTER
344F  206934           JSR SEND.1     INDIRECTLY TO SYSTEM'S
                ;                     TVT OUTPUT ROUTINE.
                ;
                ;
3452  AD0034    IF.PR  LDA PRINTR     IS PRINTER SELECTED?
3455  F006             BEQ IF.USR     IF NOT, TEST NEXT DEVICE.
```

```
                          ;
3457   AD0334                   LDA CHAR       IF SO, SEND CHARACTER
345A   206C34                   JSR SEND.2     INDIRECTLY TO SYSTEM'S
                          ;                    PRINTER DRIVER.
                          ;
                          ;
345D   AD0234             IF.USR LDA USER      IS USER-WRITTEN OUTPUT
                          ;                    SUBROUTINE SELECTED?
3460   F006                     BEQ EXIT       IF NOT, RETURN.
                          ;
3462   AD0334                   LDA CHAR       IF SO, SEND CHARACTER
3465   206F34                   JSR SEND.3     INDIRECTLY TO USER-WRITTEN
                          ;                    SUBROUTINE.
                          ;
3468   60                 EXIT   RTS           RETURN TO CALLER.
                          ;
                          ;
                          ;
                          ;         VECTORED SUBROUTINE CALLS
                          ;
                          ;
                          ;
3469   6C0A30             SEND.1 JMP (ROMTVT)
                          ;
346C   6C0C30             SEND.2 JMP (ROMPRT)
                          ;
346F   6C0E30             SEND.3 JMP (USROUT)
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ; *******************************************
                          ;
                          ;     SPECIALIZED CHARACTER OUTPUT ROUTINES
                          ;
                          ; *******************************************
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;         PRINT A CARRIAGE RETURN-LINE FEED
                          ;
                          ;
3472   A90D               CR.LF  LDA #CR       SEND A CARRIAGE RETURN
3474   204034                   JSR PR.CHR
3477   A90A                     LDA #LF        AND A LINE-FEED TO ALL
3479   204034                   JSR PR.CHR     CURRENTLY-SELECTED DEVICES.
347C   60                       RTS            THEN RETURN.
                          ;
                          ;
```

```
                      ;
                      ;
                      ;
                      ;       PRINT A SPACE:
                      ;
                      ;
                      ;
347D  A920            SPACE  LDA #$20      LOAD ACCUMULATOR WITH AN
347F  204034                 JSR PR.CHR    ASCII SPACE AND PRINT IT.
3482  60                     RTS           THEN RETURN.
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ; ******************************************
                      ;
                      ;                 PRINT BYTE
                      ;
                      ; ******************************************
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;       PR.BYT OUTPUTS THE ACCUMULATOR, IN HEX,
                      ;   TO ALL CURRENTLY-SELECTED DEVICES.
                      ;
                      ;
                      ;
3483  48              PR.BYT PHA           SAVE BYTE.
3484  4A                     LSR A         DETERMINE ASCII FOR 4 MSB...
3485  4A                     LSR A
3486  4A                     LSR A
3487  4A                     LSR A
3488  20B631                 JSR ASCII     ...IN THE BYTE.
348B  204034                 JSR PR.CHR    PRINT THAT ASCII CHAR TO
                      ;                    CURRENT DEVICE(S).
348E  68                     PLA           DETERMINE ASCII FOR 4 LSB
348F  20B631                 JSR ASCII     IN THE ORIGINAL BYTE.
3492  204034                 JSR PR.CHR    PRINT THAT CHARACTER.
3495  60                     RTS           RETURN TO CALLER.
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
```

```
                        ;
                        ; *****************************************
                        ;
                        ;           REPETITIVE CHARACTER OUTPUT
                        ;
                        ; *****************************************
                        ;
                        ;
                        ;
                        ;      PRINT X SPACES:
                        ;
                        ;
3496   A920            SPACES LDA #$20       LOAD A WITH ASCII SPACE.
                        ;
                        ;                    PRINT IT X TIMES:
                        ;
                        ;
                        ;
                        ;
                        ;      PRINT X CHARACTERS:
                        ;
                        ;
                        ;
3498   8E0434          CHARS  STX REPEAT     PRINT CHAR IN A X TIMES.
349B   48              RPLOOP PHA            SAVE CHAR TO BE REPEATED.
349C   AE0434                 LDX REPEAT     REPEAT COUNTER TIMED OUT?
349F   F00A                   BEQ RPTEND     IF SO, EXIT.  IF NOT,
34A1   CE0434                 DEC REPEAT     DECREMENT REPEAT COUNTER.
34A4   204034                 JSR PR.CHR     PRINT CHARACTER.
                        ;
34A7   68                     PLA            RESTORE CHARACTER TO A.
34A8   18                     CLC            LOOP BACK TO PRINT IT
34A9   90F0                   BCC RPLOOP     AGAIN IF NECESSARY.
                        ;
34AB   68              RPTEND PLA            CLEAN UP STACK AND
34AC   60                     RTS            RETURN TO CALLER.
                        ;
                        ;
                        ;
                        ;      PRINT X NEWLINES
                        ;
                        ;
34AD   8E0434          CR.LFS STX REPEAT     INITIALIZE REPEAT COUNTER.
34B0   AE0434          CRLOOP LDX REPEAT     EXIT IF REPEAT COUNTER
34B3   F009                   BEQ END.CR     HAS TIMED OUT.
34B5   CE0434                 DEC REPEAT     DECREMENT REPEAT COUNTER.
34B8   207234                 JSR CR.LF      PRINT A CARRIAGE RETURN
                        ;                    AND A LINE FEED.
34BB   18                     CLC            LOOP BACK TO SEE IF DONE
34BC   90F2                   BCC CRLOOP     YET.
                        ;
34BE   60              END.CR RTS            RETURN TO CALLER.
                        ;
                        ;
```

```
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ; ******************************************
                         ;
                         ;                  PRINT A MESSAGE
                         ;
                         ; ******************************************
                         ;
                         ;
                         ;
                         ;
                         ;          Xth POINTER IN ZERO PAGE
                         ;           POINTS TO THE MESSAGE.
                         ;
                         ;
34BF   8E0534          PR.MSG STX TEMP.X       SAVE X REGISTER, WHICH
                         ;                     SPECIFIES MESSAGE POINTER.
                         ;
34C2   B501                   LDA 1,X          SAVE MESSAGE POINTER.
34C4   48                     PHA
34C5   B500                   LDA 0,X
34C7   48                     PHA
                         ;
34C8   AE0534          LOOP   LDX TEMP.X       RESTORE ORIGINAL X, SO IT
                         ;                     SPECIFIES MESSAGE POINTER.
34CB   A100                   LDA (0,X)        GET NEXT CHARACTER FROM
34CD   C9FF                   CMP #ETX         MESSAGE.  IS MESSAGE OVER?
34CF   F00C                   BEQ MSGEND       IF SO, HANDLE END OF MESSAGE.
                         ;
34D1   F600                   INC 0,X          IF NOT, INCREMENT POINTER.
34D3   D002                   BNE NEXT         SO IT POINTS TO NEXT
34D5   F601                   INC 1,X          CHARACTER IN MESSAGE.
34D7   204034          NEXT   JSR PR.CHR       PRINT THE CHARACTER.
34DA   18                     CLC              LOOP BACK FOR NEXT
34DB   90EB                   BCC LOOP         CHARACTER...
                         ;
                         ;
34DD   68              MSGEND PLA              RESTORE ORIGINAL MESSAGE
34DE   9500                   STA 0,X          POINTER.
34E0   68                     PLA
34E1   9501                   STA 1,X
34E3   60                     RTS              RETURN TO CALLER, WITH
                         ;                     MESSAGE POINTER PRESERVED.
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
```

```
                        ;
                        ;
                        ;
                        ; ***************************************
                        ;
                        ;          PRINT THE FOLLOWING TEXT
                        ;
                        ; ***************************************
                        ;
                        ;
                        ;
                        ;
                        ;
34E4   68               PRINT: PLA              PULL RETURN ADDRESS FROM
34E5   AA                      TAX              STACK AND SAVE IT IN X AND
34E6   68                      PLA              Y REGISTERS.
34E7   A8                      TAY
                        ;
34E8   201235                  JSR PUSHSL       SAVE THE SELECT POINTER.
34EB   8E0532                  STX SELECT       SET SELECT EQUAL TO
34EE   8C0632                  STY SELECT+1     RETURN ADDRESS.
                        ;
                        ;
34F1   200D33                  JSR INC.SL       ADVANCE SELECT TO STX.
                        ;
34F4   200D33           NEXTCH JSR INC.SL       SELECT NEXT CHARACTER.
34F7   209532                  JSR GET.SL       GET IT.
34FA   C9FF                    CMP #ETX         IS IT END OF MESSAGE?
34FC   F006                    BEQ ENDIT        IF SO, RETURN.
34FE   204034                  JSR PR.CHR       IF NOT, PRINT CHARACTER.
3501   18                      CLC              LOOP BACK FOR NEXT
3502   90F0                    BCC NEXTCH       CHARACTER...
                        ;
                        ;
3504   AE0532           ENDIT  LDX SELECT
3507   AC0632                  LDY SELECT+1
350A   202B35                  JSR POP.SL       RESTORE SELECT POINTER.
350D   98                      TYA              PUSH ADDRESS OF ETX ONTO
350E   48                      PHA
350F   8A                      TXA              ...THE STACK.
3510   48                      PHA
3511   60                      RTS              RETURN (TO BYTE IMMEDIATELY
                        ;                       FOLLOWING THE ETX.)
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ; ***************************************
```

```
6400                          ;
6410                          ;          SAVE, RESTORE SELECT POINTER
6420                          ;
6430                          ; ******************************************
6440                          ;
6450                          ;
6460                          ;
6470                          ;
6480                          ;
6490   3512  68               PUSHSL PLA            PULL RETURN ADDRESS FROM
6500   3513  8D0634                  STA RETURN     STACK AND SAVE IT IN RETURN.
6510   3516  68                      PLA
6520   3517  8D0734                  STA RETURN+1
6530                          ;
6540                          ;
6550   351A  AD0632                  LDA SELECT+1   PUSH SELECT POINTER ONTO
6560   351D  48                      PHA            THE STACK.
6570   351E  AD0532                  LDA SELECT
6580   3521  48                      PHA
6590                          ;
6600                          ;
6610   3522  AD0734                  LDA RETURN+1   PUSH RETURN ADDRESS BACK
6620   3525  48                      PHA            ON THE STACK.
6630   3526  AD0634                  LDA RETURN
6640   3529  48                      PHA
6650                          ;
6660                          ;
6670   352A  60                      RTS            RETURN TO CALLER.  CALLER
6680                          ;                     WILL FIND SELECT ON STACK.
6690                          ;
6700                          ;
6710                          ;
6720                          ;
6730                          ;
6740                          ;
6750                          ;
6760                          ;
6770   352B  68               POP.SL PLA            SAVE RETURN ADDRESS.
6780   352C  8D0634                  STA RETURN
6790   352F  68                      PLA
6800   3530  8D0734                  STA RETURN+1
6810                          ;
6820                          ;
6830   3533  68                      PLA            LOAD SELECT FROM STACK
6840   3534  8D0532                  STA SELECT
6850   3537  68                      PLA
6860   3538  8D0632                  STA SELECT+1
6870                          ;
6880                          ;
6890   353B  AD0734                  LDA RETURN+1   PLACE RETURN ADDRESS BACK
6900   353E  48                      PHA            ON STACK.
6910   353F  AD0634                  LDA RETURN
6920   3542  48                      PHA
6930                          ;
```

```
                    ;
3543  60                  RTS          RETURN TO CALLER.
                    ;
```

CROSS REFERENCE LISTING:

ALL.ON	342C	ALLOFF	3436	ASCII	31B6	CHAR	3403
CHARS	3498	CR	000D	CR.LF	3472	CR.LFS	34AD
CRLOOP	34B0	END.CR	34BE	ENDIT	3504	ETX	00FF
EXIT	3468	GET.SL	3295	IF.PR	3452	IF.USR	345D
INC.SL	330D	LF	000A	LOOP	34C8	MSGEND	34DD
NEXT	34D7	NEXTCH	34F4	OFF	0000	ON	00FF
PARAMS	3000	POP.SL	352B	PR.BYT	3483	PR.CHR	3440
PR.MSG	34BF	PR.OFF	341A	PR.ON	3414	PRINT	34E4
PRINTR	3400	PUSHSL	3512	REPEAT	3404	RETURN	3406
ROMPRT	300C	ROMTVT	300A	RPLOOP	349B	RPTEND	34AB
SELECT	3205	SEND.1	3469	SEND.2	346C	SEND.3	346F
SPACE	347D	SPACES	3496	TEMP.X	3405	TVSUBS	3100
TVT	3401	TVT.ON	3408	TVTOFF	340E	USER	3402
USR.ON	3420	USROFF	3426	USROUT	300E	VMPAGE	3200

Appendix C5:

Two Hexdump Tools

```
;       APPENDIX C5: ASSEMBLER LISTING OF
;                    TWO HEXDUMP TOOLS
;
;
;
; SEE CHAPTER 8 OF TOP-DOWN ASSEMBLY LANGUAGE
; PROGRAMMING FOR YOUR COMMODORE 64 AND VIC-20
;
;
;                   KEN SKIER
;
;
;        COPYRIGHT (C) 1984 BY KENNETH SKIER
;              LEXINGTON, MASSACHUSETTS
;
;
;
;
;
;
;
;
;
;
;
;********************************************
;
;                   CONSTANTS
;
;********************************************
;
;
;
;
;
     CR     = $0D   CARRIAGE RETURN.
;
     LF     = $0A   LINE FEED.
;
     TEX    = $7F   THIS CHARACTER MUST START
;                   ANY MESSAGE.
;
     ETX    = $FF   THIS CHARACTER MUST END
;                   ANY MESSAGE.
;
;
;
;
;
;
;
;
;
;
```

```
1540                ;
1550                ;
1560                ;
1570                ; ******************************************
1580                ;
1590                ;           EXTERNAL ADDRESSES
1600                ;
1610                ; ******************************************
1620                ;
1630                ;
1640                ;
1650                ;
1660                ;
1670                ;
1680                ;
1690                ;
1700                ;
1710                ;
1720                      TVSUBS = $3100
1730                                    STARTING PAGE OF DISPLAY
1740                ;                   CODE.
1750
1760                      CLR.TV = TVSUBS
1770                      ASCII  = TVSUBS+$B6
1780                ;
1790                ;
1800                      VMPAGE = $3200
1810                                    STARTING PAGE OF VISIBLE
1820                ;                   MONITOR CODE.
1830                ;
1840                      SELECT = VMPAGE+5
1850                      VISMON = VMPAGE+7
1860                      GET.SL = VMPAGE+$95
1870                      INC.SL = VMPAGE+$10D
1880                ;
1890                ;
1900                      PRPAGE = $3400
1910                                    STARTING PAGE OF PRINT
1920                ;                   UTILITIES.
1930                ;
1940                      TVT.ON = PRPAGE+8
1950                      TVTOFF = PRPAGE+$0E
1960                      PR.ON  = PRPAGE+$14
1970                      PR.OFF = PRPAGE+$1A
1980                      PR.CHR = PRPAGE+$40
1990                      CR.LF  = PRPAGE+$72
2000                      CR.LFS = PRPAGE+$AD
2010                      SPACE  = PRPAGE+$7D
2020                      SPACES = PRPAGE+$96
2030                      PR.BYT = PRPAGE+$83
2040                      PRINT: = PRPAGE+$E4
2050                      PUSHSL = PRPAGE+$112
2060                      POP.SL = PRPAGE+$12B
2070                ;
```

```
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;  ******************************************
                    ;
                    ;                  VARIABLES
                    ;
                    ;  ******************************************
                    ;
                    ;
                    ;
0000   = 3550             * =   $3550
                    ;
                    ;
                    ;
                    ;
                    ;
3550   00           COUNTR .BYTE 0        THIS BYTE COUNTS THE LINES
                    ;                     DUMPED BY TVDUMP.
                    ;
3551   07           MASK   .BYTE 7        FOR OUTPUT A  (SUITABLE FOR
                    ;                     C-64.)  USE "DB 3" FOR
                    ;                     OUTPUT B (SUITABLE FOR VIC
                    ;                     VIC-20.)
                    ;
3552   0000         SA     .WORD 0        POINTER TO START OF MEMORY
                    ;                     TO BE DUMPED BY PRDUMP.
                    ;
3554   FFFF         EA     .WORD $FFFF    POINTER TO LAST BYTE TO
                    ;                     BE DUMPED BY PRDUMP.
                    ;
                    ;
3556   00           COLUMN .BYTE 0        DATA CELL: USED BY PRLINE
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;  ******************************************
                    ;
                    ;                  TVDUMP
                    ;
                    ;  ******************************************
                    ;
```

```
                         ;
                         ;
                         ;
                         ;
3557  200834      TVDUMP JSR TVT.ON     SELECT TVT AS OUTPUT DEVICE.
355A  A904               LDA #4         SET COUNTR TO NUMBER OF
355C  8D5035             STA COUNTR     LINES TO BE DUMPED.
                         ;
355F  AD0532             LDA SELECT     SET SELECT TO BEGINNING OF
3562  29F0               AND #$F0       A HEX LINE (16 BYTES) BY
3564  8D0532             STA SELECT     ZEROING 4 LSB IN SELECT.
                         ;
                         ;
3567  209B35      DUMPLN JSR PR.ADR     PRINT THE SELECTED ADDRESS.
356A  207D34             JSR SPACE      PRINT A SPACE ON THE SCREEN.
                         ;
                         ;
356D  207D34      DMPBYT JSR SPACE      PRINT A SPACE ON THE SCREEN.
3570  209435             JSR DUMPSL     DUMP SELECTED BYTE.
3573  200D33             JSR INC.SL     SELECT NEXT BYTE.
                         ;
3576  AD0532             LDA SELECT     IS IT THE BEGINNING OF A
3579  2D5135             AND MASK       NEW SCREEN LINE?  (ARE 3 LSB,
                         ;              0, FOR OUTPUT A, OR 2 LSB 0,
                         ;              FOR OUTPUT B?)
                         ;
357C  D0EF               BNE DMPBYT     IF NOT, DUMP NEXT BYTE...
357E  207234             JSR CR.LF      IF SO, ADVANCE TO A NEW LINE
                         ;              ON SCREEN.
                         ;
3581  AD0532             LDA SELECT     DOES THIS ADDRESS MARK THE
3584  290F               AND #$0F       BEGINNING OF A NEW HEX LINE?
                         ;              (ARE 4 LSB 0?)
                         ;
3586  D003               BNE IFDONE
3588  207234             JSR CR.LF      IF SO, ADVANCE TO A NEW
                         ;              LINE ON SCREEN.
                         ;
358B  CE5035      IFDONE DEC COUNTR     DUMPED LAST LINE YET?
358E  D0D7               BNE DUMPLN     IF NOT, DUMP NEXT LINE.
                         ;
                         ;
3590  200E34             JSR TVTOFF     DE-SELECT TVT AS OUTPUT
                         ;              DEVICE.
                         ;
3593  60                 RTS            RETURN TO CALLER.
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
```

```
                          ;
                          ; ********************************************
                          ;
                          ;              DUMP SELECTED BYTE
                          ;
                          ; ********************************************
                          ;
                          ;
                          ;
                          ;
                          ;
3594   209532      DUMPSL JSR GET.SL      GET CURRENTLY-SELECTED BYTE
3597   208334             JSR PR.BYT      AND PRINT IT IN HEX FORMAT.
359A   60                 RTS             RETURN TO CALLER.
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ; ********************************************
                          ;
                          ;            PRINT SELECTED ADDRESS
                          ;
                          ; ********************************************
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
359B   AD0632      PR.ADR LDA SELECT+1    FIRST PRINT THE HIGH BYTE...
359E   208334             JSR PR.BYT
35A1   AD0532             LDA SELECT      ...THEN PRINT THE LOW BYTE.
35A4   208334             JSR PR.BYT
35A7   60                 RTS
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ; ********************************************
                          ;
                          ;              PRINTING HEXDUMP
                          ;
```

```
                            ; *******************************************
                            ;
                            ;
                            ;
                            ;
                            ;
35A8   20C335       PRDUMP JSR TITLE        DISPLAY THE TITLE
35AB   20E335              JSR SETADS       LET USER SET START ADDRESS
                            ;               AND END ADDRESS OF MEMORY TO
                            ;               BE DUMPED.  (SETADS RETURNS
                            ;               WITH SELECT EQUAL TO EA.)
35AE   209A37              JSR GOTOSA       SET SELECT EQUAL TO SA.
35B1   201434              JSR PR.ON        SELECT PRINTER FOR OUTPUT.
                            ;
35B4   20E536              JSR HEADER       OUTPUT HEXDUMP HEADER.
                            ;
                            ;
35B7   203C37       HXLOOP JSR PRLINE       DUMP ONE LINE.
35BA   10FB                BPL HXLOOP       DUMPED LAST LINE? IF NOT,
                            ;               DUMP NEXT LINE.
                            ;
35BC   207234              JSR CR.LF        IF SO, GO TO A NEW LINE.
                            ;
35BF   201A34              JSR PR.OFF       DE-SELECT PRINTER FOR OUTPUT.
                            ;
35C2   60                  RTS              RETURN TO CALLER.
                            ;
                            ;
                            ;
                            ;
                            ;
                            ;
                            ;
                            ;
                            ;
                            ;
                            ; *******************************************
                            ;
                            ;       PRINT THE HEXDUMP TITLE ON SCREEN
                            ;
                            ; *******************************************
                            ;
                            ;
                            ;
                            ;
35C3   200834       TITLE  JSR TVT.ON       SELECT SCREEN FOR OUTPUT.
35C6   201A34              JSR PR.OFF       DE-SELECT PRINTER.
35C9   20E434              JSR PRINT:       OUTPUT THE FOLLOWING TEXT:
35CC   7F                  .BYTE TEX        TEXT STRING MUST START
                            ;               WITH A START OF TEXT CHAR.
35CD   0D5052494E          .BYTE CR,'PRINTING HEXDUMP',CR,LF,LF     ;
       54494E4720
       4845584455
       4D500D0A0A
```

```
35E1  FF                    .BYTE ETX       TEXT STRING MUST END WITH
                      ;                     AN END OF TEXT CHARACTER.
35E2  60                    RTS             RETURN TO CALLER.
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ; *******************************************
                      ;
                      ;        LET USER SET STARTING ADDRESS AND
                      ;        END ADDRESS OF A BLOCK OF MEMORY:
                      ;
                      ; *******************************************
                      ;
                      ;
                      ;
                      ;
                      ;
35E3  200834          SETADS JSR TVT.ON     SELECT SCREEN FOR OUTPUT
35E6  20E434                JSR PRINT:      PUT PROMPT ON SCREEN:
35E9  7F                    .BYTE TEX
35EA  0D0A534554            .BYTE CR,LF,'SET STARTING ADDRESS ' ;
      2053544152
      54494E4720
      4144445245
      535320
3601  414E442050            .BYTE 'AND PRESS "Q".'              ;
      5245535320
      2251222E
360F  FF                    .BYTE ETX
3610  200732                JSR VISMON      CALL VISIBLE MONITOR, SO
                      ;                     USER CAN SELECT START ADDRESS
                      ;                     OF THE BLOCK.
                      ;
3613  206136                JSR SAHERE      SET SA EQUAL TO SELECT
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;                     HAVING SET THE START ADDRESS,
                      ;                     SA, LET'S SET THE END ADDRESS,
                      ;                     EA.
                      ;
                      ;
                      ;
                      ;
```

```
                           ;
3616  200834      SET.EA JSR 1VT.ON       SELECT SCREEN FOR OUTPUT.
3619  20E434             JSR PRINT:       PUT PROMPT ON SCREEN:
361C  7F                 .BYTE TEX
361D  0D0A534554         .BYTE CR,LF,'SET END ADDRESS '   ;
      20454E4420
      4144445245
      535320
362F  414E442050         .BYTE 'AND PRESS "Q".',ETX       ;
      5245535320
      2251222EFF
                         ;
363E  200732             JSR VISMON       LET USER SELECT END ADDRESS.
                         ;
3641  38                 SEC              IF USER TRIED TO SET AN
3642  AD0632             LDA SELECT+1     ADDRESS LESS THAN THE
3645  CD5335             CMP SA+1         STARTING ADDRESS,
3648  9024               BCC TOOLOW       MAKE USER DO IT OVER.
364A  D008               BNE EAHERE       IF SELECT>SA, SET EA EQUAL TO
                         ;                SELECT.  THAT WILL MAKE EA>SA.
                         ;
                         ;
                         ;
364C  AD0532             LDA SELECT
364F  CD5235             CMP SA
3652  901A               BCC TOOLOW
                         ;
                         ;
                         ;
                         ;
3654  AD0632      EAHERE LDA SELECT+1     SET EA EQUAL TO SELECT.
3657  8D5535             STA EA+1
365A  AD0532             LDA SELECT
365D  8D5435             STA EA
3660  60                 RTS              RETURN WITH EA SET BY CALLER
                         ;                (JSR EAHERE); EA SET BY USER
                         ;                (JSR SET.EA); OR SA AND EA
                         ;                SET BY USER (JSR SETADS).
                         ;
3661  AD0632      SAHERE LDA SELECT+1     SET SA EQUAL TO SELECT.
3664  8D5335             STA SA+1
3667  AD0532             LDA SELECT
366A  8D5235             STA SA
366D  60                 RTS              RETURN.
                         ;
366E  20E434      TOOLOW JSR PRINT:       SINCE USER SET ENDING
                         ;                ADDRESS TOO LOW, PUT A
                         ;                PROMPT ON THE SCREEN:
3671  7F                 .BYTE TEX
3672  0D0A0A0A20         .BYTE CR,LF,LF,LF,' ERROR                ;
      4552524F52
      21212120
3680  454E442041         .BYTE 'END ADDRESS LESS THAN START ADDRESS,';
      4444524553
```

```
5320          53204C4553
5330          5320544841
5340          4E20535441
5350          5254204144
5360          4452455353
5370          2C
5380   36A4   2057484943              .BYTE ' WHICH IS ',ETX
5390          4820495320
5400          FF
5410   36AF   20B536                  JSR PR.SA        PRINT START ADDRESS.
5420                          ;
5430   36B2   4C1636                  JMP SET.EA       AND LET THE USER SET A
5440                          ;                        NEW END ADDRESS.
5450                          ;
5460                          ;
5470                          ;
5480                          ;
5490                          ;
5500                          ;
5510                          ;
5520                          ;
5530                          ;
5540                          ; ********************************************
5550                          ;
5560                          ;          PRINT START ADDRESS
5570                          ;
5580                          ; ********************************************
5590                          ;
5600                          ;
5610                          ;
5620                          ;
5630   36B5   A924            PR.SA   LDA #'$'         PRINT A DOLLAR SIGN, TO
5640   36B7   204034                  JSR PR.CHR       INDICATE HEXADECIMAL.
5650                          ;
5660   36BA   AD5335                  LDA SA+1         PRINT HIGH BYTE OF START
5670   36BD   208334                  JSR PR.BYT       ADDRESS.
5680                          ;
5690   36C0   AD5235                  LDA SA           PRINT LOW BYTE OF START
5700   36C3   208334                  JSR PR.BYT
5710   36C6   60                      RTS              RETURN TO CALLER.
5720                          ;
5730                          ;
5740                          ;
5750                          ;
5760                          ;
5770                          ;
5780                          ;
5790                          ; ********************************************
5800                          ;
5810                          ;          PRINT END ADDRESS
5820                          ;
5830                          ; ********************************************
5840                          ;
5850                          ;
```

```
                        ;
                        ;
                        ;
36C7  A924              PR.EA  LDA #'$'      PRINT A DOLLAR SIGN, TO
36C9  204034                   JSR PR.CHR    INDICATE HEXADECIMAL.
36CC  AD5535                   LDA EA+1      PRINT HIGH BYTE OF END
36CF  208334                   JSR PR.BYT    ADDRESS.
36D2  AD5435                   LDA EA        PRINT LOW BYTE OF END
36D5  208334                   JSR PR.BYT    ADDRESS.
36D8  60                       RTS           RETURN TO CALLER.
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ; ****************************************
                        ;
                        ;          PRINT RANGE OF ADDRESSES
                        ;
                        ; ****************************************
                        ;
                        ;
                        ;
                        ;
                        ;
36D9  20B536            RANGE  JSR PR.SA     PRINT STARTING ADDRESS.
36DC  A92D                     LDA #'-'      PRINT A HYPHEN.
36DE  204034                   JSR PR.CHR
36E1  20C736                   JSR PR.EA     PRINT END ADDRESS.
36E4  60                       RTS           RETURN TO CALLER.
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ; ****************************************
                        ;
                        ;               PRINT HEADER
                        ;
                        ; ****************************************
                        ;
                        ;
                        ;
                        ;
```

```
                           ;
36E5  20E434         HEADER JSR PRINT:
36E8  7F                    .BYTE TEX
36E9  0D0A0A4455            .BYTE CR,LF,LF,'DUMPING ';
      4D50494E47
      20
36F4  FF                    .BYTE ETX
36F5  20D936                JSR RANGE
36F8  207234                JSR CR.LF
36FB  20E434                JSR PRINT:
36FE  7F0A0A                .BYTE TEX,LF,LF
3701  2020202020            .BYTE '      0  1  2  3  4  5  6  7  ';
      2020203020
      2031202032
      2020332020
      3420203520
      2036202037
      2020
3721  3820203920            .BYTE '8  9  A  B  C  D  E  F'          ;
      2041202042
      2020432020
      4420204520
      2046
3737  0D0A0AFF              .BYTE CR,LF,LF,ETX
373B  60                    RTS
                           ;
                           ;
                           ;
                           ;
                           ;
                           ;
                           ;
                           ;
                           ;
                           ;
                           ;  *******************************************
                           ;
                           ;           DUMP ONE LINE TO PRINTER
                           ;
                           ;  *******************************************
                           ;
                           ;
                           ;
                           ;
                           ;
373C  207234         PRLINE JSR CR.LF
373F  AD0532                LDA SELECT      DETERMINE STARTING COLUMN.
3742  48                    PHA             FOR THIS DUMP.
3743  290F                  AND #$0F
3745  8D5635                STA COLUMN      NOW COLUMN HOLDS NUMBER OF
                           ;                HEX COLUMN IN WHICH WE DUMP
                           ;                THE FIRST BYTE.
3748  68                    PLA             SET SELECT TO BEGINNING OF
3749  29F0                  AND #$F0        A HEX LINE.
```

```
374B  8D0532           STA SELECT
374E  209B35           JSR PR.ADR    PRINT LINE'S START ADDRESS.
3751  A203             LDX #3        SPACE 3 TIMES--TO THE
3753  209634           JSR SPACES    FIRST HEX COLUMN.
                ;
                ;
3756  AD5635           LDA COLUMN    DO WE DUMP FROM THE FIRST
                ;                    HEX COLUMN?
3759  F00D             BEQ COL.OK    IF SO, WERE AT THE CORRECT
                ;                    COLUMN NOW.
                ;
375B  A203      LOOP   LDX #3        IF NOT, SPACE 3 TIMES FOR
375D  209634           JSR SPACES    EACH BYTE NOT DUMPED.
3760  200D33           JSR INC.SL
3763  CE5635           DEC COLUMN
3766  D0F3             BNE LOOP
                ;
3768  209435    COL.OK JSR DUMPSL    DUMP SELECTED BYTE.
376B  207D34           JSR SPACE     SPACE ONCE.
376E  207D37           JSR NEXTSL    SELECT NEXT BYTE
                ;
3771  3009             BMI EXIT      MINUS MEANS WE'VE DUMPED
                ;                    THROUGH TO THE END ADDRESS.
                ;
                ;
3773  AD0532    NOT.EA LDA SELECT    DUMPED ENTIRE LINE?
3776  290F             AND #$0F      (ARE 4 LSB OF SELECT 0?)
3778  C900             CMP #0        IF SO, WE'VE DUMPED THE
                ;                    ENTIRE LINE.  IF NOT,
377A  D0EC             BNE COL.OK    SELECT NEXT BYTE AND DUMP IT.
377C  60        EXIT   RTS           RETURN MINUS IF EA DUMPED...
                ;                    OR PLUS IF EA NOT DUMPED.
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; *******************************************
                ;
                ;       SELECT NEXT BYTE (IF < END ADDRESS)
                ;
                ; *******************************************
                ;
                ;
                ;
                ;
                ;
377D  38        NEXTSL SEC
377E  AD0632           LDA SELECT+1  HIGH BYTE OF SELECT LESS
```

```
3781   CD5535            CMP EA+1       THAN HIGH BYTE OF EA?
3784   900B              BCC SL.OK      IF SO, SELECT<END ADDRESS.
3786   D00F              BNE NO.INC     IF SELECT>EA, DON'T
                 ;                      INCREMENT SELECT.
                 ;
3788   38                SEC            SELECT IS IN SAME PAGE AS EA.
3789   AD0532            LDA SELECT
378C   CD5435            CMP EA
378F   B006              BCS NO.INC
                 ;
3791   200D33    SL.OK   JSR INC.SL     SINCE SELECT NOT GREATER THAN
                 ;                      EA, WE MAY INCREMENT SELECT.
                 ;
3794   A900              LDA #0         SET "INCREMENTED" RETURN
3796   60                RTS            CODE AND RETURN.
                 ;
3797   A9FF      NO.INC  LDA #$FF       SET "NO INCREMENT" RETURN
3799   60                RTS            CODE AND RETURN.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; *******************************************
                 ;
                 ;                SELECT START ADDRESS
                 ;
                 ; *******************************************
                 ;
                 ;
                 ;
                 ;
                 ;
379A   AD5235    GOTOSA  LDA SA         SET SELECT EQUAL TO SA.
379D   8D0532            STA SELECT
37A0   AD5335            LDA SA+1
37A3   8D0632            STA SELECT+1
37A6   60                RTS            AND RETURN.
```

CROSS REFERENCE LISTING:

ASCII	31B6	CLR.TV	3100	COL.OK	3768	COLUMN	3556
COUNTR	3550	CR	000D	CR.LF	3472	CR.LFS	34AD
DMPBYT	356D	DUMPLN	3567	DUMPSL	3594	EA	3554
EAHERE	3654	ETX	00FF	EXIT	377C	GET.SL	3295
GOTOSA	379A	HEADER	36E5	HXLOOP	35B7	IFDONE	358B
INC.SL	330D	LF	000A	LOOP	375B	MASK	3551
NEXTSL	377D	NO.INC	3797	NOT.EA	3773	POP.SL	352B
PR.ADR	359B	PR.BYT	3483	PR.CHR	3440	PR.EA	36C7
PR.OFF	341A	PR.ON	3414	PR.SA	36B5	PRDUMP	35A8
PRINT	34E4	PRLINE	373C	PRPAGE	3400	PUSHSL	3512
RANGE	36D9	SA	3552	SAHERE	3661	SELECT	3205
SET.EA	3616	SETADS	35E3	SL.OK	3791	SPACE	347D
SPACES	3496	TEX	007F	TITLE	35C3	TOOLOW	366E
TVDUMP	3557	TVSUBS	3100	TVT.ON	3408	TVTOFF	340E
VISMON	3207	VMPAGE	3200				

Appendix C6:

Table-Driven Disassembler (Top Level and Utility Subroutines)

```
;       APPENDIX C6: ASSEMBLER LISTING OF
;                    TABLE-DRIVEN DISASSEMBLER
;
;          TOP-LEVEL AND UTILITY SUBROUTINES
;
;
;
;
; SEE CHAPTER 9 OF TOP-DOWN ASSEMBLY LANGUAGE
; PROGRAMMING FOR YOUR COMMODORE 64 AND VIC-20
;
;
;                    BY KEN SKIER
;
;
;          COPYRIGHT (C) 1984 BY KENNETH SKIER
;                 LEXINGTON, MASSACHUSETTS
;
;
;
;
;
;
;
;
;
;
;
; ********************************************
;
;                    CONSTANTS
;
; ********************************************
;
;
;
;
;
        CR      = $0D   CARRIAGE RETURN.
;
        LF      = $0A   LINE FEED.
;
;
        TEX     = $7F   THIS CHARACTER MUST START
;                       ANY MESSAGE.
;
        ETX     = $FF   THIS CHARACTER MUST END
;                       ANY MESSAGE.
;
;
;
;
;
```

```
;
;
;
;
; ******************************************
;
;                EXTERNAL ADDRESSES
;
; ******************************************
;
;
;
;
;
        PARAMS = $3000
;                       SYSTEM PARAMETERS
;
        TVCOLS = PARAMS+3
;
;
        VMPAGE = $3200
;                       STARTING PAGE OF VISIBLE
;                       MONITOR CODE.

        SELECT = VMPAGE+5
        VISMON = VMPAGE+7
        GET.SL = VMPAGE+$95
        INC.SL = VMPAGE+$10D
        DEC.SL = VMPAGE+$11A
;
;
        PRPAGE = $3400
;                       STARTING PAGE OF PRINT
;                       UTILITIES.

        TVT.ON = PRPAGE+8
        TVTOFF = PRPAGE+$0E
        PR.ON  = PRPAGE+$14
        PR.OFF = PRPAGE+$1A
        PR.CHR = PRPAGE+$40
        CR.LF  = PRPAGE+$72
        SPACE  = PRPAGE+$7D
        SPACES = PRPAGE+$96
        PR.BYT = PRPAGE+$83
        PRINT: = PRPAGE+$E4
        PUSHSL = PRPAGE+$112
        POP.SL = PRPAGE+$12B
;
;
        HEX.PG = $3500
;                       ADDRESS OF PAGE IN WHICH
;                       HEXDUMP CODE STARTS.
;
        SA     = HEX.PG+$52
```

```
                          EA      = SA+2
                          DUMPSL  = HEX.PG+$94
                          PR.ADR  = HEX.PG+$9B
                          RANGE   = HEX.PG+$1D9
                          SETADS  = HEX.PG+$E3
                          NEXTSL  = HEX.PG+$27D
                          GOTOSA  = HEX.PG+$29A
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;          DISASSEMBLER TABLES:
                     ;
                     ;
                     ;
                          DSPAGE = $3900
                                          STARTING PAGE OF DISASSEMBLER
                     ;
                     ;
                          SUBS    = DSPAGE+$226          ; <-------CHANGED FROM "+ $21B" 9/29/83
                          MNAMES  = DSPAGE+$250
                          MCODES  = DSPAGE+$300
                          MODES   = DSPAGE+$400
                     ;
                     ;
                     ;
                     ;
                     ; ********************************************
                     ;
                     ;                  VARIABLES
                     ;
                     ; ********************************************
                     ;
                     ;
                     ;
0000    = 3900                * =  DSPAGE
                     ;
                     ;
                     ;
                     ;
                     ;
3900  05             DISLNS .BYTE 5        NUMBER OF LINES TO BE
                                           DISASSEMBLED BY TV.DIS.
                     ;
                     ;
3901  00             LINUM  .BYTE 0        DATA CELL: USED BY TV.DIS.
                     ;
3902  00             LETTER .BYTE 0        COUNTS LETTERS PRINTED IN
                                           A MNEMONIC.  USED BY MNEMON.
                     ;
                     ;
3903  00             TEMP.X .BYTE 0        DATA CELL USED BY MNEMON.
                     ;
3904  0000           SUBPTR .WORD 0        POINTER TO A SUBROUTINE.
                                           SET, USED BY MODE.X
                     ;
```

```
                      ;
3906  00              OPBYTS .BYTE 0        DATA CELL: USED BY FINISH.
                      ;
3907  00              OPCHRS .BYTE 0        DATA CELL: USED BY FINISH.
                      ;
3908  0B              ADRCOL .BYTE 11       STARTING COLUMN FOR ADDRESS
                      ;                     FIELD.
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ; ********************************************
                      ;
                      ;              TV-DISASSEMBLER
                      ;
                      ; ********************************************
                      ;
                      ;
                      ;
                      ;
                      ;
3909  200834          TV.DIS JSR TVT.ON     SELECT SCREEN FOR OUTPUT.
390C  AD0039                 LDA DISLNS     INITIALIZE LINE COUNTER WITH
390F  8D0139                 STA LINUM      # OF LINES TO DISASSEMBLE.
                      ;
3912  A9FF                   LDA #$FF       SET END ADDRESS TO $FFFF,
3914  8D5435                 STA EA         SO NEXTSL WILL ALWAYS
3917  8D5535                 STA EA+1       INCREMENT SELECT POINTER.
391A  207234                 JSR CR.LF      ADVANCE TO A NEW LINE.
                      ;
391D  207D39          TVLOOP JSR DSLINE     DISASSEMBLE ONE LINE.
3920  CE0139                 DEC LINUM      DONE LAST LINE YET?
3923  D0F8                   BNE TVLOOP     IF NOT, DO NEXT ONE.
3925  60                     RTS            IF SO, RETURN.
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ; ********************************************
                      ;
                      ;            PRINTING DISASSEMBLER
                      ;
                      ; ********************************************
                      ;
```

```
                                 ;
                                 ;
                                 ;
                                 ;
3926  201A34            PR.DIS JSR PR.OFF      DE-SELECT PRINTER
3929  200834                   JSR TVT.ON      SELECT SCREEN FOR OUTPUT.
392C  20E434                   JSR PRINT:      DISPLAY TITLE.
392F  7F0D0A                   .BYTE TEX,CR,LF
3932  2020202020               .BYTE '      PRINTING DISASSEMBLER.' ;
      5052494E54
      494E472044
      4953415353
      454D424C45
      522E
394D  0D0AFF                   .BYTE CR,LF,ETX
                         ;
3950  20E335                   JSR SETADS      LET USER SET START, END
                         ;                      ADDRESSES OF MEMORY TO BE
                         ;                      DISASSEMBLED.
3953  201434                   JSR PR.ON       SELECT PRINTER FOR OUTPUT.
3956  20E434                   JSR PRINT:
3959  7F0D0A                   .BYTE TEX,CR,LF
395C  4449534153               .BYTE 'DISASSEMBLING ';
      53454D424C
      494E4720
396A  FF                       .BYTE ETX
396B  20D936                   JSR RANGE       PRINT RANGE OF MEMORY TO
                         ;                      BE DISASSEMBLED.
396E  209A37                   JSR GOTOSA      MAKE SELECT POINT TO START
                         ;                      OF BLOCK.
                         ;
3971  207234                   JSR CR.LF       ADVANCE TO A NEW LINE.
3974  207D39            PRLOOP JSR DSLINE      DISASSEMBLE ONE LINE.
3977  10FB                     BPL PRLOOP      IF IT WASN'T THE LAST LINE,
                         ;                      DISASSEMBLE THE NEXT ONE.
                         ;
                         ;
3979  201A34                   JSR PR.OFF      DE-SELECT PRINTER FOR OUTPUT.
                         ;
397C  60                       RTS             RETURN TO CALLER.
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;
                         ;  *******************************************
                         ;
                         ;              DISASSEMBLE ONE LINE.
                         ;
```

```
                       ; *******************************************
                       ;
                       ;
                       ;
                       ;
                       ;
397D  209532     DSLINE JSR GET.SL       GET CURRENTLY-SELECTED BYTE.
3980  48                PHA              SAVE IT ON STACK.
3981  209239            JSR MNEMON       PRINT MNEMONIC REPRESENTED
                       ;                 BY THAT OPCODE.
3984  207D34            JSR SPACE        SPACE ONCE.
3987  68                PLA              RESTORE OPCODE.
3988  20AF39            JSR OPERND       PRINT OPERAND REQUIRED BY
                       ;                 THAT OPCODE.
398B  20013A            JSR FINISH       FINISH THE LINE BY PRINTING
                       ;                 FIELDS 3-6.  FINISH LEAVES
                       ;                 SELECT POINTING TO LAST
                       ;                 BYTE OF INSTRUCTION.
                       ;
398E  207D37            JSR NEXTSL       SELECT NEXT BYTE, IF
                       ;                 SELECT<EA.
3991  60                RTS              RETURN W/RETURNCODE FROM
                       ;                 NEXTSL.  SELECT POINTS TO
                       ;                 NEXT OPCODE, OR SELECT
                       ;                 EQUALS EA.
                       ;
                       ;
                       ;
                       ;
                       ;
                       ;
                       ;
                       ; *******************************************
                       ;
                       ;                PRINT MNEMONIC
                       ;
                       ; *******************************************
                       ;
                       ;
                       ;
                       ;
                       ;
3992  A203       MNEMON LDX #3           WE'LL PRINT THREE LETTERS.
3994  8E0239            STX LETTER
3997  AA                TAX              PREPARE TO USE OPCODE AS AN
                       ;                 INDEX.
                       ;
3998  BD003C            LDA MCODES,X     LOOK UP MNEMONIC CODE FOR
                       ;                 THAT OPCODE.  MCODES IS
                       ;                 TABLE OF MNEMONIC CODES.
                       ;
399B  AA                TAX              PREPARE TO USE THAT MNEMONIC
                       ;                 CODE AS AN INDEX.
399C  BD503B     MNLOOP LDA MNAMES,X     GET A MNEMONIC CHARACTER.
```

```
                      ;                          (MNAMES IS A LIST OF
                      ;                          MNEMONIC NAMES.)
                      ;
399F   8E0339                STX TEMP.X          SAVE X-REGISTER, SINCE
                      ;                          PRINTING MAY CHANGE X.
39A2   204034                JSR PR.CHR          PRINT THE MNEMONIC CHARACTER.
39A5   AE0339                LDX TEMP.X          RESTORE X,
39A8   E8                    INX                 ADJUST INDEX FOR NEXT LETTER.
39A9   CE0239                DEC LETTER          PRINTED 3 LETTERS YET?
39AC   D0EE                  BNE MNLOOP          IF NOT, PRINT NEXT ONE.
39AE   60                    RTS                 IF SO, RETURN TO CALLER.
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;  ******************************************
                      ;
                      ;                   PRINT OPERAND
                      ;
                      ;  ******************************************
                      ;
                      ;
                      ;
                      ;
                      ;
39AF   AA             OPERND TAX                 LOOK UP ADDRESSING MODE
39B0   BD003D                LDA MODES,X         CODE FOR THIS OPCODE.
                      ;
39B3   AA                    TAX                 X NOW INDICATES ADDRESSING
                      ;                          MODE.
                      ;
39B4   20B839                JSR MODE.X          HANDLE THAT ADDRESSING MODE.
39B7   60                    RTS                 RETURN TO CALLER.
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;  ******************************************
                      ;
                      ;          HANDLE ADDRESSING MODE "X"
                      ;
                      ;  ******************************************
```

```
4770                       ;
4780                       ;
4790                       ;
4800                       ;
4810                       ;
4820                       ;
4830  39B8  BD263B         MODE.X LDA SUBS,X     GET LOW BYTE OF Xth POINTER
4840  39BB  8D0439                STA SUBPTR     IN TABLE OF SUBROUTINE
4850                       ;                     POINTERS.
4860  39BE  E8                    INX            ADJUST INDEX FOR NEXT BYTE.
4870  39BF  BD263B                LDA SUBS,X     GET HIGH BYTE OF POINTER.
4880  39C2  8D0539                STA SUBPTR+1
4890  39C5  6C0439                JMP (SUBPTR)   JUMP TO SUBROUTINE SPECIFIED
4900                       ;                     BY SUBROUTINE POINTER.
4910                       ;                     THAT SUBROUTINE WILL RETURN
4920                       ;                     TO THE CALLER OF MODE.X,
4930                       ;                     NOT TO MODE.X ITSELF.
4940                       ;
4950                       ;
4960                       ;
4970                       ;
4980                       ;
4990                       ;
5000                       ;
5010                       ;
5020                       ;
5030                       ;
5040                       ; ******************************************
5050                       ;
5060                       ;          DISASSEMBLER UTILITIES
5070                       ;
5080                       ; ******************************************
5090                       ;
5100                       ;
5110                       ;
5120                       ;
5130                       ;
5140                       ;          PRINT ONE-BYTE OPERAND
5150                       ;
5160                       ;
5170                       ;
5180  39C8  200D33         ONEBYT JSR INC.SL     ADVANCE TO BYTE FOLLOWING
5190                       ;                     OPCODE.
5200  39CB  209435                JSR DUMPSL     DUMP THAT BYTE.
5210  39CE  60                    RTS            RETURN TO CALLER.
5220                       ;
5230                       ;
5240                       ;
5250                       ;
5260                       ;
5270                       ;          PRINT TWO-BYTE OPERAND:
5280                       ;
5290                       ;
5300                       ;
```

```
5310  39CF  200D33        TWOBYT JSR INC.SL      ADVANCE TO FIRST BYTE OF
5320                       ;                     OPERAND.
5330  39D2  209532               JSR GET.SL      LOAD THAT BYTE INTO ACC.
5340  39D5  48                   PHA             SAVE IT.
5350  39D6  200D33               JSR INC.SL      ADVANCE TO 2ND BYTE OF
5360                       ;                     OPERAND.
5370  39D9  209435               JSR DUMPSL      DUMP IT.
5380  39DC  68                   PLA             RESTORE FIRST BYTE TO ACC.
5390  39DD  208334               JSR PR.BYT      DUMP IT.
5400  39E0  60                   RTS             RETURN TO CALLER.
5410                       ;
5420                       ;
5430                       ;
5440                       ;
5450                       ;
5460                       ;     PRINT LEFT, RIGHT PARENTHESES
5470                       ;
5480                       ;
5490                       ;
5500  39E1  A928          LPAREN LDA #$28        PRINT LEFT PAREN.
5510  39E3  D002                 BNE SENDIT
5520                       ;
5530                       ;
5540  39E5  A929          RPAREN LDA #$29        PRINT RIGHT PAREN.
5550                       ;
5560  39E7  204034        SENDIT JSR PR.CHR
5570  39EA  60                   RTS
5580                       ;
5590                       ;
5600                       ;
5610                       ;
5620                       ;
5630                       ;       PRINT A COMMA AND AN "X"
5640                       ;
5650                       ;
5660                       ;
5670  39EB  A92C          XINDEX LDA #','
5680  39ED  204034               JSR PR.CHR      PRINT A COMMA.
5690  39F0  A958                 LDA #'X'
5700  39F2  204034               JSR PR.CHR      PRINT AN "X".
5710  39F5  60                   RTS
5720                       ;
5730                       ;
5740                       ;
5750                       ;
5760                       ;
5770                       ;       PRINT A COMMA AND A "Y"
5780                       ;
5790                       ;
5800                       ;
5810  39F6  A92C          YINDEX LDA #','
5820  39F8  204034               JSR PR.CHR      PRINT COMMA.
5830  39FB  A959                 LDA #'Y'
5840  39FD  204034               JSR PR.CHR      PRINT A "Y".
```

```
3A00  60                RTS
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; *******************************************
                ;
                ;       FINISH THE LINE
                ;
                ; *******************************************
                ;
                ;
                ;
                ;           NOTE:     EVERY ADDRESSING MODE
                ;                     SUBROUTINE MUST END BY
                ;                     SETTING X EQUAL TO THE
                ;                     NUMBER OF BYTES IN THE
                ;                     OPERAND, AND ACC EQUAL
                ;                     TO THE NUMBER OF
                ;                     CHARACTERS IN OPERAND.
                ;
                ;
3A01  8D0739    FINISH STA OPCHRS     SAVE THE LENGTH OF THE
3A04  8E0639           STX OPBYTS     OPERAND, IN CHARACTERS AND
                ;                     IN BYTES.  0 MEANS NO
                ;                     OPERAND.
                ;
3A07  CA               DEX            IF NECESSARY, DECREMENT THE
                ;                     SELECT POINTER SO IT POINTS
3A08  3006             BMI SEL.OK     TO THE OPCODE.
3A0A  201A33    LOOP.1 JSR DEC.SL
3A0D  CA               DEX
3A0E  10FA             BPL LOOP.1
                ;
                                      NOW SELECT POINTS TO OPCODE.
                ;
3A10  08        SEL.OK PHP            SAVE CALLER'S DECIMAL FLAG.
3A11  D8               CLD            PREPARE FOR BINARY ADDITION.
                ;
3A12  38               SEC            SPACE OVER TO THE COLUMN
3A13  AD0839           LDA ADRCOL     FOR THE ADDRESS FIELD:
3A16  E904             SBC #4         OPERAND FIELD STARTED IN
                ;                     COLUMN 4...
3A18  ED0739           SBC OPCHRS     AND INCLUDES OPCHRS
                ;                     CHARACTERS.
3A1B  28               PLP            RESTORE CALLER'S DECIMAL FLAG
3A1C  AA               TAX
3A1D  209634           JSR SPACES     PRINT ENOUGH SPACES TO
```

```
6390                       ;                        REACH ADDRESS COLUMN.
6400                       ;
6410  3A20  209B35               JSR PR.ADR         PRINT ADDRESS OF OPCODE.
6420  3A23  207D34               JSR SPACE          SPACE AFTER OPCODE'S ADDRESS.
6430                       ;
6440  3A26  209435         LOOP.2 JSR DUMPSL        DUMP SELECTED BYTE.
6450
6460  3A29  AD0330               LDA TVCOLS         IS SCREEN < 24 COLUMNS WIDE?
6470  3A2C  38                   SEC
6480  3A2D  C918                 CMP #24
6490  3A2F  9003                 BCC DUMPED         IF SO, DON'T SPACE AGAIN.
6500                       ;
6510                       ;                        SCREEN IS > 24 COLUMNS WIDE.
6520
6530  3A31  207D34               JSR SPACE          SO SPACE AFTER DUMPING BYTE.
6540                       ;
6550  3A34                 DUMPED                   WE'VE DUMPED SELECTED BYTE.
6560  3A34  200D33               JSR INC.SL         SELECT NEXT BYTE.
6570  3A37  CE0639               DEC OPBYTS         DUMPED LAST BYTE IN
6580                       ;                        INSTRUCTION?
6590  3A3A  10EA                 BPL LOOP.2         IF NOT, DUMP NEXT BYTE.
6600  3A3C  201A33               JSR DEC.SL         BACK UP SELECT, SO IT POINTS
6610                       ;                        TO LAST BYTE IN OPERAND.
6620                       ;
6630                       ;
6640                       ;                        IF SO, GO TO A NEW LINE:
6650                       ;
6660  3A3F  207234         FINEND JSR CR.LF         HAVING DISASSEMBLED ONE LINE,
6670                       ;                        GO TO A NEW LINE.
6680  3A42  60                   RTS                RETURN TO CALLER.
                           ;
```

```
                    CROSS REFERENCE LISTING:


ADRCOL  3908      CR      000D      CR.LF   3472      DEC.SL  331A
DISLNS  3900      DSLINE  397D      DSPAGE  3900      DUMPED  3A34
DUMPSL  3594      EA      3554      ETX     00FF      FINEND  3A3F
FINISH  3A01      GET.SL  3295      GOTOSA  379A      HEX.PG  3500
INC.SL  330D      LETTER  3902      LF      000A      LINUM   3901
LOOP.1  3A0A      LOOP.2  3A26      LPAREN  39E1      MCODES  3C00
MNAMES  3B50      MNEMON  3992      MNLOOP  399C      MODE.X  39B8
MODES   3D00      NEXTSL  377D      ONEBYT  39C8      OPBYTS  3906
OPCHRS  3907      OPERND  39AF      PARAMS  3000      POP.SL  352B
PR.ADR  359B      PR.BYT  3483      PR.CHR  3440      PR.DIS  3926
PR.OFF  341A      PR.ON   3414      PRINT   34E4      PRLOOP  3974
PRPAGE  3400      PUSHSL  3512      RANGE   36D9      RPAREN  39E5
SA      3552      SEL.OK  3A10      SELECT  3205      SENDIT  39E7
SETADS  35E3      SPACE   347D      SPACES  3496      SUBPTR  3904
SUBS    3B26      TEMP.X  3903      TEX     007F      TV.DIS  3909
TVCOLS  3003      TVLOOP  391D      TVT.ON  3408      TVTOFF  340E
TWOBYT  39CF      VISMON  3207      VMPAGE  3200      XINDEX  39EB
YINDEX  39F6
```

Appendix C7:

Table-Driven Disassembler (Addressing Mode Subroutines)

```
;       APPENDIX C7: ASSEMBLER LISTING OF
;                    TABLE-DRIVEN DISASSEMBLER:
;
;                    ADDRESSING MODE SUBROUTINES
;
;
;
;
;
;
;
; SEE CHAPTER 9 OF TOP-DOWN ASSEMBLY LANGUAGE
; PROGRAMMING FOR YOUR COMMODORE 64 AND VIC-20
;
;
;                    BY KEN SKIER
;
;
;            COPYRIGHT (C) 1984 BY KENNETH SKIER
;                 LEXINGTON, MASSACHUSETTS
;
;
;
;
;
;
;
;
;
;
;
;
;
;
;
; *******************************************
;
;                  CONSTANTS
;
; *******************************************
;
;
;
;
;
     CR      = $0D   CARRIAGE RETURN.
;
     LF      = $0A   LINE FEED.
;
;
     TEX     = $7F   THIS CHARACTER MUST START
;                    ANY MESSAGE.
;
```

```
        ETX    = $FF    THIS CHARACTER MUST END
;                       ANY MESSAGE.
;
;
;
;
;
;
;
;
;
;
;
;
;
; ******************************************
;
;                 EXTERNAL ADDRESSES
;
; ******************************************
;
;
;
;
;
;
;
;
;
;
;
        VMPAGE = $3200
;                       STARTING PAGE OF VISIBLE
;                       MONITOR CODE.
;
        SELECT = VMPAGE+5
        VISMON = VMPAGE+7
        GET.SL = VMPAGE+$95
        INC.SL = VMPAGE+$10D
        DEC.SL = VMPAGE+$11A
;
;
        PRPAGE = $3400
;                       STARTING PAGE OF PRINT
;                       UTILITIES.
;
        PR.CHR = PRPAGE+$40
        CR.LF  = PRPAGE+$72
        SPACE  = PRPAGE+$7D
        SPACES = PRPAGE+$96
        PR.BYT = PRPAGE+$83
        PUSHSL = PRPAGE+$112
        POP.SL = PRPAGE+$12B
;
```

```
                          ;
                                HEX.PG = $3500
                          ;                    ADDRESS OF PAGE IN WHICH
                          ;                    HEXDUMP CODE STARTS.
                          ;
                                PR.ADR = HEX.PG+$9B
                                NEXTSL = HEX.PG+$27D
                          ;
                          ;
                                DSPAGE = $3900
                          ;                    START OF DISASSEMBLER CODE.
                          ;
                                ONEBYT = DSPAGE+$C8
                                TWOBYT = DSPAGE+$CF
                                LPAREN = DSPAGE+$E1
                                RPAREN = DSPAGE+$E5
                                XINDEX = DSPAGE+$EB
                                YINDEX = DSPAGE+$F6
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
0000    = 3A50                  * =  DSPAGE+$150
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ; ********************************************
                          ;
                          ;          ADDRESSING MODE SUBROUTINES
                          ;
                          ; ********************************************
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;          ABSOLUTE MODE
                          ;
                          ;
                          ;
3A50    20CF39            ABSLUT JSR TWOBYT    PRINT A TWO-BYTE OPERAND.
3A53    A202                    LDX #2         OPERAND HAS TWO BYTES...
```

```
3A55   A904                 LDA #4         ...AND FOUR CHARACTERS.
3A57   60                   RTS            RETURN TO CALLER.
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;         ABSOLUTE,X MODE
                    ;
                    ;
                    ;
3A58   20503A       ABS.X   JSR ABSLUT
3A5B   20EB39               JSR XINDEX     PRINT A COMMA AND AN "X".
3A5E   A202                 LDX #2         OPERAND HAS 2 BYTES...
3A60   A906                 LDA #6         ...AND SIX CHARACTERS.
3A62   60                   RTS            RETURN TO CALLER.
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;         ABSOLUTE,Y MODE
                    ;
                    ;
                    ;
3A63   20503A       ABS.Y   JSR ABSLUT
3A66   20F639               JSR YINDEX
3A69   A202                 LDX #2
3A6B   A906                 LDA #6
3A6D   60                   RTS
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;         ACCUMULATOR MODE
                    ;
                    ;
3A6E   A941         ACC     LDA #'A'       PRINT THE LETTER "A".
3A70   204034               JSR PR.CHR
3A73   A200                 LDX #0         OPERAND HAS NO BYTES...
3A75   A901                 LDA #1         ...AND ONE CHARACTER.
3A77   60                   RTS            RETURN TO CALLER.
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;         IMPLIED MODE
                    ;
                    ;
                    ;
3A78   A200         IMPLID  LDX #0         OPERAND HAS NO BYTES...
3A7A   A900                 LDA #0         ...AND NO CHARACTERS.
```

```
3A7C   60              RTS
                ;
                ;
                ;
                ;
                ;
                ;          IMMEDIATE MODE
                ;
                ;
                ;
3A7D   A923     IMMEDT LDA #'#'         PRINT A "#" CHARACTER.
3A7F   204034          JSR PR.CHR
                ;
3A82   A924            LDA #'$'         PRINT A DOLLAR SIGN TO
3A84   204034          JSR PR.CHR       INDICATE HEXADECIMAL.
3A87   20C839          JSR ONEBYT       PRINT ONE-BYTE OPERAND IN
                ;                       HEXADECIMAL FORMAT.
3A8A   A201            LDX #1           OPERAND HAS ONE BYTE...
3A8C   A904            LDA #4           ...AND FOUR CHARACTERS.
3A8E   60              RTS              RETURN TO CALLER.
                ;
                ;
                ;
                ;
                ;
                ;          INDIRECT MODE
                ;
                ;
                ;
3A8F   20E139   INDRCT JSR LPAREN       PRINT LEFT PARENTHESIS.
3A92   20503A          JSR ABSLUT       PRINT TWO-BYTE OPERAND.
3A95   20E539          JSR RPAREN       PRINT RIGHT PARENTHESIS.
3A98   A906            LDA #6           A HOLDS NUMBER OF CHARACTERS
                ;                       IN OPERAND.
3A9A   A202            LDX #2           X HOLDS NUMBER OF BYTES IN
                ;                       OPERAND.
3A9C   60              RTS              RETURN TO CALLER.
                ;
                ;
                ;
                ;
                ;
                ;          INDIRECT,X MODE
                ;
                ;
                ;
3A9D   20E139   IND.X  JSR LPAREN
3AA0   20F33A          JSR ZERO.X       PRINT A ZERO PAGE ADDRESS,
                ;                       A COMMA, AND THE LETTER "X".
3AA3   20E539          JSR RPAREN
3AA6   A201            LDX #1           ONE BYTE IN OPERAND.
3AA8   A906            LDA #6           6 CHARACTERS IN OPERAND.
3AAA   60              RTS
                ;
```

```
                          ;
                          ;
                          ;
                          ;
                          ;      INDIRECT,Y MODE
                          ;
                          ;
                          ;
3AAB   20E139     IND.Y   JSR LPAREN
3AAE   20EB3A             JSR ZEROPG     PRINT A ZERO PAGE ADDRESS.
3AB1   20E539             JSR RPAREN
3AB4   20F639             JSR YINDEX     PRINT A COMMA AND A "Y".
3AB7   A201               LDX #1         OPERAND HAS 1 BYTE...
3AB9   A906               LDA #6         ...AND 8 CHARACTERS.
3ABB   60                 RTS
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;         RELATIVE MODE
                          ;
                          ;
                          ;
3ABC   200D33     RELATV  JSR INC.SL     SELECT NEXT BYTE.
3ABF   201235             JSR PUSHSL     SAVE SELECT POINTER ON STACK.
3AC2   209532             JSR GET.SL     GET OPERAND BYTE.
3AC5   48                 PHA            SAVE IT ON STACK.
3AC6   200D33             JSR INC.SL     INCREMENT SELECT POINTER
                          ;              SO IT POINTS TO NEXT OPCODE.
                          ;              (RELATIVE BRANCHES ARE
                          ;              RELATIVE TO NEXT OPCODE.)
3AC9   68                 PLA            RESTORE OPERAND BYTE TO ACC.
3ACA   C900               CMP #0         IS IT PLUS OR MINUS?
3ACC   1003               BPL FORWRD     IF PLUS, IT MEANS A FORWARD
                          ;              BRANCH.
                          ;
                          ;              OPERAND IS MINUS, SO WE'LL
                          ;              BRANCH BACKWARD.
3ACE   CE0632             DEC SELECT+1   BRANCHING BACKWARD IS LIKE
                          ;              BRANCHING FORWARD FROM ONE
                          ;              PAGE LOWER IN MEMORY.
                          ;
                          ;
3AD1   08         FORWRD  PHP            SAVE CALLER'S DECIMAL FLAG.
3AD2   D8                 CLD            CLEAR DECIMAL MODE, FOR
                          ;              BINARY ADDITION.
3AD3   18                 CLC            PREPARE TO ADD.
3AD4   6D0532             ADC SELECT     ADD OPERAND BYTE TO SELECT.
3AD7   9003               BCC RELEND
3AD9   EE0632             INC SELECT+1
3ADC   8D0532     RELEND  STA SELECT     NOW SELECT POINTS TO ADDRESS
                          ;              SPECIFIED BY RELATIVE
                          ;              BRANCH INSTRUCTION.
```

```
3ADF   28                   PLP             RESTORE CALLER'S DECIMAL
                     ;                      FLAG.
3AE0   209B35               JSR PR.ADR      PRINT ADDRESS SPECIFIED
                     ;                      BY INSTRUCTION.
3AE3   202B35               JSR POP.SL      MAKE SELECT POINT TO
                     ;                      ADDRESS OF OPERAND.
3AE6   A201                 LDX #1          OPERAND HAD ONE BYTE...
3AE8   A904                 LDA #4          AND FOUR CHARACTERS.
3AEA   60                   RTS             RETURN TO CALLER.
                     ;
                     ;
                     ;
                     ;
                     ;          ZERO PAGE MODE
                     ;
                     ;
                     ;
                     ;
3AEB   20C839        ZEROPG JSR ONEBYT      PRINT ONE-BYTE OPERAND.
3AEE   A201                 LDX #1          OPERAND HAS ONE BYTE...
3AF0   A902                 LDA #2          ...AND TWO CHARACTERS.
3AF2   60                   RTS
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;          ZERO PAGE, X  MODE
                     ;
                     ;
                     ;
3AF3   20EB3A        ZERO.X JSR ZEROPG      PRINT THE ZERO PAGE ADDRESS.
3AF6   20EB39               JSR XINDEX      PRINT A COMMA AND AN "X".
3AF9   A201                 LDX #1          OPERAND HAS 1 BYTE...
3AFB   A904                 LDA #4          ...AND FOUR CHARACTERS.
3AFD   60                   RTS             RETURN TO CALLER.
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;          ZERO PAGE ,Y MODE
                     ;
                     ;
                     ;
3AFE   20EB3A        ZERO.Y JSR ZEROPG
3B01   20F639               JSR YINDEX
3B04   A201                 LDX #1
3B06   A904                 LDA #4
3B08   60                   RTS
                     ;
                     ;
                     ;
                     ;
```

```
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ; ******************************************
                        ;
                        ;          A PSEUDO-ADDRESSING MODE
                        ;        FOR EMBEDDED TEXT: TEXT MODE.
                        ;
                        ; ******************************************
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;     THE PSEUDO-OPCODE TEX ($7F) BEGINS ANY
                        ; STRING OF TEXT AND PRINT CONTROL CHARACTERS.
                        ; THE PSEUDO-TEXT CHARACTER ETX ($FF) ENDS ANY
                        ; SUCH STRING.  TEX HAS A PSEUDO-ADDRESSING
                        ; MODE: TEXT MODE.  IN TEXT MODE, WE PRINT THE
                        ; STRING AND RETURN, WITHOUT DUMPING THE LINE
                        ; IN HEX.  THE STRING MAY BE OF ANY LENGTH.
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
3B09   68               TXMODE PLA             POP RETURN ADDRESS TO
3B0A   68                      PLA             OPERND.
                        ;
3B0B   68                      PLA             POP RETURN ADDRESS TO
3B0C   68                      PLA             DSLINE.
                        ;
                        ;                      NOW DSLINE'S CALLER IS ON
                        ;                      THE STACK.
                        ;
                        ;
3B0D   207D37                  JSR NEXTSL      ADVANCE PAST TEX PSEUDO-OP.
3B10   300D                    BMI TXEXIT      RETURN IF REACHED EA.
3B12   209532                  JSR GET.SL      GET THE CHARACTER.
3B15   C9FF                    CMP #ETX        IS IT END OF TEXT?
3B17   F006                    BEQ TXEXIT      IF SO, STRING ENDED.
3B19   204034                  JSR PR.CHR      IF NOT, PRINT CHARACTER.
```

```
3B1C   18                    CLC                BRANCH BACK TO GET NEXT
3B1D   90EE                  BCC TXMODE+4       CHARACTER.
                       ;
                       ;
3B1F   207234          TXEXIT JSR CR.LF         ADVANCE TO A NEW LINE.
3B22   207D37                JSR NEXTSL         ADVANCE TO NEXT OPCODE.
3B25   60                    RTS                RETURN TO CALLER OF DSLINE.
                       ;
                       ;
                       ;
                       ;
                       ;
                       ;
                       ;
                       ;
                       ;
                       ;
                       ; ******************************************
                       ;
                       ;     TABLE OF ADDRESSING MODE SUBROUTINES
                       ;
                       ; ******************************************
                       ;
                       ;
                       ;
                       ;
                       ;
3B26   783A            SUBS  .WORD IMPLID       ADDRESSING MODE 0 IS INVALID,
                       ;                        HENCE IMPLIED.
3B28   6E3A                  .WORD ACC
3B2A   7D3A                  .WORD IMMEDT
3B2C   EB3A                  .WORD ZEROPG
3B2E   F33A                  .WORD ZERO.X
3B30   FE3A                  .WORD ZERO.Y
3B32   503A                  .WORD ABSLUT
3B34   583A                  .WORD ABS.X
3B36   633A                  .WORD ABS.Y
3B38   783A                  .WORD IMPLID
3B3A   BC3A                  .WORD RELATV
3B3C   9D3A                  .WORD IND.X
3B3E   AB3A                  .WORD IND.Y
3B40   8F3A                  .WORD INDRCT
3B42   093B                  .WORD TXMODE
```

```
                    CROSS REFERENCE LISTING:



ABS.X  3A58     ABS.Y  3A63     ABSLUT 3A50     ACC    3A6E
CR     000D     CR.LF  3472     DEC.SL 331A     DSPAGE 3900
ETX    00FF     FORWRD 3AD1     GET.SL 3295     HEX.PG 3500
IMMEDT 3A7D     IMPLID 3A78     INC.SL 330D     IND.X  3A9D
IND.Y  3AAB     INDRCT 3A8F     LF     000A     LPAREN 39E1
NEXTSL 377D     ONEBYT 39C8     POP.SL 352B     PR.ADR 359B
PR.BYT 3483     PR.CHR 3440     PRPAGE 3400     PUSHSL 3512
RELATV 3ABC     RELEND 3ADC     RPAREN 39E5     SELECT 3205
SPACE  347D     SPACES 3496     SUBS   3B26     TEX    007F
TWOBYT 39CF     TXEXIT 3B1F     TXMODE 3B09     VISMON 3207
VMPAGE 3200     XINDEX 39EB     YINDEX 39F6     ZERO.X 3AF3
ZERO.Y 3AFE     ZEROPG 3AEB
```

Appendix C8:

Table-Driven Disassembler (Tables)

```
;        APPENDIX C8: ASSEMBLER LISTING OF
;                     TABLE-DRIVEN DISASSEMBLER
;
;                     TABLES
;
;
;
;
; SEE CHAPTER 9 OF TOP-DOWN ASSEMBLY LANGUAGE
; PROGRAMMING FOR YOUR COMMODORE 64 AND VIC-20
;
;
;                     BY KEN SKIER
;
;
;           COPYRIGHT (C) 1984 BY KENNETH SKIER
;                 LEXINGTON, MASSACHUSETTS
;
;
;
;
;
;
;
;
;
;
;
;********************************************
;
;                   CONSTANTS
;
;********************************************
;
;
;
;
;
;
;
        TEX     = $7F
;                 THIS CHARACTER MUST START
;                 ANY MESSAGE.
;
;
        ETX     = $FF
;                 THIS CHARACTER MUST END
;                 ANY MESSAGE.
;
;
;
;
;
```

```
1540                          ;
1550                          ;
1560                          ;
1570                          ;
1580                          ;
1590                          ;
1600                          ;
1610                          ;
1620                          ;
1630                          ;
1640                          ;
1650                          ;
1660                          ;
1670                               DSPAGE = $3900
1680                          ;                  STARTING PAGE OF DISASSEMBLER
1690                          ;
1700                          ;
1710                          ;
1720                          ;
1730                          ;
1740                          ; ******************************************
1750                          ;
1760                          ;          LIST OF MNEMONICS
1770                          ;
1780                          ; ******************************************
1790                          ;
1800                          ;
1810                          ;
1820   0000   = 3B50               * =  DSPAGE+$250
1830                          ;
1840                          ;
1850                          ;
1860                          ;
1870                          ;
1880   3B50   7F             MNAMES .BYTE TEX      SINCE THIS TABLE IS A
1890                          ;                    STRING OF CHARACTERS, START
1900                          ;                    IT WITH THE TEX PSEUDO-OP.
1910                          ;
1920   3B51   424144               .BYTE 'BAD'
1930   3B54   414443               .BYTE 'ADC'
1940   3B57   414E44               .BYTE 'AND'
1950   3B5A   41534C               .BYTE 'ASL'
1960   3B5D   424343               .BYTE 'BCC'
1970   3B60   424353               .BYTE 'BCS'
1980   3B63   424551               .BYTE 'BEQ'
1990   3B66   424954               .BYTE 'BIT'
2000   3B69   424D49               .BYTE 'BMI'
2010   3B6C   424E45               .BYTE 'BNE'
2020   3B6F   42504C               .BYTE 'BPL'
2030   3B72   42524B               .BYTE 'BRK'
2040   3B75   425643               .BYTE 'BVC'
2050   3B78   425653               .BYTE 'BVS'
2060   3B7B   434C43               .BYTE 'CLC'
2070   3B7E   434C44               .BYTE 'CLD'
```

```
2080   3B81   434C49               .BYTE 'CLI'
2090   3B84   434C56               .BYTE 'CLV'
2100   3B87   434D50               .BYTE 'CMP'
2110   3B8A   435058               .BYTE 'CPX'
2120   3B8D   435059               .BYTE 'CPY'
2130   3B90   444543               .BYTE 'DEC'
2140   3B93   444558               .BYTE 'DEX'
2150   3B96   444559               .BYTE 'DEY'
2160   3B99   454F52               .BYTE 'EOR'
2170   3B9C   494E43               .BYTE 'INC'
2180   3B9F   494E58               .BYTE 'INX'
2190   3BA2   494E59               .BYTE 'INY'
2200   3BA5   4A4D50               .BYTE 'JMP'
2210   3BA8   4A5352               .BYTE 'JSR'
2220   3BAB   4C4441               .BYTE 'LDA'
2230   3BAE   4C4458               .BYTE 'LDX'
2240   3BB1   4C4459               .BYTE 'LDY'
2250   3BB4   4C5352               .BYTE 'LSR'
2260   3BB7   4E4F50               .BYTE 'NOP'
2270   3BBA   4F5241               .BYTE 'ORA'
2280   3BBD   504841               .BYTE 'PHA'
2290   3BC0   504850               .BYTE 'PHP'
2300   3BC3   504C41               .BYTE 'PLA'
2310   3BC6   504C50               .BYTE 'PLP'
2320   3BC9   524F4C               .BYTE 'ROL'
2330   3BCC   524F52               .BYTE 'ROR'
2340   3BCF   525449               .BYTE 'RTI'
2350   3BD2   525453               .BYTE 'RTS'
2360   3BD5   534243               .BYTE 'SBC'
2370   3BD8   534543               .BYTE 'SEC'
2380   3BDB   534544               .BYTE 'SED'
2390   3BDE   534549               .BYTE 'SEI'
2400   3BE1   535441               .BYTE 'STA'
2410   3BE4   535458               .BYTE 'STX'
2420   3BE7   535459               .BYTE 'STY'
2430   3BEA   544158               .BYTE 'TAX'
2440   3BED   544159               .BYTE 'TAY'
2450   3BF0   545358               .BYTE 'TSX'
2460   3BF3   545841               .BYTE 'TXA'
2470   3BF6   545853               .BYTE 'TXS'
2480   3BF9   545941               .BYTE 'TYA'
2490   3BFC   544558               .BYTE 'TEX'
2500                        ;
2510   3BFF   FF                   .BYTE ETX       SINCE THIS IS THE END OF A
2520                        ;                      STRING OF CHARACTERS, USE
2530                        ;                      ETX TO INDICATE END OF TEXT.
2540                        ;
2550                        ;
2560                        ;
2570                        ;
2580                        ;
2590                        ;
2600                        ;
2610                        ;
```

```
                              ;
                              ;
                              ;
                              ; ******************************************
                              ;
                              ;         TABLE OF MNEMONIC CODES
                              ;
                              ; ******************************************
                              ;
                              ;
                              ;
                              ;
                              ;
                              ;    A MNEMONIC'S CODE IS ITS OFFSET INTO
                              ;  MNAMES, THE LIST OF MNEONIC NAMES.
                              ;
                              ;
                              ;
3C00    226A010101   MCODES .BYTE $22,$6A,1,1,1,$6A,$0A,1,$70
        6A0A0170
3C09    6A0A01016A          .BYTE $6A,$0A,1,1,$6A,$0A,1
        0A01
3C10    1F6A010101          .BYTE $1F,$6A,1,1,1,$6A,$0A,1
        6A0A01
3C18    2B6A010101          .BYTE $2B,$6A,1,1,1,$6A,$0A,1
        6A0A01
3C20    5807010116          .BYTE $58,7,1,1,$16,7,$79,1
        077901
3C28    7607790116          .BYTE $76,7,$79,1,$16,7,$79,1
        077901
3C30    1907010101          .BYTE $19,7,1,1,1,7,$79,1
        077901
3C38    8807010101          .BYTE $88,7,1,1,1,7,$79,1
        077901
3C40    7F49010101          .BYTE $7F,$49,1,1,1,$49,$64,1
        496401
3C48    6D49640155          .BYTE $6D,$49,$64,1,$55,$49,$64,1
        496401
3C50    2549010101          .BYTE $25,$49,1,1,1,$49,$64,1
        496401
3C58    3149010101          .BYTE $31,$49,1,1,1,$49,$64,1
        496401
3C60    8204010101          .BYTE $82,4,1,1,1,4,$7C,1
        047C01
3C68    73047C0155          .BYTE $73,4,$7C,1,$55,4,$7C,1
        047C01
3C70    2804010101          .BYTE $28,4,1,1,1,4,$7C,1
        047C01
3C78    8E04010101          .BYTE $8E,4,1,1,1,4,$7C,$AC
        047CAC
3C80    0191010197          .BYTE 1,$91,1,1,$97,$91,$94,1
        919401
3C88    4601A30197          .BYTE $46,1,$A3,1,$97,$91,$94,1
        919401
```

```
3C90  0D91010197        .BYTE $0D,$91,1,1,$97,$91,$94,1
      919401
3C98  A991A30101        .BYTE $A9,$91,$A3,1,1,$91,1,1
      910101
3CA0  615B5E0161        .BYTE $61,$5B,$5E,1,$61,$5B,$5E,1
      5B5E01
3CA8  9D5B9A0161        .BYTE $9D,$5B,$9A,1,$61,$5B,$5E,1
      5B5E01
3CB0  105B010161        .BYTE $10,$5B,1,1,$61,$5B,$5E,1
      5B5E01
3CB8  345B9E0161        .BYTE $34,$5B,$9E,1,$61,$5B,$5E,1
      5B5E01
3CC0  3D3701013D        .BYTE $3D,$37,1,1,$3D,$37,$40,1
      374001
3CC8  523743013D        .BYTE $52,$37,$43,1,$3D,$37,$40,1
      374001
3CD0  1C37010101        .BYTE $1C,$37,1,1,1,$37,$40,1
      374001
3CD8  2E37010101        .BYTE $2E,$37,1,1,1,$37,$40,1
      374001
3CE0  3A8501013A        .BYTE $3A,$85,1,1,$3A,$85,$4C,1
      854C01
3CE8  4F8567013A        .BYTE $4F,$85,$67,1,$3A,$85,$4C,1
      854C01
3CF0  1385010101        .BYTE $13,$85,1,1,1,$85,$4C,1
      854C01
3CF8  8B85010101        .BYTE $8B,$85,1,1,1,$85,$4C,1
      854C01
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ; ********************************************
                  ;
                  ;        TABLE OF ADDRESSING MODE CODES
                  ;
                  ; ********************************************
                  ;
                  ;
                  ;
                  ;
                  ;     AN ADDRESSING MODE'S CODE IS ITS OFFSET
                  ; INTO SUBS, THE TABLE OF ADDRESSING MODE
                  ; SUBROUTINES.
                  ;
                  ;
                  ;
```

```
3700                          ;
3710                          ;
3720                          ;
3730   3D00   1216000000   MODES   .BYTE 18,22,0,0,0,6,6,0
3740          060600
3750   3D08   1204020000           .BYTE 18,4,2,0,0,12,12,0
3760          0C0C00
3770   3D10   1418000000           .BYTE 20,24,0,0,0,14,14,0
3780          0E0E00
3790   3D18   1210000000           .BYTE 18,16,0,0,0,22,22,0
3800          161600
3810   3D20   0C16000006           .BYTE 12,22,0,0,6,6,6,0
3820          060600
3830   3D28   120402000C           .BYTE 18,4,2,0,12,12,12,0
3840          0C0C00
3850   3D30   1418000000           .BYTE 20,24,0,0,0,8,8,0
3860          080800
3870   3D38   1210000000           .BYTE 18,16,0,0,0,14,14,0
3880          0E0E00
3890   3D40   1216000000           .BYTE 18,22,0,0,0,6,6,0
3900          060600
3910   3D48   120C02000C           .BYTE 18,12,2,0,12,12,12,0
3920          0C0C00
3930   3D50   1418000000           .BYTE 20,24,0,0,0,8,8,0
3940          080800
3950   3D58   1210000000           .BYTE 18,16,0,0,0,14,14,0
3960          0E0E00
3970   3D60   1216000000           .BYTE 18,22,0,0,0,6,6,0
3980          060600
3990   3D68   120402001A           .BYTE 18,4,2,0,26,12,12,0
4000          0C0C00
4010   3D70   1418000000           .BYTE 20,24,0,0,0,8,8,0
4020          080800
4030   3D78   1210000000           .BYTE 18,16,0,0,0,14,14,28
4040          0E0E1C
4050                          ;
4060   3D80   0016000006           .BYTE 0,22,0,0,6,6,6,0
4070          060600
4080   3D88   120012000C           .BYTE 18,0,18,0,12,12,12,0
4090          0C0C00
4100   3D90   1418000008           .BYTE 20,24,0,0,8,8,10,0
4110          080A00
4120   3D98   1210120000           .BYTE 18,16,18,0,0,14,0,0
4130          0E0000
4140   3DA0   0416040006           .BYTE 4,22,4,0,6,6,6,0
4150          060600
4160   3DA8   120412000C           .BYTE 18,4,18,0,12,12,12,0
4170          0C0C00
4180   3DB0   1418000008           .BYTE 20,24,0,0,8,8,10,0
4190          080A00
4200   3DB8   141012000E           .BYTE 20,16,18,0,14,14,16,0
4210          0E1000
4220   3DC0   0416000006           .BYTE 4,22,0,0,6,6,6,0
4230          060600
```

```
4240   3DC8  120412000C       .BYTE 18,4,18,0,12,12,12,0
4250         0C0C00
4260   3DD0  1418000000       .BYTE 20,24,0,0,0,8,8,0
4270         080800
4280   3DD8  1210000000       .BYTE 18,16,0,0,0,14,14,0
4290         0E0E00
4300   3DE0  0416000006       .BYTE 4,22,0,0,6,6,6,0
4310         060600
4320   3DE8  120412000C       .BYTE 18,4,18,0,12,12,12,0
4330         0C0C00
4340   3DF0  1418000000       .BYTE 20,24,0,0,0,8,8,0
4350         080800
4360   3DF8  1210000000       .BYTE 18,16,0,0,0,14,14,0
4370         0E0E00
```

```
                 CROSS REFERENCE LISTING:


DSPAGE 3900      ETX    00FF      MCODES 3C00      MNAMES 3B50
MODES  3D00      TEX    007F
```

Appendix C9:

Move Utilities

```
;        APPENDIX C9: ASSEMBLER LISTING OF
;                     MOVE UTILITIES
;
;
;
; SEE CHAPTER 10 OF TOP-DOWN ASSEMBLY LANGUAGE
; PROGRAMMING FOR YOUR COMMODORE 64 AND VIC-20
;
;                  BY KEN SKIER
;
;
;         COPYRIGHT (C) 1984 BY KENNETH SKIER
;              LEXINGTON, MASSACHUSETTS
;
;
;
;
;
;
;
; *******************************************
;
;                 CONSTANTS
;
; *******************************************
;
;
;
;
;
       CR      = $0D  CARRIAGE RETURN.
       LF      = $0A  LINE FEED.
       TEX     = $7F  START OF TEXT CHARACTER.
       ETX     = $FF  END OF TEXT CHARACTER.
;
;
;
;
;
;
; *******************************************
;
;            EXTERNAL ADDRESSES
;
; *******************************************
;
;
;
;
;
;
;
;
       VMPAGE   = $3200
```

```
                                        STARTING PAGE OF VISIBLE
                     ;                  MONITOR CODE.
                     ;
                        SELECT   = VMPAGE+5
                        VISMON   = VMPAGE+7
                     ;
                     ;
                     ;
                        PRPAGE   = $3400
                                        STARTING PAGE OF PRINT CODE.
                     ;
                        TVT.ON   = PRPAGE+8
                        PRINT:   = PRPAGE+$E4
                        PUSHSL   = PRPAGE+$112
                        POP.SL   = PRPAGE+$12B
                     ;
                     ;
                        HEX.PG   = $3500
                                        ADDRESS OF PAGE IN WHICH
                     ;                  HEXDUMP CODE STARTS.
                     ;                  (HEXDUMP CODE STARTS AT
                     ;                  $3550, BUT IT'S EASIER TO
                     ;                  COUNT FROM $3500.)
                     ;
                        SETADS   = HEX.PG+$E3
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ; ******************************************
                     ;
                     ;                VARIABLES
                     ;
                     ; ******************************************
                     ;
                     ;
                     ;
                     ;
                     ;
0000    = 37B0          * = $37B0
                     ;
                     ;
                        SA       = HEX.PG+$52
                                        POINTER TO START ADDRESS
                     ;                  OF BLOCK TO BE MOVED.
                     ;
                        EA       = SA+2
                                        POINTER TO END OF BLOCK TO
                     ;                  BE MOVED.
                     ;
```

```
37B0   0000           NUM     .WORD 0         NUMBER OF BYTES IN BLOCK
                      ;                       TO BE MOVED.  ZERO MEANS
                      ;                       BLOCK CONTAINS 1 BYTE.
                      ;
                      ;
37B2   0000           DEST    .WORD 0         POINTER TO BLOCK'S
                      ;                       DESTINATION.
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                                             THESE TWO "PAGE POINTERS"
                                             GET AND PUT BYTES:
                              GETPTR   = $FB
                              PUTPTR   = GETPTR+2
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ; ****************************************
                      ;
                      ;                 MOVE TOOL
                      ;
                      ; ****************************************
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
37B4   200834         MOVER   JSR TVT.ON      SELECT SCREEN FOR OUTPUT.
37B7   20E434                 JSR PRINT:      DISPLAY A TITLE.
37BA   7F0D0A                 .BYTE TEX,CR,LF
37BD   2020202020             .BYTE '     MOVE TOOL.'    ;
       4D4F564520
       544F4F4C2E
37CC   0D0A0AFF               .BYTE CR,LF,LF,ETX
                      ;
37D0   20E335                 JSR SETADS      GET START ADDRESS, END
                      ;                       ADDRESS FROM USER.
                      ;
37D3   20B938                 JSR SET.DA      GET DESTINATION ADDRESS
                      ;                       FROM USER.
                      ;                       WITH THOSE POINTERS SET,
                      ;                       WE'RE READY TO EXECUTE MOV.EA:
                      ;
```

```
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ; ********************************************
                     ;
                     ;  MOV.EA:    MOVE BLOCK SPECIFIED BY SA, EA, DEST
                     ;
                     ; ********************************************
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;  RETURN CODES:
                     ;
                     ;
                            ERROR    = 0    THIS RETURN CODE MEANS
                     ;                      SA < EA, SO MOVE ABORTED.
                            OKAY     = $FF  THIS RETURN CODE MEANS
                     ;                      MOVE ACCOMPLISHED.
                     ;
                     ;
37D6  AE5535         MOV.EA LDX EA+1        SET NUM EQUAL TO EA - SA:
37D9  38                    SEC
37DA  AD5435                LDA EA
37DD  ED5235                SBC SA
37E0  8DB037                STA NUM
37E3  B002                  BCS MOVE.1
37E5  CA                    DEX
37E6  38                    SEC
37E7  8A             MOVE.1 TXA
37E8  ED5335                SBC SA+1
37EB  8DB137                STA NUM+1
37EE  B003                  BCS MOVNUM
                     ;
37F0  A900           ER.RTN LDA #ERROR      IF EA < SA,
37F2  60                    RTS             RETURN WITH ERROR CODE.
                     ;
                     ;
                     ;
                     ;
                     ; ******************************************
                     ;
                     ; MOVNUM: MOVE BLOCK SPECIFIED BY SA, NUM, DEST.
                     ;
                     ; ******************************************
                     ;
                     ;
                     ;
37F3  A003           MOVNUM LDY #3          SAVE ZERO PAGE BYTES THAT
37F5  B9FB00         LOOP.1 LDA GETPTR,Y    WILL BE CHANGED.
```

```
37F8   48                   PHA
37F9   88                   DEY
37FA   10F9                 BPL LOOP.1
                 ;
                 ;
37FC   38                   SEC                IF DEST>SA, BRANCH TO MOVE-UP
37FD   AD5335               LDA SA+1
3800   CDB337               CMP DEST+1
3803   9040                 BCC MOVEUP
3805   D018                 BNE MOVEDN
                 ;                             IF DEST<SA, BRANCH TO
                 ;                             MOVE-DOWN.
3807   AD5235               LDA SA
380A   CDB237               CMP DEST
380D   9036                 BCC MOVEUP
380F   D00E                 BNE MOVEDN         IF DEST EQUALS SA,
3811   A000          OK.RTN LDY #0             RETURN BEARING "OKAY" CODE.
                 ;                             RESTORE ZERO PAGE BYTES
3813   68            LOOP.2 PLA                THAT WERE CHANGED.
3814   99FB00               STA GETPTR,Y
3817   C8                   INY
3818   C004                 CPY #4
381A   D0F7                 BNE LOOP.2
381C   A9FF                 LDA #OKAY          RETURN W/"OKAY" CODE.
381E   60                   RTS
                 ;
                 ;
                 ;
381F   20A438        MOVEDN JSR LOPAGE         SET PAGE POINTERS TO LOWEST
                 ;                             PAGES IN ORIGIN, DESTINATION
                 ;                             BLOCKS.
                 ;
                 ;
3822   A000                 LDY #0             INITIALIZE PAGE INDEX TO
                 ;                             BOTTOM OF PAGE.
                 ;
3824   AEB137               LDX NUM+1          USE X TO COUNT THE NUMBER
                 ;                             OF PAGES TO MOVE.  MORE THAN
                 ;                             ONE PAGE TO MOVE?
3827   F00E                 BEQ LESSDN         IF NOT, MOVE LESS THAN A
                 ;                             PAGE.
                 ;
                 ;                             IF SO,
3829   B1FB          PAGEDN LDA (GETPTR),Y     MOVE A PAGE DOWN,
382B   91FD                 STA (PUTPTR),Y     STARTING AT THE BOTTOM.
382D   C8                   INY                INCREMENT PAGE INDEX.
382E   D0F9                 BNE PAGEDN         IF PAGE NOT MOVED, MOVE
                 ;                             NEXT BYTE...
                 ;
3830   E6FC                 INC GETPTR+1       INCREMENT PAGE POINTERS.
3832   E6FE                 INC PUTPTR+1
3834   CA                   DEX                DECREMENT PAGE COUNT.
3835   D0F2                 BNE PAGEDN         IF A PAGE LEFT TO MOVE,
                 ;                             MOVE IT AS A PAGE.
```

```
3700                      ;
3710  3837  88            LESSDN DEY
3720  3838  C8                   INY              MOVE LESS THAN A PAGE
3730  3839  B1FB                 LDA (GETPTR),Y   DOWN, STARTING AT THE
3740  383B  91FD                 STA (PUTPTR),Y   BOTTOM.
3750  383D  CCB037               CPY NUM          MOVED LAST BYTE?
3760  3840  D0F6                 BNE LESSDN+1     IF NOT, MOVE NEXT BYTE...
3770  3842  4C1138               JMP OK.RTN       IF SO, RETURN BEARING
3780                      ;                       "OKAY" CODE.
3790                      ;
3800                      ;
3810                      ;
3820  3845  ADB137        MOVEUP LDA NUM+1        MORE THAN A PAGE TO MOVE?
3830  3848  F048                 BEQ LESSUP       IF NOT, MOVE LESS THAN A
3840                      ;                       PAGE.
3850                      ;
3860                      ;
3870                      ;
3880                      ;                       TO MOVE MORE THAN A PAGE,
3890                      ;                       SET PAGE POINTERS TO
3900                      ;                       HIGHEST PAGES IN ORIGIN,
3910                      ;                       DESTINATION BLOCKS.
3920                      ;
3930                      ;
3940                      ;                       TO DO THIS, FIRST SET
3950                      ;                       (X,Y) EQUAL TO NUM - $FF,
3960                      ;                       (RELATIVE ADDRESS OF
3970                      ;                       HIGHEST PAGE IN A BLOCK.)
3980                      ;
3990                      ;
4000  384A  ACB137               LDY NUM+1
4010  384D  ADB037               LDA NUM
4020  3850  38                   SEC
4030  3851  E9FF                 SBC #$FF
4040  3853  B001                 BCS NEXT.1
4050  3855  88                   DEY
4060
4070  3856  AA            NEXT.1 TAX
4080                      ;
4090                      ;                       NOW (X,Y) - NUM - $FF.
4100                      ;                       X IS LOW BYTE, Y IS HIGH BYTE
4110                      ;
4120                      ;
4130  3857  84FE                 STY PUTPTR+1
4140  3859  8A                   TXA
4150  385A  18                   CLC
4160  385B  6D5235               ADC SA
4170  385E  85FB                 STA GETPTR
4180  3860  9001                 BCC NEXT.2
4190  3862  C8                   INY
4200                      ;
4210                      ;
4220  3863  98            NEXT.2 TYA
4230  3864  6D5335               ADC SA+1
```

```
3867   85FC              STA GETPTR+1
                  ;
                  ;                        NOW GETPTR IS SA+NUM-$FF.
                  ;                        (LAST PAGE IN SOURCE BLOCK.)
                  ;
                  ;
3869   8A                TXA
386A   18                CLC
386B   6DB237            ADC DEST
386E   85FD              STA PUTPTR
3870   9002              BCC NEXT.3
3872   E6FE              INC PUTPTR+1
                  ;
                  ;
3874   A5FE       NEXT.3 LDA PUTPTR+1
3876   6DB337            ADC DEST+1
3879   85FE              STA PUTPTR+1
                  ;                        NOW PUTPTR IS DEST+NUM-$FF.
                  ;                        (LAST PAGE IN DEST BLOCK.)
                  ;
                  ;
                  ;
387B   AEB137            LDX NUM+1         LOAD X WITH NUMBER OF
                  ;                        PAGES TO MOVE.
                  ;
387E   A0FF       PAGEUP LDY #$FF          SET PAGE INDEX TO TOP OF
                  ;                        PAGE.

3880   B1FB       LOOP.3 LDA (GETPTR),Y    MOVE A PAGE UP, STARTING
3882   91FD              STA (PUTPTR),Y    AT THE TOP OF THE BLOCK.
3884   88                DEY               DECREMENT PAGE INDEX.
                  ;                        ABOUT TO MOVE LAST BYTE
                  ;                        IN PAGE?
3885   D0F9              BNE LOOP.3        IF NOT, HANDLE NEXT BYTE.
                  ;                        AS BEFORE.
                  ;
                  ;
                  ;
3887   B1FB              LDA (GETPTR),Y    IF SO, MOVE THIS BYTE FROM
3889   91FD              STA (PUTPTR),Y    SOURCE TO DESTINATION.
388B   C6FC              DEC GETPTR+1
388D   C6FE              DEC PUTPTR+1      DECREMENT PAGE POINTERS.
388F   CA                DEX               DECREMENT PAGE COUNTER.
3890   D0EC              BNE PAGEUP        IF A PAGE LEFT TO MOVE,
                  ;                        MOVE IT AS A PAGE....
                  ;
                  ;
3892   20A438     LESSUP JSR LOPAGE        MOVE LESS THAN A PAGE UP,
3895   ACB037            LDY NUM           STARTING AT THE TOP.
                  ;
3898   B1FB       MOVE.6 LDA (GETPTR),Y    COPY A BYTE FROM ORIGIN
389A   91FD              STA (PUTPTR),Y    TO DESTINATION.
389C   88                DEY               DECREMENT PAGE INDEX.
389D   C0FF              CPY #$FF          COPIED THE LAST BYTE?
```

```
389F   D0F7              BNE MOVE.6      IF NOT, HANDLE AS BEFORE...
38A1   4C1138            JMP OK.RTN      IF SO, RETURN BEARING
                   ;                     "OKAY" CODE.
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ; ******************************************
                   ;
                   ;      SET PAGE POINTERS TO BOTTOM OF
                   ;        ORIGIN, DESTINATION BLOCKS.
                   ;
                   ; ******************************************
                   ;
                   ;
                   ;
                   ;
                   ;
38A4   AD5235      LOPAGE LDA SA
38A7   85FB              STA GETPTR
38A9   AD5335            LDA SA+1
38AC   85FC              STA GETPTR+1
                   ;
                   ;
38AE   ADB237            LDA DEST
38B1   85FD              STA PUTPTR
38B3   ADB337            LDA DEST+1
38B6   85FE              STA PUTPTR+1
                   ;
                   ;
38B8   60                RTS
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ; ******************************************
                   ;
                   ;     LET USER SET DESTINATION ADDRESS
                   ;
                   ; ******************************************
                   ;
                   ;
                   ;
```

```
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
38B9   200834      SET.DA JSR TVT.ON      LET USER SET DESTINATION
38BC   20E434             JSR PRINT:
38BF   7F0D0A             .BYTE TEX,CR,LF                          ;
38C2   5345542044         .BYTE 'SET DESTINATION AND PRESS Q.'     ;
       455354494E
       4154494F4E
       20414E4420
       5052455353
       20512E
38DE   FF                 .BYTE ETX
38DF   200732             JSR VISMON      LET USER SET AN ADDRESS.

38E2   AD0532      DAHERE LDA SELECT      SET DEST EQUAL TO SELECT.
38E5   8DB237             STA DEST
38E8   AD0632             LDA SELECT+1
38EB   8DB337             STA DEST+1
                          ;
38EE   60                 RTS             RETURN.
```

CROSS REFERENCE LISTING:

CR	000D	DAHERE	38E2	DEST	37B2	EA	3554
ER.RTN	37F0	ERROR	0000	ETX	00FF	GETPTR	00FB
HEX.PG	3500	LESSDN	3837	LESSUP	3892	LF	000A
LOOP.1	37F5	LOOP.2	3813	LOOP.3	3880	LOPAGE	38A4
MOV.EA	37D6	MOVE.1	37E7	MOVE.6	3898	MOVEDN	381F
MOVER	37B4	MOVEUP	3845	MOVNUM	37F3	NEXT.1	3856
NEXT.2	3863	NEXT.3	3874	NUM	37B0	OK.RTN	3811
OKAY	00FF	PAGEDN	3829	PAGEUP	387E	POP.SL	352B
PRINT	34E4	PRPAGE	3400	PUSHSL	3512	PUTPTR	00FD
SA	3552	SELECT	3205	SET.DA	38B9	SETADS	35E3
TEX	007F	TVT.ON	3408	VISMON	3207	VMPAGE	3200

Appendix C10:

Simple Text Editor (Top Level and Display Subroutines)

```
;         APPENDIX C10: ASSEMBLER LISTING OF
;                       A SIMPLE TEXT EDITOR (TOP
;                       LEVEL AND DISPLAY SUBROUTINES)
;
;
;
;
;
;      SEE CHAPTER 11 OF TOP-DOWN ASSEMBLY LANGUAGE
;      PROGRAMMING FOR YOUR COMMODORE 64 AND VIC-20
;
;
;                    BY KEN SKIER
;
;
;            COPYRIGHT (C) 1984 BY KENNETH SKIER
;                  LEXINGTON, MASSACHUSETTS
;
;
;
;
;
;
;
;
;
;
;
;
; ********************************************
;
;                     CONSTANTS
;
; ********************************************
;
;
;
;
;
      CR     = $0D   CARRIAGE RETURN.
;
      LF     = $0A   LINE FEED.
;
      TEX    = $7F   THIS CHARACTER MUST START
;                    ANY MESSAGE.
;
      ETX    = $FF   THIS CHARACTER MUST END
;                    ANY MESSAGE.
;
      INSCHR = 'I'   GRAPHIC FOR INSERT MODE
      OVRCHR = 'O'   GRAPHIC FOR OVERSTRIKE MODE.
;
;
;
```

```
;
;
;
;
;
; ******************************************
;
;             EXTERNAL ADDRESSES
;
; ******************************************
;
;
;
        TV.PTR = $FB       POINTER TO A SCREEN ADDRESS.
        PARAMS = $3000     SYSTEM DATA BLOCK.
;
;
        TVCOLS = PARAMS+3
        TVROWS = PARAMS+4
        ARROW  = PARAMS+7
;
;
;
        TVSUBS = $3100
        CLR.XY = TVSUBS+$13
        TVHOME = TVSUBS+$2B
        TVTOXY = TVSUBS+$3C
        TVDOWN = TVSUBS+$76
        TVSKIP = TVSUBS+$7F
        TVPLUS = TVSUBS+$81
        TV.PUT = TVSUBS+$9B
        VUBYTE = TVSUBS+$A3
        TVPUSH = TVSUBS+$C4
        TV.POP = TVSUBS+$D3
;
;
        VMPAGE = $3200
                      STARTING PAGE OF VISIBLE
;                     MONITOR CODE.

        SELECT = VMPAGE+5
        GET.SL = VMPAGE+$95
        INC.SL = VMPAGE+$10D
        DEC.SL = VMPAGE+$11A
;
;
        PRPAGE = $3400
                      STARTING PAGE OF PRINT
;                     UTILITIES.

        TVT.ON = PRPAGE+8
        TVTOFF = PRPAGE+$0E
        PR.ON  = PRPAGE+$14
        PR.OFF = PRPAGE+$1A
```

```
                                    PR.CHR = PRPAGE+$40
                                    PRINT: = PRPAGE+$E4
                                    PUSHSL = PRPAGE+$112
                                    POP.SL = PRPAGE+$12B
                             ;
                             ;
                                    HEX.PG = $3500
                                                    ADDRESS OF PAGE IN WHICH
                             ;                      HEXDUMP CODE STARTS.
                             ;
                                    SA     = HEX.PG+$52
                                    EA     = SA+2
                                    SETADS = HEX.PG+$E3
                                    NEXTSL = HEX.PG+$27D
                                    GOTOSA = HEX.PG+$29A
                             ;
                             ;
                                    EDPAGE = $3E00
                                                    STARTING PAGE OF EDITOR.

                                    EDITIT = EDPAGE+$C8
                             ;
                             ;
                             ;
                             ;
                             ;
                             ;
                             ; ******************************************
                             ;
                             ;                 VARIABLES
                             ;
                             ; ******************************************
                             ;
                             ;
                             ;
      0000   = 3E00                 * =  EDPAGE
                             ;
                             ;
                             ;
      3E00   00              COUNTR .BYTE 0         COUNTER USED BY LINE.2.
      3E01   00              EDMODE .BYTE 0         FLAG: 0 FOR OVERSTRIKE,
                             ;                            1 FOR INSERT.
                             ;
                             ;
                             ;
                             ; *********************************************
                             ;
                             ;            TEXT EDITOR: TOP LEVEL
                             ;
                             ; *********************************************
                             ;
                             ;
                             ;
                             ;
```

```
                                ;
                                ;
      3E02  200F3E     EDITOR JSR SETBUF      INITIALIZE BUFFER POINTERS.
      3E05  20373E     EDLOOP JSR SHOWIT      SHOW USER A PORTION OF
                                ;             EDIT BUFFER.
      3E08  20C83E            JSR EDITIT      LET THE USER EDIT THE BUFFER
                                ;             OR MOVE ABOUT WITHIN IT.
      3E0B  18                CLC
      3E0C  18                CLC             LOOP BACK TO SHOW THE
      3E0D  90F6              BCC EDLOOP      CURRENT TEXT.
                                ;
                                ;
                                ;
                                ;
                                ;
                                ;
                                ;
                                ;
                                ;
                                ;
                                ; ********************************************
                                ;
                                ;          INITIALIZE BUFFER POINTERS
                                ;
                                ; ********************************************
                                ;
                                ;
                                ;
                                ;
      3E0F  200834     SETBUF JSR TVT.ON      SELECT SCREEN.
      3E12  20E434            JSR PRINT:      DISPLAY "SET UP EDIT BUFFER."
      3E15  7F0D0A0A          .BYTE TEX,CR,LF,LF
      3E19  5345542055        .BYTE 'SET UP EDIT BUFFER.' ;
            5020454449
            5420425546
            4645522E
      3E2C  0D0A0AFF          .BYTE CR,LF,LF,ETX
      3E30  20E335            JSR SETADS      LET USER SET LOCATION AND
                                ;             SIZE OF EDIT BUFFER.
      3E33  209A37            JSR GOTOSA      MAKE SELECT PT TO START OF
                                              BUFFER...
      3E36  60                RTS             AND RETURN TO CALLER.
                                ;
                                ;
                                ;
                                ;
                                ;
                                ;
                                ;
                                ; ********************************************
                                ;
                                ;     DISPLAY A PORTION OF EDIT BUFFER
                                ;
                                ; ********************************************
```

```
                ;
                ;
                ;
                ;
                ;
3E37  20C431    SHOWIT JSR TVPUSH    SAVE THE ZERO PAGE BYTES
                ;                    WE'LL USE.
3E3A  202B31           JSR TVHOME    SET HOME POSITION OF EDIT
                ;                    DISPLAY.
                ;
                ;
3E3D  AE0330           LDX TVCOLS    CLEAR THREE ROWS FOR
3E40  A003             LDY #3        THE EDIT DISPLAY.
3E42  201331           JSR CLR.XY
                ;
                ;
3E45  202B31           JSR TVHOME    RESTORE TV.PTR TO HOME
                ;                    POSITION OF EDIT DISPLAY.
3E48  207631           JSR TVDOWN    SET TV.PTR TO BEGINNING
3E4B  20C431           JSR TVPUSH    OF LINE TWO AND SAVE IT.
3E4E  205E3E           JSR LINE.2    DISPLAY TEXT IN LINE TWO.
                ;
                ;
3E51  20D331           JSR TV.POP    SET TV.PTR TO BEGINNING OF
3E54  207631           JSR TVDOWN    OF THIRD LINE OF EDIT
                ;                    DISPLAY.
3E57  20883E           JSR LINE.3    DISPLAY THIRD LINE OF EDIT
                ;                    DISPLAY.
                ;
3E5A  20D331           JSR TV.POP    RESTORE ZERO PAGE BYTES USED.
3E5D  60               RTS           RETURN TO CALLER, WITH EDIT
                ;                    DISPLAY ON SCREEN, REST OF
                ;                    SCREEN UNCHANGED, AND ZERO
                ;                    PAGE PRESERVED.
                ;
                ;
                ;
                ;
                ;
                ;
                ; *******************************************
                ;
                ;                DISPLAY TEXT LINE
                ;
                ; *******************************************
                ;
                ;
                ;
                ;
                ;
3E5E  201235    LINE.2 JSR PUSHSL    SAVE SELECT POINTER.
3E61  AD0330           LDA TVCOLS    SET X EQUAL TO
3E64  4A               LSR A         HALF THE WIDTH
3E65  AA               TAX           OF THE SCREEN.
```

```
3E66  CA                     DEX
                      ;
3E67  201A33      LOOP.1 JSR DEC.SL      DECREMENT SELECT...
3E6A  CA                 DEX
3E6B  10FA               BPL LOOP.1      ...X TIMES.
                      ;
3E6D  AD0330             LDA TVCOLS      INITIALIZE COUNTR.
3E70  8D003E             STA COUNTR      (WE'LL DISPLAY TVCOLS
                      ;                  CHARACTERS.)
3E73  209532      LOOP.2 JSR GET.SL      GET A CHARACTER FROM BUFFER.
3E76  209B31             JSR TV.PUT      PUT IT ON SCREEN.
3E79  207F31             JSR TVSKIP      GO TO NEXT SCREEN POSITION.
3E7C  200D33             JSR INC.SL      ADVANCE TO NEXT BYTE IN
                      ;                  BUFFER.
3E7F  CE003E             DEC COUNTR      DONE LAST CHARACTER IN ROW?
3E82  10EF               BPL LOOP.2      IF NOT, DO NEXT CHARACTER.
                      ;
                      ;
3E84  202B35             JSR POP.SL      RESTORE SELECT FROM STACK.
3E87  60                 RTS             RETURN TO CALLER.
                      ;
                      ;
                      ;
                      ;
                      ;
                      ; ********************************************
                      ;
                      ;          DISPLAY STATUS LINE
                      ;
                      ; ********************************************
                      ;
                      ;
                      ;
                      ;
                      ;
3E88  AD0330      LINE.3 LDA TVCOLS      SELECT CENTER POSITION...
3E8B  4A                 LSR A           NOW A IS TVCOLS/2
3E8C  E902               SBC #2          NOW A (TVCOLS/2)-2
3E8E  208131             JSR TVPLUS      NOW TV.PTR IS POINTING TWO
                      ;                  CHARACTERS TO THE LEFT OF
                      ;                  CENTER OF LINE 3 OF THE
                      ;                  EDIT DISPLAY.
3E91  AD013E             LDA EDMODE      WHAT IS CURRENT MODE?
3E94  C901               CMP #1          IS IT INSERT MODE?
3E96  D005               BNE OVMODE      IF NOT, IT MUST BE OVERSTRIKE
                      ;                  MODE.
3E98  A949               LDA #INSCHR     IF SO, GET INSERT GRAPHIC.
3E9A  18                 CLC
3E9B  9002               BCC TVMODE
3E9D  A94F        OVMODE LDA #OVRCHR     LOAD A W/OVERSTRIKE CHARACTER.
3E9F  209B31      TVMODE JSR TV.PUT      PUT MODE GRAPHIC ON SCREEN.
3EA2  A902               LDA #2          MOVE TWO POSITIONS TO THE
3EA4  208131             JSR TVPLUS      RIGHT, SO TV.PTR POINTS TO
                      ;                  CENTER OF LINE 3 OF EDIT
```

```
                                         DISPLAY.
4240                   ;
4250  3EA7  AD0730          LDA ARROW     DISPLAY AN UP-ARROW HERE.
4260  3EAA  209B31          JSR TV.PUT
4270                   ;
4280  3EAD  A902            LDA #2        GO TWO POSITIONS TO THE
4290  3EAF  208131          JSR TVPLUS    RIGHT, SO TV.PTR POINTS TO
4300                   ;                  FIELD RESERVED FOR THE
4310                   ;                  ADDRESS OF THE CURRENT CHARACTER
4320  3EB2  AD0632          LDA SELECT+1  DISPLAY ADDRESS OF CURRENT
4330  3EB5  20A331          JSR VUBYTE
4340  3EB8  AD0532          LDA SELECT
4350  3EBB  20A331          JSR VUBYTE
4360                   ;
4370  3EBE  60              RTS           RETURN TO CALLER.
```

CROSS REFERENCE LISTING:

ARROW	3007	CLR.XY	3113	COUNTR	3E00	CR	000D
DEC.SL	331A	EA	3554	EDITIT	3EC8	EDITOR	3E02
EDLOOP	3E05	EDMODE	3E01	EDPAGE	3E00	ETX	00FF
GET.SL	3295	GOTOSA	379A	HEX.PG	3500	INC.SL	330D
INSCHR	0049	LF	000A	LINE.2	3E5E	LINE.3	3E88
LOOP.1	3E67	LOOP.2	3E73	NEXTSL	377D	OVMODE	3E9D
OVRCHR	004F	PARAMS	3000	POP.SL	352B	PR.CHR	3440
PR.OFF	341A	PR.ON	3414	PRINT	34E4	PRPAGE	3400
PUSHSL	3512	SA	3552	SELECT	3205	SETADS	35E3
SETBUF	3E0F	SHOWIT	3E37	TEX	007F	TV.POP	31D3
TV.PTR	00FB	TV.PUT	319B	TVCOLS	3003	TVDOWN	3176
TVHOME	312B	TVMODE	3E9F	TVPLUS	3181	TVPUSH	31C4
TVROWS	3004	TVSKIP	317F	TVSUBS	3100	TVT.ON	3408
TVTOFF	340E	TVTOXY	313C	VMPAGE	3200	VUBYTE	31A3

Appendix C11:

Simple Text Editor (EDITIT Subroutine)

```
;        APPENDIX C11: ASSEMBLER LISTING OF
;                      A SIMPLE TEXT EDITOR
;                      EDITIT SUBROUTINE
;
;
;
;
;
; SEE CHAPTER 11 OF TOP-DOWN ASSEMBLY LANGUAGE
; PROGRAMMING FOR YOUR COMMODORE 64 AND VIC-20
;
;
;                      BY KEN SKIER
;
;
;         COPYRIGHT (C) 1984 BY KENNETH SKIER
;              LEXINGTON, MASSACHUSETTS
;
;
;
;
;
;
;
;
;
;
;
;
; ********************************************
;
;                  CONSTANTS
;
; ********************************************
;
;
;
;
;
    CR      = $0D   CARRIAGE RETURN.
;
    LF      = $0A   LINE FEED.
;
;
    TEX     = $7F   THIS CHARACTER MUST START
;                   ANY MESSAGE.
;
    ETX     = $FF   THIS CHARACTER MUST END
;                   ANY MESSAGE.
;
;
;
;
;
```

```
;
;
;
;
; ******************************************
;
;             EXTERNAL  ADDRESSES
;
; ******************************************
;
;
;
;
;
    VMPAGE = $3200
;                    STARTING PAGE OF VISIBLE
;                    MONITOR CODE.
    SELECT = VMPAGE+5
    VISMON = VMPAGE+7
    GET.SL = VMPAGE+$95
    GETKEY = VMPAGE+$E0
    INC.SL = VMPAGE+$10D
    DEC.SL = VMPAGE+$11A
    PUT.SL = VMPAGE+$12D
;
;
    PRPAGE = $3400
;                    STARTING PAGE OF PRINT
;                    UTILITIES.
;
    PR.ON  = PRPAGE+$14
    PR.OFF = PRPAGE+$1A
    PR.CHR = PRPAGE+$40
    PUSHSL = PRPAGE+$112
    POP.SL = PRPAGE+$12B
;
;
    HEX.PG = $3500
;                    ADDRESS OF PAGE IN WHICH
;                    HEXDUMP CODE STARTS.
;
    SA     = HEX.PG+$52
    EA     = SA+2
    SAHERE = HEX.PG+$161
    NEXTSL = HEX.PG+$27D
    GOTOSA = HEX.PG+$29A
;
;
    MOVERS = $37B0
;                    START OF MOVE OBJECT CODE.
;
    DEST   = MOVERS+2
    MOV.EA = MOVERS+$26
    DAHERE = MOVERS+$132
```

```
2080                          ;
2090                                EDPAGE = $3E00
2100                          ;                   STARTING PAGE OF EDITOR.
2110
2120                                EDKEYS = EDPAGE+$C0
2130                          ;
2140                          ;
2150                          ;
2160                          ;
2170                          ;
2180                          ;
2190                          ; *********************************************
2200                          ;
2210                          ;          VARIABLES
2220                          ;
2230                          ; *********************************************
2240                          ;
2250                          ;
2260                          ;
2270                                EDMODE = EDPAGE+1      0 FOR OVERSTRIKE MODE.
2280                          ;                            1 FOR INSERT.
2290                          ;
2300    0000   = 3EC0               * = EDKEYS
2310                          ;
2320                          ;        EDIT FUNCTION KEYS
2330                          ;
2340                          ;                    THE EDITOR RECOGNIZES THE
2350                          ;                    FOLLOWING KEYS AS FUNCTION KEYS.
2360                          ;                    ASSIGN A FUNCTION TO A KEY
2370                          ;                    BY STORING THE DESIRED KEY
2380                          ;                    CODE FROM YOUR SYSTEM'S
2390                          ;                    KEYHANDLER INTO ONE OF THE
2400                          ;                    FOLLOWING DATA BYTES:
2410                          ;
2420                          ;
2430    3EC0   93             FLSHKY .BYTE $93     THIS KEY FLUSHES THE
2440                          ;                    BUFFER OF ANY TEXT.  $93 IS
2450                          ;                    THE "CLR" KEY.  THUS, "CLR"
2460                          ;                    TO FLUSH THE BUFFER.
2470                          ;
2480                          ;
2490    3EC1   94             MODEKY .BYTE $94     THIS KEY CAUSES THE EDIT
2500                          ;                    TO CHANGE MODES, FROM INSERT
2510                          ;                    TO OVERSTRIKE, AND VICE VERSA.
2520                          ;                    $94 IS "INS" KEY.  THUS, PRESS
2530                          ;                    "INS" TO CHANGE MODES.
2540                          ;
2550    3EC2   1D             NEXTKY .BYTE $1D     THIS KEY SELECTS THE NEXT
2560                          ;                    CHARACTER IN THE BUFFER.
2570                          ;                    $1D IS THE RIGHT-ARROW.
2580                          ;                    THUS, PRESS RIGHT-ARROW TO
2590                          ;                    MOVE TO THE RIGHT THROUGH
2600                          ;                    THE TEXT BUFFER.
2610                          ;
```

```
3EC3   9D          PREVKY .BYTE $9D      THIS KEY SELECTS THE PREVIOUS
                    ;                    CHARACTER IN THE BUFFER.
                    ;                    $9D IS THE LEFT-ARROW.
                    ;                    THUS, PRESS LEFT-ARROW TO
                    ;                    MOVE TO THE LEFT THROUGH
                    ;                    THE TEXT BUFFER.
                    ;
3EC4   10          PRTKEY .BYTE $10      THIS KEY PRINTS THE BUFFER.
                    ;                    $10 IS CONTROL-P.  THUS,
                    ;                    PRESS CONTROL-P TO PRINT
                    ;                    THE BUFFER.
                    ;
3EC5   14          RUBKEY .BYTE $14      THIS KEY RUBS OUT THE
                    ;                    CURRENT CHARACTER.  THUS, PRESS
                    ;                    THE  DELETE KEY TO DELETE THE
                    ;                    CURRENT CHARACTER.
                    ;
3EC6   51          QUITKY .BYTE 'Q'      TWO QUIT KEYS IN A ROW
                    ;                    CAUSE THE EDITOR TO RETURN
                    ;                    TO ITS CALLER.
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;                    OTHER VARIABLES:
                    ;
3EC7   00          TEMPCH .BYTE 0        THIS BYTE USED BY EDITIT.
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ; **********************************************
                    ;
                    ;          TEXT EDITOR: UPDATE SUBROUTINE
                    ;
                    ; **********************************************
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
3EC8   20E032      EDITIT JSR GETKEY     GET A KEYSTROKE FROM USER
                    ;                    USER.
3ECB   CDC63E             CMP QUITKY     IS IT THE "QUIT" KEY?
3ECE   D017               BNE DO.KEY     IF NOT, DO WHAT THE KEY
```

```
                                            REQUIRES.
                        ;
                        ;
3ED0   48                     PHA           IF IT IS THE "QUIT" KEY, SAVE
3ED1   20E032                 JSR GETKEY    IT AND GET A NEW KEY FROM
                                            USER.
                        ;
3ED4   CDC63E                 CMP QUITKY    IS THIS A "QUIT" KEY, TOO?
3ED7   D004                   BNE NOTEND    IF NOT, THEN THIS IS NOT THE
                                            END OF THE EDIT SESSION.
                        ;
                        ;
                        ;                   END THE EDT SESSION?
3ED9   68               ENDEDT PLA          POP FIRST "QUIT" KEY FROM
                                            STACK.
                        ;
3EDA   68                     PLA           POP RETURN ADDRESS TO
3EDB   68                     PLA           EDITOR'S TOP LEVEL.
3EDC   60                     RTS           RETURN TO EDITOR'S CALLER.
                        ;
3EDD   8DC73E           NOTEND STA TEMPCH   SAVE TH KEY THAT FOLLOWED
                                            THE "QUIT" KEY.
                        ;
3EE0   68                     PLA           POP FIRST "QUIT" KEY FROM STACK.
3EE1   20E73E                 JSR DO.KEY    DO WHAT IT REQUIRES.
3EE4   ADC73E                 LDA TEMPCH    RECOVER THE KEY THAT FOLLOWED
                                            THE "QUIT" KEY.
                        ;
                        ;
                        ;                   "DO.KEY" DOES WHAT THE KEY
                        ;                   IN THE ACCUMULATOR REQUIRES:
                        ;
3EE7   CDC13E           DO.KEY CMP MODEKY   IS IT THE "CHANGE MODE" KEY?
3EEA   D00B                   BNE IFNEXT    IF NOT, PERFORM NEXT TEST.
3EEC   CE013E                 DEC EDMODE    IF SO, CHANGE THE EDITOR'S
3EEF   1005                   BPL DO.END    MODE.
3EF1   A901                   LDA #1
3EF3   8D013E                 STA EDMODE
3EF6   60               DO.END RTS          RETURN TO CALLER.
                        ;
                        ;
3EF7   CDC23E           IFNEXT CMP NEXTKY   IS IT THE "NEXT" KEY?
3EFA   D004                   BNE IFPREV    IF NOT, PERFORM NEXT TEST.
                        ;
3EFC   20793F                 JSR NEXTCH    IF SO, ADVANCE TO NEXT
                                            CHARACTER...
                        ;
3EFF   60                     RTS           ...AND RETURN.
                        ;
                        ;
3F00   CDC33E           IFPREV CMP PREVKY   IS IT THE "PREVI0US" KEY?
3F03   D004                   BNE IF.RUB    IF NOT, PERFORM NEXT TEST.
3F05   20873F                 JSR PREVSL    IF SO, BACK UP TO PREVIOUS
3F08   60                     RTS           CHARACTER AND RETURN.
                        ;
                        ;
3F09   CDC53E           IF.RUB CMP RUBKEY   IS IT THE "RUBOUT" KEY?
3F0C   D004                   BNE IF.PRT    IF NOT, PERFORM NEXT TEST.
3F0E   20DD3F                 JSR DELETE    IF SO, DELETE CURRENT
3F11   60                     RTS           CHARACTER AND RETURN.
                        ;
```

```
3700                     ;
3710  3F12  CDC43E       IF.PRT CMP PRTKEY      IS IT THE "PRINT" KEY?
3720  3F15  D004                BNE IFFLSH      IF NOT, PERFORM NEXT TEST.
3730  3F17  20C53F              JSR PRTBUF      IF SO, PRINT THE BUFFER...
3740  3F1A  60                  RTS             ...AND RETURN.
3750                     ;
3760                     ;
3770                     ;
3780  3F1B  CDC03E       IFFLSH CMP FLSHKY      IS IT THE "FLUSH" KEY?
3790  3F1E  D004                BNE CHARKY      IF NOT, IT MUST BE A CHARACTER
3800                     ;                      KEY.
3810  3F20  20B43F              JSR FLUSH       IF SO, FLUSH THE BUFFER.
3820  3F23  60                  RTS             AND RETURN.
3830                     ;
3840                     ;
3850                     ;
3860                     ;
3870                     ;    OK.  IT'S NOT AN EDITOR FUNCTION KEY, SO IT
3880                     ; MUST BE A CHARACTER KEY.  DEPENDING ON THE
3890                     ; CURRENT MODE, WE'LL EITHER INSERT OR OVERSTRIKE
3900                     ;                      THE CURRENT CHARACTER.
3910                     ;
3920  3F24  AE013E       CHARKY LDX EDMODE      ARE WE IN OVERSTRIKE MODE?
3930  3F27  F004                BEQ STRIKE      IF SO, OVERSTRIKE THE CURRENT
3940                     ;                      CHARACTER.
3950  3F29  20343F              JSR INSERT      IF NOT, INSERT THE CHARACTER.
3960  3F2C  60                  RTS             RETURN.
3970                     ;
3980  3F2D  202D33       STRIKE JSR PUT.SL      REPLACE CURRENT CHARACTER
3990                     ;                      WITH NEW CHARACTER.
4000  3F30  207D37              JSR NEXTSL      SELECT NEXT CHARACTER.
4010  3F33  60                  RTS             RETURN.
4020                     ;
4030                     ;
4040                     ;
4050                     ;
4060                     ;
4070  3F34  48           INSERT PHA             SAVE THE CHARACTER TO BE
4080                     ;                      INSERTED, WHILE WE MAKE ROOM
4090                     ;                      FOR IT IN THE BUFFER...
4100  3F35  201235              JSR PUSHSL      SAVE THE CURRENT ADDRESS.
4110  3F38  AD5335              LDA SA+1        SAVE THE BUFFER'S ADDRESS.
4120  3F3B  48                  PHA
4130  3F3C  AD5235              LDA SA
4140  3F3F  48                  PHA
4150                     ;
4160                     ;
4170  3F40  AD5535              LDA EA+1        SAVE BUFFER'S END ADDRESS.
4180  3F43  48                  PHA
4190  3F44  AD5435              LDA EA
4200  3F47  48                  PHA
4210                     ;
4220                     ;
4230  3F48  206136              JSR SAHERE      SET SA EQUAL TO SELECT, SO
```

```
                 ;                   CURRENT LOCATION WILL BE
                 ;                   START OF THE BLOCK WE'LL
                 ;                   MOVE.
                 ;
                 ;
3F4B  207D37            JSR NEXTSL   ADVANCE TO NEXT CHARACTER
                 ;                   POSITION IN THE BUFFER.
3F4E  3011              BMI ENDINS   IF WE'RE AT THE END OF THE
                 ;                   BUFFER, WE'LL OVERSTRIKE
                 ;                   INSTEAD OF INSERTING.
                 ;
                 ;
3F50  20E238            JSR DAHERE   SET DEST EQUAL TO SELECT.
                 ;                   DESTINATION OF BLOCK MOVE
                 ;                   WILL BE ONE BYTE ABOVE
                 ;                   BLOCK'S INITIAL LOCATION.
                 ;
                 ;
3F53  AD5435            LDA EA       DECREMENT END ADDRESS
3F56  D003              BNE DEC.EA
3F58  CE5535            DEC EA+1
3F5B             DEC.EA
3F5B  CE5435            DEC EA
                 ;
                 ;
                 ;
3F5E  20D637     OPENUP JSR MOV.EA   OPEN UP ONE BYTE OF SPACE
                 ;                   AT CURRENT CHARACTER'S
                 ;                   LOCATION, BY MOVING TO DEST
                 ;                   THE BLOCK SPECIFIED BY SA, EA.
                 ;
                 ;
3F61  68         ENDINS PLA          RESTORE EA SO IT POINTS
3F62  8D5435            STA EA       TO END OF BUFFER.
3F65  68                PLA
3F66  8D5535            STA EA+1
                 ;
                 ;
3F69  68                PLA          RESTORE SA SO IT POINTS TO
3F6A  8D5235            STA SA       START OF BUFFER.
3F6D  68                PLA
3F6E  8D5335            STA SA+1
                 ;
                 ;
3F71  202B35            JSR POP.SL   RESTORE SELECT SO IT POINTS
                 ;                   TO CURRENT CHARACTER POSITION.
                 ;
                 ;
3F74  68                PLA          RESTORE NEW CHARACTER TO
                 ;                   ACCUMULATOR.  WE'VE CREATED
                 ;                   A ONE-BYTE SPACE FOR IT, SO
3F75  202D3F            JSR STRIKE   WE NEED ONLY OVERSTRIKE IT
3F78  60                RTS          AND RETURN.
3F79  209532     NEXTCH JSR GET.SL   GET CURRENT CHARACTER.
```

```
3F7C  C9FF             CMP #ETX         IS IT END OF TEXT CHARACTER?
3F7E  F004             BEQ AN.ETX       IF SO, RETURN TO CALLER,
                ;                       BEARING A NEGATIVE RETURN CODE.
                ;
3F80  207D37           JSR NEXTSL       IF NOT, SELECT NEXT BYTE IN
                ;                       BUFFER.
3F83  60               RTS              RETURN PLUS IF WE INCREMENTED
                ;                       SELECT; MINUS IF SELECT
                ;                       ALREADY EQUALLED EA.
                ;
3F84  A9FF      AN.ETX LDA #$FF         SINCE WE'RE ON AN ETX, WE
3F86  60               RTS              WILL RETURN MINUS, WITHOUT
                ;                       INCREMENTING SELECT.
                ;
                ;
                ;
                ;
3F87  38        PREVSL SEC              PREPARE TO COMPARE.
3F88  AD5335           LDA SA+1         IS SELECT IN A HIGHER PAGE
3F8B  CD0632           CMP SELECT+1     THAN START OF BUFFER?
3F8E  900C             BCC SL.OK        IF SO, SELECT MAY BE DECREMENTED
3F90  D010             BNE NOT.OK       IF SELECT IS IN A LOWER
                ;                       PAGE THAN SA, IT'S NOT OK.
                ;
                ;                       SELECT IS IN SAME PAGE AS SA.
3F92  AD5235           LDA SA           IS SELECT>SA?
3F95  CD0532           CMP SELECT
3F98  F017             BEQ NO.DEC       IF SELECT EQUALS SA, DON'T
                ;                       DECREMENT SELECT.
3F9A  B006             BCS NOT.OK       IF SELECT<SA, DON'T DECREMENT
                ;                       SELECT.
3F9C  201A33    SL.OK  JSR DEC.SL       SELECT>SA, SO WE MAY
                ;                       DECREMENT SELECT AND IT
                ;                       WILL REMAIN IN THE BUFFER.
3F9F  A900             LDA #0           SET A POSITIVE RETURN CODE...
3FA1  60               RTS              ...AND RETURN.
                ;
                ;
3FA2  AD5235    NOT.OK LDA SA           SINCE SELECT<SA, IT IS NOT
3FA5  8D0532           STA SELECT       EVEN IN THE EDIT BUFFER.  SO
3FA8  AD5335           LDA SA+1         MAKE SELECT LEGAL, BY SETTING
3FAB  8D0632           STA SELECT+1     IT EQUAL TO SA.
3FAE  A900             LDA #0           SET A POSITIVE RETURN CODE...
3FB0  60               RTS              ...AND RETURN.
                ;
                ;
3FB1  A9FF      NO.DEC LDA #$FF         SELECT EQUALS SA, SO CHANGE
3FB3  60               RTS              NOTHING.  RETURN WITH
                ;                       NEGATIVE RTURN CODE.
                ;
                ;
                ;
3FB4  209A37    FLUSH  JSR GOTOSA       SET SELECT EQUAL TO SA.
3FB7  A9FF      FLOOP  LDA #ETX         PUT AN ETX CHARACTER
```

```
3FB9   202D33            JSR PUT.SL     INTO THE BUFFER.
3FBC   207D37            JSR NEXTSL     ADVANCE TO NEXT POSITION IN
                 ;                      BUFFER.
3FBF   10F6              BPL FLOOP      IF WE HAVEN'T REACHED END
                 ;                      OF BUFFER, PUT AN ETX INTO
                 ;                      THIS POSITION, TOO.
                 ;
3FC1   209A37            JSR GOTOSA     HAVING FILLED BUFFER WITH
                 ;                      ETC CHARACTERS, RESET SELECT
                 ;                      TO BEGINNING OF BUFFER.
3FC4   60                RTS            RETURN.
3FC5   209A37    PRTBUF  JSR GOTOSA     SET SELECT TO START OF BUFFER
3FC8   201434            JSR PR.ON      SELECT PRINTER FOR OUTPUT.
3FCB   209532    PRLOOP  JSR GET.SL     GET CURRENT CHARACTER.
3FCE   C9FF              CMP #ETX       IS IT ETX?
3FD0   F008              BEQ ENDPRT     IF SO, WE'RE DONE.
3FD2   204034            JSR PR.CHR     IF NOT, PRINT IT.
3FD5   207D37            JSR NEXTSL     SELECT NEXT CHARACTER
3FD8   10F1              BPL PRLOOP     IF WE HAVEN'T REACHED THE
                 ;                      END OF THE BUFFER, HANDLE
                 ;                      THE CURRENT CHARACTER AS BEFORE.
3FDA   4C1A34    ENDPRT  JMP PR.OFF     HAVING REACHED END OF MESSAGE
                 ;                      OR END OF BUFFER, RETURN TO
                 ;                      CALLER OF EDITIT, DESELECTING
                 ;                      THE PRINTER AS WE DO SO.
                 ;
                 ;
3FDD   201235    DELETE  JSR PUSHSL     SAVE CURRENT ADDRESS.
3FE0   AD5335            LDA SA+1       SAVE BUFFER'S START ADDRESS.
3FE3   48                PHA
3FE4   AD5235            LDA SA
3FE7   48                PHA
                 ;
3FE8   20E238            JSR DAHERE     SET DEST EQUAL TO SELECT,
                 ;                      BECAUSE WE'LL MOVE A BLOCK OF
                 ;                      TEXT DOWN TO HERE, TO CLOSE UP
                 ;                      THE BUFFER AT THE CURRENT
                 ;                      CHARACTER.
3FEB   207D37            JSR NEXTSL     ADVANCE BY ONE BYTE THROUGH
                 ;                      BUFFER, IF POSSIBLE.
3FEE   206136            JSR SAHERE     SET SA EQUAL TO SELECT, BECAUSE
                 ;                      THIS IS THE START OF THE BLOCK
                 ;                      WE'LL MOVE DOWN.
                 ;                      NOTE: THE ENDING ADDRESS OF
                 ;                      THE BLOCK IS THE END ADDRESS
                 ;                      OF THE TEXT BUFFER.
3FF1   20D637            JSR MOV.EA     MOVE BLOCK SPECIFIED BY
                 ;                      SA, EA TO DEST.
                 ;
                 ;
3FF4   68                PLA            RESTORE INITIAL SA (WHICH
3FF5   8D5235            STA SA         IS THE START ADDRESS OF THE
3FF8   68                PLA            TEXT BUFFER, NOT OF THE BLOCK
3FF9   8D5335            STA SA+1       WE JUST MOVED.)
```

```
5860   3FFC   202B35            JSR POP.SL      RESTORE CURRENT ADDRESS.
5870   3FFF   60                RTS             RETURN TO CALLER.
```

CROSS REFERENCE LISTING:

AN.ETX	3F84	CHARKY	3F24	CR	000D	DAHERE	38E2
DEC.EA	3F5B	DEC.SL	331A	DELETE	3FDD	DEST	37B2
DO.END	3EF6	DO.KEY	3EE7	EA	3554	EDITIT	3EC8
EDKEYS	3EC0	EDMODE	3E01	EDPAGE	3E00	ENDEDT	3ED9
ENDINS	3F61	ENDPRT	3FDA	ETX	00FF	FLOOP	3FB7
FLSHKY	3EC0	FLUSH	3FB4	GET.SL	3295	GETKEY	32E0
GOTOSA	379A	HEX.PG	3500	IF.PRT	3F12	IF.RUB	3F09
IFFLSH	3F1B	IFNEXT	3EF7	IFPREV	3F00	INC.SL	330D
INSERT	3F34	LF	000A	MODEKY	3EC1	MOV.EA	37D6
MOVERS	37B0	NEXTCH	3F79	NEXTKY	3EC2	NEXTSL	377D
NO.DEC	3FB1	NOT.OK	3FA2	NOTEND	3EDD	OPENUP	3F5E
POP.SL	352B	PR.CHR	3440	PR.OFF	341A	PR.ON	3414
PREVKY	3EC3	PREVSL	3F87	PRLOOP	3FCB	PRPAGE	3400
PRTBUF	3FC5	PRTKEY	3EC4	PUSHSL	3512	PUT.SL	332D
QUITKY	3EC6	RUBKEY	3EC5	SA	3552	SAHERE	3661
SELECT	3205	SL.OK	3F9C	STRIKE	3F2D	TEMPCH	3EC7
TEX	007F	VISMON	3207	VMPAGE	3200		

Appendix C12:

Extending the Visible Monitor

```
;     APPENDIX C12: ASSEMBLER LISTING OF
;                   VISIBLE MONITOR EXTENSIONS
;
;
;
;
;
; SEE CHAPTER 12 OF TOP-DOWN ASSEMBLY LANGUAGE
; PROGRAMMING FOR YOUR COMMODORE 64 AND VIC-20
;
;
;               BY KEN SKIER
;
;
;       COPYRIGHT (C) 1984 BY KENNETH SKIER
;          LEXINGTON, MASSACHUSETTS
;
;
;
;
;
;
;
;
;
;
;
;
;
;
;
;
;
;
;
;
; ********************************************
;
;            EXTERNAL ADDRESSES
;
; ********************************************
;
;
;
;
;
;
   PRPAGE = $3400
;                  STARTING PAGE OF PRINT
;                  UTILITIES.
;
   PRINTR = PRPAGE
   USER   = PRPAGE+2
```

```
                                  ;
                                  ;
                                       HEX.PG = $3500
                                  ;                 ADDRESS OF PAGE IN WHICH
                                  ;                 HEXDUMP CODE STARTS.
                                  ;
                                       TVDUMP = HEX.PG+$57
                                       PRDUMP = HEX.PG+$A8
                                  ;
                                  ;
                                       DSPAGE = $3900
                                  ;                 STARTING PAGE OF DISASSEMBLER
                                  ;
                                       TV.DIS = DSPAGE+9
                                       PR.DIS = DSPAGE+$26
                                  ;
                                       MOVERS = $37B0
                                  ;                 START OF MOVE OBJECT CODE.

                                       MOVER  = MOVERS+4
                                  ;
                                  ;
                                       EDPAGE = $3E00
                                  ;                 ADDRESS OF PAGE IN WHICH
                                  ;                 EDITOR CODE BEGINS.

                                       EDITOR = EDPAGE+2
                                  ;
                                  ;
                                  ;
                                  ;
                                  ;
                                  ;
                                  ;
                                  ;
0000   = 30B0                          * =  $30B0
                                  ;
                                  ;
                                  ; *****************************************
                                  ;
                                  ;     EXTENSIONS TO THE VISIBLE MONITOR
                                  ;
                                  ; *****************************************
                                  ;
                                  ;
                                  ;
30B0   C950                       EXTEND CMP #'P'       IS IT THE 'P' KEY?
30B2   D009                              BNE IF.U       IF NOT, PERFORM NEXT TEST.
30B4   AD0034                            LDA PRINTR     IF SO, TOGGLE THE PRINTER
30B7   49FF                              EOR #$FF       FLAG...
30B9   8D0034                            STA PRINTR
30BC   60                                RTS            AND RETURN TO CALLER.
                                  ;
30BD   C955                       IF.U   CMP #'U'       IS IT THE 'U' KEY?
```

```
30BF   D009             BNE IF.H       IF NOT, PERFORM NEXT TEST.
30C1   AD0234           LDA USER       IF SO, TOGGLE THE USER-
30C4   49FF             EOR #$FF       PROVIDED OUTPUT FLAG...
30C6   8D0234           STA USER
30C9   60               RTS            AND RETURN.
                ;
30CA   C948     IF.H    CMP #'H'       IS IT THE 'H' KEY?
30CC   D00D             BNE IF.M       IF NOT, PERFORM NEXT TEST.
30CE   AD0034           LDA PRINTR     IS THE PRINTER SELECTED?
30D1   D004             BNE NEXT.1     IF SO, PRINT A HEXDUMP.
30D3   205735           JSR TVDUMP     IF NOT, DUMP TO SCREEN...
30D6   60               RTS            AND RETURN.
                ;
30D7   20A835   NEXT.1  JSR PRDUMP     PRINT A HEXDUMP...
30DA   60               RTS            ...AND RETURN.
                ;
30DB   C94D     IF.M    CMP #'M'       IS IT THE 'M' KEY?
30DD   D004             BNE IF.DIS     IF NOT, PRFORM NEXT TEST.
30DF   20B437           JSR MOVER      IF SO, LET USER SPECIFY AND
30E2   60               RTS            AND MOVE A BLOCK OF MEMORY.
                ;
30E3   C93F     IF.DIS  CMP #'?'       IS IT THE '?' KEY?
30E5   D00D             BNE IF.T       IF NOT, PERFORM NEXT TEST.
30E7   AD0034           LDA PRINTR     IS THE PRINTER SELECTED?
30EA   D004             BNE NEXT.2     IF SO, PRINT A DISASSEMBLY.
30EC   200939           JSR TV.DIS     IF NOT, DISASSEMBLE TO THE
30EF   60               RTS            SCREEN AND RETURN.
                ;
30F0   202639   NEXT.2  JSR PR.DIS     PRINT A DISASSEMBLY...
30F3   60               RTS            AND RETURN.
                ;
30F4   C954     IF.T    CMP #'T'       IS IT THE 'T' KEY?
30F6   D004             BNE EXIT       IF NOT, RETURN.
30F8   20023E           JSR EDITOR     IF SO, CALL THE SIMPLE
30FB   60               RTS            TEXT EDITOR AND RETURN.
                ;
30FC   60       EXIT    RTS            EXTEND THE VISIBLE MONITOR
                ;                      EVEN FURTHER BY REPLACING
                ;                      THIS 'RTS' WITH A 'JMP' TO
                ;                      MORE TEST-AND-BRANCH CODE.
```

CROSS REFERENCE LISTING:

```
DSPAGE 3900     EDITOR 3E02     EDPAGE 3E00     EXIT   30FC
EXTEND 30B0     HEX.PG 3500     IF.DIS 30E3     IF.H   30CA
IF.M   30DB     IF.T   30F4     IF.U   30BD     MOVER  37B4
MOVERS 37B0     NEXT.1 30D7     NEXT.2 30F0     PR.DIS 3926
PRDUMP 35A8     PRINTR 3400     PRPAGE 3400     TV.DIS 3909
TVDUMP 3557     USER   3402
```

Appendix C13:

System Data Block for the Commodore VIC-20

```
;           APPENDIX C13: ASSEMBLER LISTING OF
;                         SYSTEM DATA BLOCK
;                         FOR THE COMMODORE VIC-20
;
;
;
;
;    SEE APPENDIX B1 OF TOP-DOWN ASSEMBLY LANGUAGE
;    PROGRAMMING FOR YOUR COMMODORE 64 AND VIC-20
;
;
;                          BY KEN SKIER
;
;
;              COPYRIGHT (C) 1984 BY KENNETH SKIER
;                     LEXINGTON, MASSACHUSETTS
;
;
;
;
;
;
;
;
;           NOTE------->  THE VIC-20 MUST HAVE AT LEAST
;                         8K OF EXPANSION RAM, BEGINNING
;                         AT $2000.  (INSTALL COMMODORE
;                         VIC-1110 9K MEMORY EXPANDER OR
;                         VIC-1111 16K MEMORY EXPANDER.)
;
;                         THE VISIBLE MONITOR WILL NOT
;                         RUN IN AN UNEXPANDED VIC, OR A
;                         VIC WITH ONLY 3K OF ADDITIONAL
;                         RAM.
;
;
;
;
;
;
        TV.PTR = $FB       POINTER TO CURRENT SCREEN
;                          LOCATION.
;
        VISMON = $3207     TOP LEVEL OF THE VISIBLE
;                          MONITOR.
;
;
        HEX.PG = $3500     ADDRESS OF PAGE IN WHICH
;                          HEXDUMP CODE STARTS.
;
        MASK    = HEX.PG+$51
        SA      = HEX.PG+$52
        EA      = SA+2
;
;
```

```
                  ;                  VIC KERNAL ROUTINES:
                  ;
                       CHKOUT = $FFC9
                       CHROUT = $FFD2
                       CLOSE  = $FFC3
                       OPEN   = $FFC0
                       SAVE   = $FFD8
                       SETLFS = $FFBA
                       SETNAM = $FFBD
                  ;
                  ;
                  ;
                  ;
                  ;
                  ; ******************************************
                  ;
                  ;              SCREEN PARAMETERS
                  ;
                  ; ******************************************
                  ;
                  ;
                  ;
                  ;
                  ;
0000   = 3000              * =  $3000
                  ;
                  ;
                  ;
                  ;
                  ;
3000   0010      HOME   .WORD $1000     THIS IS THE ADDRESS OF THE
                  ;                     CHARACTER IN THE UPPER LEFT
                  ;                     CORNER OF THE SCREEN, IN  A
                  ;                     VIC WITH AT LEAST 8K OF
                  ;                     EXPANSION MEMORY.
3002   16        ROWINC .BYTE 22        ADDRESS DIFFERENCE FROM ONE
                  ;                     ROW TO THE NEXT.
3003   15        TVCOLS .BYTE 21        NUMBER OF COLUMNS ON SCREEN,
                  ;                     COUNTING FROM ZERO.
3004   18        TVROWS .BYTE 24        NUMBER OF ROWS ON SCREEN,
                  ;                     COUNTING FROM ZERO.
3005   11        HIPAGE .BYTE $11       HIGHEST PAGE IN SCREEN MEMORY.
3006   20        BLANK  .BYTE $20       VIC DISPLAY CODE FOR A BLANK.
                  ;                     (IN NORMAL VIDEO MODE.)
3007   1E        ARROW  .BYTE $1E       VIC DISPLAY CODE FOR UP-ARROW.
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
```

```
                      ; ******************************************
                      ;
                      ;           INPUT/OUTPUT VECTORS
                      ;
                      ; ******************************************
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
3008  3530           ROMKEY .WORD VICKEY   POINTER TO ROUTINE THAT GETS
                      ;                    AN ASCII CHARACTER FROM THE
                      ;                    KEYBOARD.  (NOTE: VICKEY
                      ;                    CALLS A ROM SUBROUTINE, BUT
                      ;                    VICKEY IS NOT A VIC ROM
                      ;                    SUBROUTINE.)
                      ;
                      ;
300A  3C30           ROMTVI .WORD VICTVT   POINTER TO ROUTINE TO PRINT
                      ;                    AN ASCII CHARACTER ON THE SCREEN
                      ;
                      ;
300C  4130           ROMPRT .WORD VICPRT   POINTER TO ROUTINE TO SEND AN
                      ;                    ASCII CHARACTER TO THE PRINTER
                      ;
                      ;
300E  1030           USROUT .WORD DUMMY    POINTER TO USER-WRITTEN OUTPUT
                      ;                    ROUTINE.  (SET HERE TO DUMMY
                      ;                    UNTIL YOU SET IT TO POINT
                      ;                    TO YOUR OWN CHARACTER-OUTPUT
                      ;                    ROUTINE.)
                      ;
                      ;
3010  60             DUMMY  RTS            THIS IS A DUMMY SUBROUTINE.
                      ;                    IT DOES NOTHING BUT RETURN.
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ;
                      ; ******************************************
                      ;
                      ;    CONVERT ASCII CHARACTER TO DISPLAY CODE
                      ;
                      ; ******************************************
                      ;
                      ;
                      ;
                      ;
                      ;
3011                 FIXCHR                A CHARACTER IS IN A.  WE
                      ;                    MUST CONVERT IT TO PROPER
```

```
2620                     ;                      VIC DISPLAY CODE.
2630                     ;
2640                     ;                      BUT FIRST, PUT A COLOR CODE
2650                     ;                      IN APPROPRIATE BYTE OF
2660                     ;                      COLOR MEMORY.  (OTHERWISE,
2670                     ;                      THAT BYTE IN COLOR MEMORY
2680                     ;                      MIGHT BE ZERO, RENDERING
2690                     ;                      THE CHARACTER INVISIBLE.)
2700                     ;
2710  3011  48                  PHA             SAVE THE CHARACTER TO BE
2720                     ;                      DISPLAYED.
2730  3012  A5FC                LDA TV.PTR+1    SAVE HIGH BYTE...
2740  3014  48                  PHA             ...OF TV.PTR.
2750  3015  18                  CLC             MAKE TV.PTR POINT
2760  3016  6984                ADC #$84        TO APPROPRIATE BYTE
2770  3018  85FC                STA TV.PTR+1    OF COLOR MEMORY.
2780  301A  A000                LDY #0
2790  301C  AD8602              LDA $286        GET CURRENT COLOR CODE.
2800
2810                     ;                      STORE IT IN APPROPRIATE
2820                     ;                      BYTE OF COLOR MEMORY:
2830  301F  91FB                STA (TV.PTR),Y
2840                     ;
2850  3021  68                  PLA             RESTORE HIGH BYTE OF TV.PTR
2860  3022  85FC                STA TV.PTR+1    TO ITS ORIGINAL VALUE.
2870
2880  3024  68                  PLA             RETRIEVE CHARACTER TO BE
2890                     ;                      DISPLAYED.
2900                     ;
2910  3025  38                  SEC             PREPARE TO COMPARE.
2920  3026  C940                CMP #$40        IS IT LESS THAN $40?  (IS
2930                     ;                      IT A NUMBER OR PUNCTUATION
2940                     ;                      MARK?)
2950  3028  900A                BCC FIXEND      IF SO, NO CONVERSION NEEDED.
2960                     ;
2970  302A  C960                CMP #$60        IS IT IN THE RANGE $40...$5F?
2980                     ;
2990  302C  9003                BCC SUB.40      IF SO, SUBTRACT $40 TO
3000                     ;                      CONVERT FROM ASCII TO VIC.
3010                     ;
3020                     ;                      IT'S > $5F.
3030
3040  302E  E920                SBC #$20        SUBTRACT $20 TO CONVERT
3050                     ;                      LOWER CASE ASCII TO VIC CODE.
3060                     ;
3070  3030  60                  RTS             AND RETURN.
3080                     ;
3090                     ;
3100  3031  38           SUB.40 SEC             PREPARE TO SUBTRACT.
3110  3032  E940                SBC #$40        SUBTRACT $40 TO CONVERT ASCII
3120                     ;                      UPPER CASE CHAR TO VIC CODE.
3130  3034  60           FIXEND RTS             RETURN, WITH A HOLDING
3140                     ;                      VIC DISPLAY CODE FOR ASCII
3150                     ;                      ORIGINALLY IN A.
```

```
                          ;
                          ;
                          ;
                          ;
                          ;
                          ; ******************************************
                          ;
                          ;    GET AN ASCII CHARACTER FROM THE KEYBOARD
                          ;
                          ; ******************************************
                          ;
                          ;
                          ;
                          ;
3035   20E4FF         VICKEY JSR $FFE4       GET A KEYBOARD CHARACTER.
3038   AA                    TAX             IS IT ZERO?
3039   F0FA                  BEQ VICKEY      ZERO MEANS NO KEY, SO
                          ;                  SCAN AGAIN.
                          ;
303B   60                    RTS             RETURN WITH ASCII CHARACTER
                          ;                  FROM THE KEYBOARD.
                          ;
                          ;
                          ;
                          ;
                          ;
                          ; ******************************************
                          ;
                          ;    PRINT AN ASCII CHARACTER ON THE SCREEN
                          ;
                          ; ******************************************
                          ;
                          ;
                          ;
                          ;
303C   A201           VICTVT LDX #1          WE'LL DEFINE LOGICAL FILE #1
                          ;                  AS AN OUTPUT CHANNEL.
                          ;
303E   4C4330                JMP OUTCHR      OUTPUT THE CHARACTER IN A ON
                          ;                  LOGICAL FILE "X".
                          ;
                          ;
                          ;
                          ;
                          ;
                          ; ******************************************
                          ;
                          ;    PRINT AN ASCII CHARACTER ON A PRINTER
                          ;
                          ; ******************************************
                          ;
                          ;
                          ;                  THIS PROCEDURE ASSUMES THAT
                          ;                  THE USER HAS USED BASIC TO
```

```
                  ;                         OPEN A DEVICE OR FILE AS
                  ;                         LOGICAL FILE #2, BEFORE
                  ;                         CALLING THE VISIBLE MONITOR.
                  ;                         LOGICAL FILE #2 MIGHT BE A
                  ;                         PRINTER, OR THE RS-232 PORT,
                  ;                         OR EVEN A DISK OR CASSETTE
                  ;                         FILE.  THE IMPORTANT THING IS
                  ;                         THAT IT'S OPEN, SO WE MAY
                  ;                         OUTPUT TEXT TO IT.
                  ;
                  ;
                  ;
                  ;
3041  A202       VICPRT LDX #2              WE'LL DEFINE LOGICAL FILE #2
                  ;                         AS AN OUTPUT CHANNEL.
                  ;
3043             OUTCHR                     NOW OUTPUT CHARACTER IN A ON
                  ;                         LOGICAL FILE "X":
                  ;
                  ;
3043  48                PHA                 SAVE CHARACTER TO BE OUTPUT.
                  ;
                  ;
3044  20C9FF            JSR CHKOUT          SET LOGICAL FILE "X" FOR OUTPUT.
                  ;
3047  68                PLA                 RETRIEVE CHARACTER TO BE SENT.
                  ;
3048  20D2FF            JSR CHROUT          OUTPUT CHARACTER IN A ON
                  ;                         THE CURRENTLY-OPEN CHANNEL.
                  ;
304B  60                RTS                 AND RETURN.
                  ;
                  ;
                  ;
                  ;
                  ;
                  ; ********************************************
                  ;
                  ;       SAVE A MACHINE LANGUAGE PROGRAM
                  ;                ON TAPE OR DISK
                  ;
                  ; ********************************************
                  ;
                  ;
                  ;
                  ;
                  ;                         THE FOLLOWING VARIABLES
                  ;                         MUST BE SET, EITHER BY THE
                  ;                         VISIBLE MONITOR OR BY A
                  ;                         BASIC PROGRAM, BEFORE MLSAVE
                  ;                         MAY BE CALLED.
                  ;
                  ;
304C  01         DEVICE .BYTE 1             DEVICE TO BE USED FOR SAVE.
```

```
                        ;             1 SPECIFIES DATASETTE.
                        ;             8 SPECIFIES DISK DRIVE.
                        ;             THIS IS DECIMAL ADDRESS 12364.
                        ;
304D  00                LENGTH .BYTE 0         LENGTH OF FILENAME.
                        ;             THIS IS DECIMAL ADDRESS 12365.
                        ;
304E  00000000          NAME   .BYTE 0,0,0,0   ROOM HERE (AT 12366) FOR A
3052  00000000                 .BYTE 0,0,0,0   FILENAME OF UP TO 20 CHARACTERS.
3056  00000000                 .BYTE 0,0,0,0
305A  00000000                 .BYTE 0,0,0,0
305E  00000000                 .BYTE 0,0,0,0
                        ;
                        ;------------------>   NOTE: THE POINTERS SA AND EA
                        ;                      MUST ALSO BE SET, TO THE
                        ;                      STARTING AND ENDING ADDRESSES
                        ;                      (RESPECTIVELY) OF THE PROGRAM
                        ;                      TO BE SAVED.  THEY MAY BE SET
                        ;                      MOST CONVENIENTLY BY SIMPLY
                        ;                      CALLING THE SUBROUTINE "SETADS"
                        ;                      AT $35E3 (13795 DECIMAL).
                        ;
                        ;
                        ;
                        ;
3062  A903              MLSAVE LDA #3          LOGICAL FILE NUMBER.
3064  AE4C30                   LDX DEVICE      DEVICE NUMBER.
3067  A8                       TAY             SECONDARY ADDRESS.
3068  20BAFF                   JSR SETLFS      CALL KERNAL ROUTINE "SETLFS".
                        ;                      NOW THE VIC KNOWS WHAT DEVICE
                        ;                      TO USE.
                        ;
306B  AD4D30                   LDA LENGTH      GET LENGTH OF FILENAME.
                        ;
306E  A24E                     LDX #LOW(NAME)
3070  A030                     LDY #HIGH(NAME)
                        ;                      NOW (X,Y) POINTS TO THE FILE
                        ;                      NAME.
3072  20BDFF                   JSR SETNAM      CALL KERNAL ROUTINE "SETNAM".
                        ;                      NOW THE VIC KNOWS THE NAME OF
                        ;                      THE FILE YOU WISH TO CREATE.
                        ;
3075  AD5235                   LDA SA
3078  85FD                     STA $FD
307A  AD5335                   LDA SA+1
307D  85FE                     STA $FE         NOW PTR AT $FD POINTS TO START
                        ;                      OF THE ML PROGRAM.
307F  A9FD                     LDA #$FD        NOW A HOLDS ZERO PAGE OFFSET
                        ;                      FOR THAT POINTER.
3081  AE5435                   LDX EA
3084  AC5535                   LDY EA+1        NOW (X,Y) POINTS TO THE END OF
                        ;                      THE ML PROGRAM.
                        ;                      BUT THE KERNAL ROUTINE "SAVE"
                        ;                      REQUIRES THAT (X,Y) POINT ONE
```

```
                       ;                    BYTE BEYOND THE END OF THE
                       ;                    ML PROGRAM.  SO INCREMENT
                       ;                    (X,Y):
3087  E8                     INX
3088  D001                   BNE XY.SET
308A  C8                     INY
                       ;                    NOW (X,Y) IS SET.
                       ;
308B  20D8FF           XY.SET JSR SAVE      CALL KERNAL ROUTINE "SAVE".
                       ;                    THIS ACTUALLY OPENS A FILE AND
                       ;                    STORES THE SPECIFIED PORTION OF
                       ;                    MEMORY ON THE SPECIFIED DEVICE.
                       ;
                       ;
308E  60                     RTS            RETURN TO CALLER (PRESUMABLY
                       ;                    BASIC OR THE VISIBLE MONITOR.)
                       ;
                       ;
                       ;
                       ;
                       ;
                       ; ********************************************
                       ;
                       ;        VISIBLE MONITOR: ENTRY FROM BASIC
                       ;
                       ; ********************************************
                       ;
                       ;
                       ;
                       ;
308F  A900             ENTRY   LDA #0
3091  48                       PHA
3092  28                       PLP          NOW THE STATUS REGISTER IS
                                            ZERO.
                       ;
                       ;                    OPEN THE SCREEN AS LOGICAL
                       ;                    FILE #1:
                       ;
3093  20BDFF                 JSR SETNAM     A ALREADY HOLDS $00,
                       ;                    INDICATING NO FILE NAME.
                       ;
3096  A901                   LDA #1         LOGICAL FILE NUMBER.
3098  A200                   LDX #0         DEVICE NUMBER OF THE SCREEN.
309A  A0FF                   LDY #255       (NO COMMAND.)
309C  20BAFF                 JSR SETLFS     VIC KERNAL ROUTINE "SETLFS"
                       ;
309F  20C0FF                 JSR OPEN       NOW THE SCREEN IS LOGICAL FILE
                       ;                    #1.
                       ;
                       ;
30A2  A903                   LDA #3         SET HEXDUMP MASK TO 3, SO
30A4  8D5135                 STA MASK       TVDUMP WILL OUTPUT EACH HEX
                       ;                    LINE AS FOUR SCREEN LINES.
                       ;
```

```
                           ;
30A7   200732                    JSR VISMON      CALL THE VISIBLE MONITOR.
                           ;
                           ;
                           ;                     NOW THE VISIBLE MONITOR HAS
                           ;                     RETURNED.
                           ;
                           ;                     SO CLOSE LOGICAL FILE #1:
30AA   A901                      LDA #1
30AC   20C3FF                    JSR CLOSE
                           ;
30AF   60                        RTS             RETURN TO CALLER (PRESUMABLY
                           ;                     BASIC.)
```

CROSS REFERENCE LISTING:

```
ARROW  3007      BLANK  3006      CHKOUT FFC9      CHROUT FFD2
CLOSE  FFC3      DEVICE 304C      DUMMY  3010      EA     3554
ENTRY  308F      FIXCHR 3011      FIXEND 3034      HEX.PG 3500
HIPAGE 3005      HOME   3000      LENGTH 304D      MASK   3551
MLSAVE 3062      NAME   304E      OPEN   FFC0      OUTCHR 3043
ROMKEY 3008      ROMPRT 300C      ROMTVT 300A      ROWINC 3002
SA     3552      SAVE   FFD8      SETLFS FFBA      SETNAM FFBD
SUB.40 3031      TV.PTR 00FB      TVCOLS 3003      TVROWS 3004
USROUT 300E      VICKEY 3035      VICPRT 3041      VICTVT 303C
VISMON 3207      XY.SET 308B
```

Appendix C14:

System Data Block for the Commodore 64

```
;         APPENDIX C14: ASSEMBLER LISTING OF
;                       SYSTEM DATA BLOCK
;                       FOR THE COMMODORE 64
;
;
;
;
; SEE APPENDIX B2 OF TOP-DOWN ASSEMBLY LANGUAGE
; PROGRAMMING FOR YOUR COMMODORE 64 AND VIC-20
;
;
;                       BY KEN SKIER
;
;
;          COPYRIGHT (C) 1984 BY KENNETH SKIER
;                 LEXINGTON, MASSACHUSETTS
;
;
;
;
;
;
;
;
;
;
   TV.PTR = $FB       POINTER TO CURRENT SCREEN
;                     LOCATION.
;
;
   VISMON = $3207     TOP LEVEL OF THE VISIBLE
;                     MONITOR.
;
;
   HEX.PG = $3500     ADDRESS OF PAGE IN WHICH
;                     HEXDUMP CODE STARTS.
;
   SA     = HEX.PG+$52
   EA     = SA+2
;
;
;                     C64 KERNAL ROUTINES:
;
   CHKOUT = $FFC9
   CHROUT = $FFD2
   CLOSE  = $FFC3
   OPEN   = $FFC0
   SAVE   = $FFD8
   SETLFS = $FFBA
   SETNAM = $FFBD
;
;
;
;
```

```
                          ;
                          ; *******************************************
                          ;
                          ;                SCREEN PARAMETERS
                          ;
                          ; *******************************************
                          ;
                          ;
                          ;
                          ;
                          ;
0000      = 3000                 * =  $3000
                          ;
                          ;
                          ;
                          ;
                          ;
3000  0004               HOME   .WORD $400     THIS IS THE ADDRESS OF THE
                          ;                    CHARACTER IN THE UPPER LEFT
                          ;                    CORNER OF THE SCREEN.
3002  28                 ROWINC .BYTE $28      ADDRESS DIFFERENCE FROM ONE
                          ;                    ROW TO THE NEXT.
3003  27                 TVCOLS .BYTE 39       NUMBER OF COLUMNS ON SCREEN,
                          ;                    COUNTING FROM ZERO.
3004  18                 TVROWS .BYTE 24       NUMBER OF ROWS ON SCREEN,
                          ;                    COUNTING FROM ZERO.
3005  07                 HIPAGE .BYTE $07      HIGHEST PAGE IN SCREEN MEMORY.
3006  20                 BLANK  .BYTE $20      C64 DISPLAY CODE FOR A BLANK.
                          ;                    (IN NORMAL VIDEO MODE.)
3007  1E                 ARROW  .BYTE $1E      C64 DISPLAY CODE FOR UP-ARROW.
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
                          ; *******************************************
                          ;
                          ;              INPUT/OUTPUT VECTORS
                          ;
                          ; *******************************************
                          ;
                          ;
                          ;
                          ;
                          ;
                          ;
3008  3530               ROMKEY .WORD C64KEY   POINTER TO ROUTINE THAT GETS
                          ;                    AN ASCII CHARACTER FROM THE
                          ;                    KEYBOARD.  (NOTE: C64KEY
                          ;                    CALLS A ROM SUBROUTINE, BUT
```

```
                    ;                   C64KEY IS NOT A C64 ROM
                    ;                   SUBROUTINE.)
                    ;
                    ;
300A  3C30          ROMTVT .WORD C64TVT POINTER TO ROUTINE TO PRINT
                    ;                   AN ASCII CHARACTER ON THE SCREEN
                    ;
                    ;
300C  4130          ROMPRT .WORD C64PRT POINTER TO ROUTINE TO SEND AN
                    ;                   ASCII CHARACTER TO THE PRINTER
                    ;
                    ;
300E  1030          USROUT .WORD DUMMY  POINTER TO USER-WRITTEN OUTPUT
                    ;                   ROUTINE.  (SET HERE TO DUMMY
                    ;                   UNTIL YOU SET IT TO POINT
                    ;                   TO YOUR OWN CHARACTER-OUTPUT
                    ;                   ROUTINE.)
                    ;
                    ;
3010  60            DUMMY  RTS          THIS IS A DUMMY SUBROUTINE.
                    ;                   IT DOES NOTHING BUT RETURN.
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ; ******************************************
                    ;
                    ;     CONVERT ASCII CHARACTER TO DISPLAY CODE
                    ;
                    ; ******************************************
                    ;
                    ;
                    ;
                    ;
                    ;
3011                FIXCHR              A CHARACTER IS IN A.  WE
                    ;                   MUST CONVERT IT TO PROPER
                    ;                   C64 DISPLAY CODE.
                    ;
                    ;                   BUT FIRST, PUT A COLOR CODE
                    ;                   IN APPROPRIATE BYTE OF
                    ;                   COLOR MEMORY.  (OTHERWISE,
                    ;                   THAT BYTE IN COLOR MEMORY
                    ;                   MIGHT BE ZERO, RENDERING
                    ;                   THE CHARACTER INVISIBLE.)
                    ;
3011  48                   PHA          SAVE THE CHARACTER TO BE
                    ;                   DISPLAYED.
3012  A5FC                 LDA TV.PTR+1 SAVE HIGH BYTE...
3014  48                   PHA          ...OF TV.PTR.
3015  18                   CLC          MAKE TV.PTR POINT
3016  69D4                 ADC #$D4     TO APPROPRIATE BYTE
```

```
2620   3018   85FC              STA TV.PTR+1    OF COLOR MEMORY.
2630   301A   A000              LDY #0
2640   301C   AD8602            LDA $286        GET CURRENT COLOR CODE.
2650
2660                     ;                      STORE IT IN APPROPRIATE
2670                     ;                      BYTE OF COLOR MEMORY:
2680   301F   91FB              STA (TV.PTR),Y
2690                     ;
2700   3021   68                PLA             RESTORE HIGH BYTE OF TV.PTR
2710   3022   85FC              STA TV.PTR+1    TO ITS ORIGINAL VALUE.
2720
2730   3024   68                PLA             RETRIEVE CHARACTER TO BE
2740                     ;                      DISPLAYED.
2750                     ;
2760   3025   38                SEC             PREPARE TO COMPARE.
2770   3026   C940              CMP #$40        IS IT LESS THAN $40?  (IS
2780                     ;                      IT A NUMBER OR PUNCTUATION
2790                     ;                      MARK?)
2800   3028   900A              BCC FIXEND      IF SO, NO CONVERSION NEEDED.
2810                     ;
2820   302A   C960              CMP #$60        IS IT IN THE RANGE $40...$5F?
2830                     ;
2840   302C   9003              BCC SUB.40      IF SO, SUBTRACT $40 TO
2850                     ;                      CONVERT FROM ASCII TO C64.
2860                     ;
2870                     ;                      IT'S > $5F.
2880
2890   302E   E920              SBC #$20        SUBTRACT $20 TO CONVERT
2900                     ;                      LOWER CASE ASCII TO C64 CODE.
2910                     ;
2920   3030   60                RTS             AND RETURN.
2930                     ;
2940                     ;
2950   3031   38         SUB.40 SEC             PREPARE TO SUBTRACT.
2960   3032   E940              SBC #$40        SUBTRACT $40 TO CONVERT ASCII
2970                     ;                      UPPER CASE CHAR TO C64 CODE.
2980   3034   60         FIXEND RTS             RETURN, WITH A HOLDING
2990                     ;                      C64 DISPLAY CODE FOR ASCII
3000                     ;                      ORIGINALLY IN A.
3010                     ;
3020                     ;
3030                     ;
3040                     ;
3050                     ;
3060                     ; ******************************************
3070                     ;
3080                     ;    GET AN ASCII CHARACTER FROM THE KEYBOARD
3090                     ;
3100                     ; ******************************************
3110                     ;
3120                     ;
3130                     ;
3140                     ;
3150   3035   20E4FF     C64KEY JSR $FFE4       GET A KEYBOARD CHARACTER.
```

```
3038  AA              TAX                IS IT ZERO?
3039  F0FA            BEQ C64KEY         ZERO MEANS NO KEY, SO
                  ;                      SCAN AGAIN.
                  ;
303B  60              RTS                RETURN WITH ASCII CHARACTER
                  ;                      FROM THE KEYBOARD.
                  ;
                  ;
                  ;
                  ;
                  ;
                  ; *********************************************
                  ;
                  ;    PRINT AN ASCII CHARACTER ON THE SCREEN
                  ;
                  ; *********************************************
                  ;
                  ;
                  ;
                  ;
303C  A201        C64TVT LDX #1          WE'LL DEFINE LOGICAL FILE #1
                  ;                      AS AN OUTPUT CHANNEL.
                  ;
303E  4C4330          JMP OUTCHR         OUTPUT THE CHARACTER IN A ON
                  ;                      LOGICAL FILE "X".
                  ;
                  ;
                  ;
                  ;
                  ;
                  ; *********************************************
                  ;
                  ;    PRINT AN ASCII CHARACTER ON A PRINTER
                  ;
                  ; *********************************************
                  ;
                  ;
                  ;                      THIS PROCEDURE ASSUMES THAT
                  ;                      THE USER HAS USED BASIC TO
                  ;                      OPEN A DEVICE OR FILE AS
                  ;                      LOGICAL FILE #2, BEFORE
                  ;                      CALLING THE VISIBLE MONITOR.
                  ;                      LOGICAL FILE #2 MIGHT BE A
                  ;                      PRINTER, OR THE RS-232 PORT,
                  ;                      OR EVEN A DISK OR CASSETTE
                  ;                      FILE.  THE IMPORTANT THING IS
                  ;                      THAT IT'S OPEN, SO WE MAY
                  ;                      OUTPUT TEXT TO IT.
                  ;
                  ;
                  ;
                  ;
3041  A202        C64PRT LDX #2          WE'LL DEFINE LOGICAL FILE #2
                  ;                      AS AN OUTPUT CHANNEL.
```

```
                           ;
3043               OUTCHR                          NOW OUTPUT CHARACTER IN A ON
                           ;                       LOGICAL FILE "X":
                           ;
                           ;
3043   48                        PHA               SAVE CHARACTER TO BE OUTPUT.
                           ;
                           ;
3044   20C9FF                    JSR CHKOUT        SET LOGICAL FILE "X" FOR OUTPUT.
                           ;
3047   68                        PLA               RETRIEVE CHARACTER TO BE SENT.
                           ;
3048   20D2FF                    JSR CHROUT        OUTPUT CHARACTER IN A ON
                           ;                       THE CURRENTLY-OPEN CHANNEL.
                           ;
304B   60                        RTS               AND RETURN.
                           ;
                           ;
                           ;
                           ;
                           ;
                           ;  ********************************************
                           ;
                           ;        SAVE A MACHINE LANGUAGE PROGRAM
                           ;                 ON TAPE OR DISK
                           ;
                           ;  ********************************************
                           ;
                           ;
                           ;
                           ;
                           ;                       THE FOLLOWING VARIABLES
                           ;                       MUST BE SET, EITHER BY THE
                           ;                       VISIBLE MONITOR OR BY A
                           ;                       BASIC PROGRAM, BEFORE MLSAVE
                           ;                       MAY BE CALLED.
                           ;
                           ;
304C   01          DEVICE .BYTE 1                  DEVICE TO BE USED FOR SAVE.
                           ;                       1 SPECIFIES DATASETTE.
                           ;                       8 SPECIFIES DISK DRIVE.
                           ;                       THIS IS DECIMAL ADDRESS 12364.
                           ;
304D   00          LENGTH .BYTE 0                  LENGTH OF FILENAME.
                           ;                       THIS IS DECIMAL ADDRESS 12365.
                           ;
304E   00000000    NAME   .BYTE 0,0,0,0            ROOM HERE (AT 12366) FOR A
3052   00000000           .BYTE 0,0,0,0            FILENAME OF UP TO 20 CHARACTERS.
3056   00000000           .BYTE 0,0,0,0
305A   00000000           .BYTE 0,0,0,0
305E   00000000           .BYTE 0,0,0,0
                           ;
                           ;---------------->      NOTE: THE POINTERS SA AND EA
                           ;                       MUST ALSO BE SET, TO THE
```

```
                        ;                       STARTING AND ENDING ADDRESSES
                        ;                       (RESPECTIVELY) OF THE PROGRAM
                        ;                       TO BE SAVED.  THEY MAY BE SET
                        ;                       MOST CONVENIENTLY BY SIMPLY
                        ;                       CALLING THE SUBROUTINE "SETADS"
                        ;                       AT $35E3 (13795 DECIMAL).
                        ;
                        ;
                        ;
                        ;
3062  A903              MLSAVE LDA #3           LOGICAL FILE NUMBER.
3064  AE4C30                   LDX DEVICE       DEVICE NUMBER.
3067  A8                       TAY              SECONDARY ADDRESS.
3068  20BAFF                   JSR SETLFS       CALL KERNAL ROUTINE "SETLFS".
                        ;                       NOW THE C64 KNOWS WHAT DEVICE
                        ;                       TO USE.
                        ;
306B  AD4D30                   LDA LENGTH       GET LENGTH OF FILENAME.
                        ;
306E  A24E                     LDX #LOW(NAME)
3070  A030                     LDY #HIGH(NAME)
                        ;                       NOW (X,Y) POINTS TO THE FILE
                        ;                       NAME.
3072  20BDFF                   JSR SETNAM       CALL KERNAL ROUTINE "SETNAM".
                        ;                       NOW THE C64 KNOWS THE NAME OF
                        ;                       THE FILE YOU WISH TO CREATE.
                        ;
3075  AD5235                   LDA SA
3078  85FD                     STA $FD
307A  AD5335                   LDA SA+1
307D  85FE                     STA $FE          NOW PTR AT $FD POINTS TO START
                        ;                       OF THE ML PROGRAM.
307F  A9FD                     LDA #$FD         NOW A HOLDS ZERO PAGE OFFSET
                        ;                       FOR THAT POINTER.
3081  AE5435                   LDX EA
3084  AC5535                   LDY EA+1         NOW (X,Y) POINTS TO THE END OF
                        ;                       THE ML PROGRAM.
                        ;                       BUT THE KERNAL ROUTINE "SAVE"
                        ;                       REQUIRES THAT (X,Y) POINT ONE
                        ;                       BYTE BEYOND THE END OF THE
                        ;                       ML PROGRAM.  SO INCREMENT
                        ;                       (X,Y):
3087  E8                       INX
3088  D001                     BNE XY.SET
308A  C8                       INY
                        ;                       NOW (X,Y) IS SET.
                        ;
308B  20D8FF            XY.SET JSR SAVE         CALL KERNAL ROUTINE "SAVE".
                        ;                       THIS ACTUALLY OPENS A FILE AND
                        ;                       STORES THE SPECIFIED PORTION OF
                        ;                       MEMORY ON THE SPECIFIED DEVICE.
                        ;
                        ;
308E  60                       RTS              RETURN TO CALLER (PRESUMABLY
```

```
                        ;                 BASIC OR THE VISIBLE MONITOR.)
                        ;
                        ;
                        ;
                        ;
                        ;
                        ; ****************************************
                        ;
                        ;        VISIBLE MONITOR: ENTRY FROM BASIC
                        ;
                        ; ****************************************
                        ;
                        ;
                        ;
                        ;
308F   A900             ENTRY   LDA #0
3091   48                       PHA
3092   28                       PLP               NOW THE STATUS REGISTER IS
                                                  ZERO.
                        ;
                        ;                         OPEN THE SCREEN AS LOGICAL
                        ;                         FILE #1:
                        ;
3093   20BDFF                   JSR SETNAM        A ALREADY HOLDS $00,
                        ;                         INDICATING NO FILE NAME.
                        ;
3096   A901                     LDA #1            LOGICAL FILE NUMBER.
3098   A200                     LDX #0            DEVICE NUMBER OF THE SCREEN.
309A   A0FF                     LDY #255          (NO COMMAND.)
309C   20BAFF                   JSR SETLFS        C64 KERNAL ROUTINE "SETLFS"
                        ;
309F   20C0FF                   JSR OPEN          NOW THE SCREEN IS LOGICAL FILE
                        ;                         #1.
                        ;
30A2   200732                   JSR VISMON        CALL THE VISIBLE MONITOR.
                        ;
                        ;
                        ;                         NOW THE VISIBLE MONITOR HAS
                        ;                         RETURNED.
                        ;
                        ;                         SO CLOSE LOGICAL FILE #1:
30A5   A901                     LDA #1
30A7   20C3FF                   JSR CLOSE
                        ;
30AA   60                       RTS               RETURN TO CALLER (PRESUMABLY
                        ;                         BASIC.)
```

CROSS REFERENCE LISTING:

ARROW	3007	BLANK	3006	C64KEY	3035	C64PRT	3041
C64TVT	303C	CHKOUT	FFC9	CHROUT	FFD2	CLOSE	FFC3
DEVICE	304C	DUMMY	3010	EA	3554	ENTRY	308F
FIXCHR	3011	FIXEND	3034	HEX.PG	3500	HIPAGE	3005
HOME	3000	LENGTH	304D	MLSAVE	3062	NAME	304E
OPEN	FFC0	OUTCHR	3043	ROMKEY	3008	ROMPRT	300C
ROMTVT	300A	ROWINC	3002	SA	3552	SAVE	FFD8
SETLFS	FFBA	SETNAM	FFBD	SUB.40	3031	TV.PTR	00FB
TVCOLS	3003	TVROWS	3004	USROUT	300E	VISMON	3207
XY.SET	308B						

Appendix D1:

Screen Utilities

```
APPENDIX D1:                  SCREEN UTILITIES


SEE CHAPTER 5 OF TOP-DOWN ASSEMBLY LANGUAGE PROGRAMMING
                    FOR YOUR COMMODORE 64 OR VIC-20.


DUMPING $3100-$31E1

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

3100   20 C4 31 20 2B 31 AE 03 30 AC 04 30 20 13 31 20
3110   D3 31 60 8E 2A 31 98 AA AD 06 30 AC 2A 31 91 FB
3120   88 10 FB 20 76 31 CA 10 EF 60 27 A2 00 A0 00 18
3130   90 0A AD 04 30 4A A8 AD 03 30 4A AA 38 EC 03 30
3140   90 03 AE 03 30 38 CC 04 30 90 03 AC 04 30 AD 00
3150   30 85 FB AD 01 30 85 FC 08 D8 8A 18 65 FB 90 03
3160   E6 FC 18 C0 00 F0 0B 18 6D 02 30 90 02 E6 FC 88
3170   D0 F5 85 FB 28 60 AD 02 30 18 90 05 20 9B 31 A9
3180   01 08 D8 18 65 FB 90 02 E6 FC 85 FB 38 AD 05 30
3190   C5 FC B0 05 AD 01 30 85 FC 28 60 20 11 30 A0 00
31A0   91 FB 60 48 4A 4A 4A 4A 20 B6 31 20 7C 31 68 20
31B0   B6 31 20 7C 31 60 08 D8 29 0F C9 0A 30 02 69 06
31C0   69 30 28 60 68 AA 68 A8 A5 FC 48 A5 FB 48 98 48
31D0   8A 48 60 68 AA 68 A8 68 85 FB 68 85 FC 98 48 8A
31E0   48 60

---------->  END OF APPENDIX D1  <----------
```

Appendix D2:

Visible Monitor (Top Level and Display Subroutines)

```
APPENDIX D2:                THE VISIBLE MONITOR
                        (TOP LEVEL & DISPLAY SUBS)


SEE CHAPTER 6 OF TOP-DOWN ASSEMBLY LANGUAGE PROGRAMMING
                 FOR YOUR COMMODORE 64 OR VIC-20.
```

DUMPING \$3200-\$32D3

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
3200	00	00	00	00	00	05	32	08	D8	20	12	32	20	E3	32	18
3210	90	F6	20	C4	31	20	25	32	20	35	32	20	5D	32	20	B0
3220	32	20	D3	31	60	A2	00	A0	00	20	3C	31	AE	03	30	A0
3230	03	20	13	31	60	A2	0B	A0	00	20	3C	31	A0	00	8C	52
3240	32	B9	53	32	20	7C	31	EE	52	32	AC	52	32	C0	0A	D0
3250	F0	60	0A	41	20	20	58	20	20	59	20	20	50	A2	00	A0
3260	01	20	3C	31	AD	06	32	20	A3	31	AD	05	32	20	A3	31
3270	20	7F	31	20	95	32	48	20	A3	31	20	7F	31	68	20	7C
3280	31	20	7F	31	A2	00	BD	01	32	20	A3	31	20	7F	31	E8
3290	E0	04	D0	F2	60	A5	FB	48	A6	FC	AD	05	32	85	FB	AD
32A0	06	32	85	FC	A0	00	B1	FB	A8	68	85	FB	86	FC	98	60
32B0	AC	00	32	38	C0	07	90	05	A0	00	8C	00	32	B9	CD	32
32C0	AA	A0	02	20	3C	31	AD	07	30	20	7C	31	60	03	06	08
32D0	0B	0E	11	14												

----------> END OF APPENDIX D2 <----------

Appendix D3:

Visible Monitor (Update Subroutine)

```
APPENDIX D3:              THE VISIBLE MONITOR
                          (UPDATE SUBROUTINE)


SEE CHAPTER 6 OF TOP-DOWN ASSEMBLY LANGUAGE PROGRAMMING
                 FOR YOUR COMMODORE 64 OR VIC-20.


DUMPING $32E0-$33EE

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

32E0   6C 08 30 20 E0 32 C9 1D D0 10 EE 00 32 AD 00 32
32F0   C9 07 D0 05 A9 00 8D 00 32 60 C9 9D D0 0B CE 00
3300   32 10 05 A9 06 8D 00 32 60 C9 20 D0 09 EE 05 32
3310   D0 03 EE 06 32 60 C9 0D D0 0C AD 05 32 D0 03 CE
3320   06 32 CE 05 32 60 AE 00 32 E0 02 D0 1B A8 A5 FB
3330   48 A6 FC AD 05 32 85 FB AD 06 32 85 FC 98 A0 00
3340   91 FB 86 FC 68 85 FB 60 C9 47 D0 23 AC 03 32 AE
3350   02 32 AD 04 32 48 AD 01 32 28 20 6C 33 08 8D 01
3360   32 8E 02 32 8C 03 32 68 8D 04 32 60 6C 05 32 48
3370   20 D5 33 30 4B A8 68 98 AE 00 32 D0 14 A2 03 18
3380   0E 05 32 2E 06 32 CA 10 F6 98 0D 05 32 8D 05 32
3390   60 E0 01 D0 18 29 0F 48 20 95 32 0A 0A 0A 0A 29
33A0   F0 8D AC 33 68 0D AC 33 20 2D 33 60 00 CA CA CA
33B0   A0 03 18 1E 01 32 88 10 F9 1D 01 32 9D 01 32 60
33C0   68 C9 93 D0 04 20 00 31 60 C9 51 D0 04 68 68 28
33D0   60 20 10 30 60 38 E9 30 90 0F C9 0A 90 0E E9 07
33E0   C9 10 B0 05 38 C9 0A B0 03 A9 FF 60 A2 00 60

---------->  END OF APPENDIX D3  <----------
```

Appendix D4:

Print Utilities

```
APPENDIX D4:                PRINT UTILITIES


SEE CHAPTER 7 OF TOP-DOWN ASSEMBLY LANGUAGE PROGRAMMING
                 FOR YOUR COMMODORE 64 OR VIC-20.


DUMPING $3400-$3543

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

3400    FF FF 00 20 00 00 0C 35 A9 FF 8D 01 34 60 A9 00
3410    8D 01 34 60 A9 FF 8D 00 34 60 A9 00 8D 00 34 60
3420    A9 FF 8D 02 34 60 A9 00 8D 02 34 60 20 08 34 20
3430    14 34 20 20 34 60 20 0E 34 20 1A 34 20 26 34 60
3440    C9 00 F0 24 8D 03 34 AD 01 34 F0 06 AD 03 34 20
3450    69 34 AD 00 34 F0 06 AD 03 34 20 6C 34 AD 02 34
3460    F0 06 AD 03 34 20 6F 34 60 6C 0A 30 6C 0C 30 6C
3470    0E 30 A9 0D 20 40 34 A9 0A 20 40 34 60 A9 20 20
3480    40 34 60 48 4A 4A 4A 4A 20 B6 31 20 40 34 68 20
3490    B6 31 20 40 34 60 A9 20 8E 04 34 48 AE 04 34 F0
34A0    0A CE 04 34 20 40 34 68 18 90 F0 68 60 8E 04 34
34B0    AE 04 34 F0 09 CE 04 34 20 72 34 18 90 F2 60 8E
34C0    05 34 B5 01 48 B5 00 48 AE 05 34 A1 00 C9 FF F0
34D0    0C F6 00 D0 02 F6 01 20 40 34 18 90 EB 68 95 00
34E0    68 95 01 60 68 AA 68 A8 20 12 35 8E 05 32 8C 06
34F0    32 20 0D 33 20 0D 33 20 95 32 C9 FF F0 06 20 40
3500    34 18 90 F0 AE 05 32 AC 06 32 20 2B 35 98 48 8A
3510    48 60 68 8D 06 34 68 8D 07 34 AD 06 32 48 AD 05
3520    32 48 AD 07 34 48 AD 06 34 48 60 68 8D 06 34 68
3530    8D 07 34 68 8D 05 32 68 8D 06 32 AD 07 34 48 AD
3540    06 34 48 60

---------->  END OF APPENDIX D4  <----------
```

Appendix D5:

Two Hexdump Tools

APPENDIX D5:   TWO HEXDUMP TOOLS

SEE CHAPTER 8 OF TOP-DOWN ASSEMBLY LANGUAGE PROGRAMMING
FOR YOUR COMMODORE 64 OR VIC-20.

DUMPING $3550-$37A6

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
3550	00	07	50	35	A6	37	00	20	08	34	A9	04	8D	50	35	AD
3560	05	32	29	F0	8D	05	32	20	9B	35	20	7D	34	20	7D	34
3570	20	94	35	20	0D	33	AD	05	32	2D	51	35	D0	EF	20	72
3580	34	AD	05	32	29	0F	D0	03	20	72	34	CE	50	35	D0	D7
3590	20	0E	34	60	20	95	32	20	83	34	60	AD	06	32	20	83
35A0	34	AD	05	32	20	83	34	60	20	C3	35	20	E3	35	20	9A
35B0	37	20	14	34	20	E5	36	20	3C	37	10	FB	20	72	34	20
35C0	1A	34	60	20	08	34	20	1A	34	20	E4	34	7F	0D	50	52
35D0	49	4E	54	49	4E	47	20	48	45	58	44	55	4D	50	0D	0A
35E0	0A	FF	60	20	08	34	20	E4	34	7F	0D	0A	53	45	54	20
35F0	53	54	41	52	54	49	4E	47	20	41	44	44	52	45	53	53
3600	20	41	4E	44	20	50	52	45	53	53	20	22	51	22	2E	FF
3610	20	07	32	20	61	36	20	08	34	20	E4	34	7F	0D	0A	53
3620	45	54	20	45	4E	44	20	41	44	44	52	45	53	53	20	41
3630	4E	44	20	50	52	45	53	53	20	22	51	22	2E	FF	20	07
3640	32	38	AD	06	32	CD	53	35	90	24	D0	08	AD	05	32	CD
3650	52	35	90	1A	AD	06	32	8D	55	35	AD	05	32	8D	54	35
3660	60	AD	06	32	8D	53	35	AD	05	32	8D	52	35	60	20	E4
3670	34	7F	0D	0A	0A	0A	20	45	52	52	4F	52	21	21	21	20
3680	45	4E	44	20	41	44	44	52	45	53	53	20	4C	45	53	53
3690	20	54	48	41	4E	20	53	54	41	52	54	20	41	44	44	52
36A0	45	53	53	2C	20	57	48	49	43	48	20	49	53	20	FF	20
36B0	B5	36	4C	16	36	A9	24	20	40	34	AD	53	35	20	83	34
36C0	AD	52	35	20	83	34	60	A9	24	20	40	34	AD	55	35	20
36D0	83	34	AD	54	35	20	83	34	60	20	B5	36	A9	2D	20	40

```
36E0    34 20 C7 36 60 20 E4 34 7F 0D 0A 0A 44 55 4D 50
36F0    49 4E 47 20 FF 20 D9 36 20 72 34 20 E4 34 7F 0A
3700    0A 20 20 20 20 20 20 20 20 30 20 20 31 20 20 32
3710    20 20 33 20 20 34 20 20 35 20 20 36 20 20 37 20
3720    20 38 20 20 39 20 20 41 20 20 42 20 20 43 20 20
3730    44 20 20 45 20 20 46 0D 0A 0A FF 60 20 72 34 AD
3740    05 32 48 29 0F 8D 56 35 68 29 F0 8D 05 32 20 9B
3750    35 A2 03 20 96 34 AD 56 35 F0 0D A2 03 20 96 34
3760    20 0D 33 CE 56 35 D0 F3 20 94 35 20 7D 34 20 7D
3770    37 30 09 AD 05 32 29 0F C9 00 D0 EC 60 38 AD 06
3780    32 CD 55 35 90 0B D0 0F 38 AD 05 32 CD 54 35 B0
3790    06 20 0D 33 A9 00 60 A9 FF 60 AD 52 35 8D 05 32
37A0    AD 53 35 8D 06 32 60
```

----------> END OF APPENDIX D5 <----------

Appendix D6:

Table-Driven Disassembler (Top Level and Utility Subroutines)

```
APPENDIX D6:                    DISASSEMBLER
                    (TOP LEVEL & UTILITY SUBROUTINES)


SEE CHAPTER 9 OF TOP-DOWN ASSEMBLY LANGUAGE PROGRAMMING
                     FOR YOUR COMMODORE 64 OR VIC-20.


DUMPING $3900-$3A42
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
3900	05	00	00	00	00	00	00	00	0B	20	08	34	AD	00	39	8D
3910	01	39	A9	FF	8D	54	35	8D	55	35	20	72	34	20	7D	39
3920	CE	01	39	D0	F8	60	20	1A	34	20	08	34	20	E4	34	7F
3930	0D	0A	20	20	20	20	20	50	52	49	4E	54	49	4E	47	20
3940	44	49	53	41	53	53	45	4D	42	4C	45	52	2E	0D	0A	FF
3950	20	E3	35	20	14	34	20	E4	34	7F	0D	0A	44	49	53	41
3960	53	53	45	4D	42	4C	49	4E	47	20	FF	20	D9	36	20	9A
3970	37	20	72	34	20	7D	39	10	FB	20	1A	34	60	20	95	32
3980	48	20	92	39	20	7D	34	68	20	AF	39	20	01	3A	20	7D
3990	37	60	A2	03	8E	02	39	AA	BD	00	3C	AA	BD	50	3B	8E
39A0	03	39	20	40	34	AE	03	39	E8	CE	02	39	D0	EE	60	AA
39B0	BD	00	3D	AA	20	B8	39	60	BD	26	3B	8D	04	39	E8	BD
39C0	26	3B	8D	05	39	6C	04	39	20	0D	33	20	94	35	60	20
39D0	0D	33	20	95	32	48	20	0D	33	20	94	35	68	20	83	34
39E0	60	A9	28	D0	02	A9	29	20	40	34	60	A9	2C	20	40	34
39F0	A9	58	20	40	34	60	A9	2C	20	40	34	A9	59	20	40	34
3A00	60	8D	07	39	8E	06	39	CA	30	06	20	1A	33	CA	10	FA
3A10	08	D8	38	AD	08	39	E9	04	ED	07	39	28	AA	20	96	34
3A20	20	9B	35	20	7D	34	20	94	35	AD	03	30	38	C9	18	90
3A30	03	20	7D	34	20	0D	33	CE	06	39	10	EA	20	1A	33	20
3A40	72	34	60													

----------> END OF APPENDIX D6 <----------

Appendix D7:

Table-Driven Disassembler (Addressing Mode Subroutines)

APPENDIX D7: DISASSEMBLER
(ADDRESSING MODE SUBROUTINES)

SEE CHAPTER 9 OF TOP-DOWN ASSEMBLY LANGUAGE PROGRAMMING
FOR YOUR COMMODORE 64 OR VIC-20.

DUMPING $3A50-$3B43

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
3A50	20	CF	39	A2	02	A9	04	60	20	50	3A	20	EB	39	A2	02
3A60	A9	06	60	20	50	3A	20	F6	39	A2	02	A9	06	60	A9	41
3A70	20	40	34	A2	00	A9	01	60	A2	00	A9	00	60	A9	23	20
3A80	40	34	A9	24	20	40	34	20	C8	39	A2	01	A9	04	60	20
3A90	E1	39	20	50	3A	20	E5	39	A9	06	A2	02	60	20	E1	39
3AA0	20	F3	3A	20	E5	39	A2	01	A9	06	60	20	E1	39	20	EB
3AB0	3A	20	E5	39	20	F6	39	A2	01	A9	06	60	20	0D	33	20
3AC0	12	35	20	95	32	48	20	0D	33	68	C9	00	10	03	CE	06
3AD0	32	08	D8	18	6D	05	32	90	03	EE	06	32	8D	05	32	28
3AE0	20	9B	35	20	2B	35	A2	01	A9	04	60	20	C8	39	A2	01
3AF0	A9	02	60	20	EB	3A	20	EB	39	A2	01	A9	04	60	20	EB
3B00	3A	20	F6	39	A2	01	A9	04	60	68	68	68	68	20	7D	37
3B10	30	0D	20	95	32	C9	FF	F0	06	20	40	34	18	90	EE	20
3B20	72	34	20	7D	37	60	78	3A	6E	3A	7D	3A	EB	3A	F3	3A
3B30	FE	3A	50	3A	58	3A	63	3A	78	3A	BC	3A	9D	3A	AB	3A
3B40	8F	3A	09	3B												

----------> END OF APPENDIX D7 <----------

Appendix D8:

Table-Driven Disassembler (Tables)

```
APPENDIX D8:            DISASSEMBLER (TABLES)


SEE CHAPTER 9 OF TOP-DOWN ASSEMBLY LANGUAGE PROGRAMMING
                FOR YOUR COMMODORE 64 OR VIC-20.


DUMPING $3B50-$3DFF
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
3B50	7F	42	41	44	41	44	43	41	4E	44	41	53	4C	42	43	43
3B60	42	43	53	42	45	51	42	49	54	42	4D	49	42	4E	45	42
3B70	50	4C	42	52	4B	42	56	43	42	56	53	43	4C	43	43	4C
3B80	44	43	4C	49	43	4C	56	43	4D	50	43	50	58	43	50	59
3B90	44	45	43	44	45	58	44	45	59	45	4F	52	49	4E	43	49
3BA0	4E	58	49	4E	59	4A	4D	50	4A	53	52	4C	44	41	4C	44
3BB0	58	4C	44	59	4C	53	52	4E	4F	50	4F	52	41	50	48	41
3BC0	50	48	50	50	4C	41	50	4C	50	52	4F	4C	52	4F	52	52
3BD0	54	49	52	54	53	53	42	43	53	45	43	53	45	44	53	45
3BE0	49	53	54	41	53	54	58	53	54	59	54	41	58	54	41	59
3BF0	54	53	58	54	58	41	54	58	53	54	59	41	54	45	58	FF
3C00	22	6A	01	01	01	6A	0A	01	70	6A	0A	01	01	6A	0A	01
3C10	1F	6A	01	01	01	6A	0A	01	2B	6A	01	01	01	6A	0A	01
3C20	58	07	01	01	16	07	79	01	76	07	79	01	16	07	79	01
3C30	19	07	01	01	01	07	79	01	88	07	01	01	01	07	79	01
3C40	7F	49	01	01	01	49	64	01	6D	49	64	01	55	49	64	01
3C50	25	49	01	01	01	49	64	01	31	49	01	01	01	49	64	01
3C60	82	04	01	01	01	04	7C	01	73	04	7C	01	55	04	7C	01
3C70	28	04	01	01	01	04	7C	01	8E	04	01	01	01	04	7C	AC
3C80	01	91	01	01	97	91	94	01	46	01	A3	01	97	91	94	01
3C90	0D	91	01	01	97	91	94	01	A9	91	A3	01	01	91	01	01
3CA0	61	5B	5E	01	61	5B	5E	01	9D	5B	9A	01	61	5B	5E	01
3CB0	10	5B	01	01	61	58	5E	01	34	5B	9E	01	61	5B	5E	01
3CC0	3D	37	01	01	3D	37	40	01	52	37	43	01	3D	37	40	01
3CD0	1C	37	01	01	01	37	40	01	2E	37	01	01	01	37	40	01

```
3CE0    3A 85 01 01 3A 85 4C 01 4F 85 67 01 3A 85 4C 01
3CF0    13 85 01 01 01 85 4C 01 8B 85 01 01 01 85 4C 01
3D00    12 16 00 00 00 06 06 00 12 04 02 00 00 0C 0C 00
3D10    14 18 00 00 00 0E 0E 00 12 10 00 00 00 16 16 00
3D20    0C 16 00 00 06 06 06 00 12 04 02 00 0C 0C 0C 00
3D30    14 18 00 00 00 08 08 00 12 10 00 00 00 0E 0E 00
3D40    12 16 00 00 00 06 06 00 12 0C 02 00 0C 0C 0C 00
3D50    14 18 00 00 00 08 08 00 12 10 00 00 00 0E 0E 00
3D60    12 16 00 00 00 06 06 00 12 04 02 00 1A 0C 0C 00
3D70    14 18 00 00 00 08 08 00 12 10 00 00 00 0E 0E 1C
3D80    00 16 00 00 06 06 06 00 12 00 12 00 0C 0C 0C 00
3D90    14 18 00 00 08 08 0A 00 12 10 12 00 00 0E 00 00
3DA0    04 16 04 00 06 06 06 00 12 04 12 00 0C 0C 0C 00
3DB0    14 18 00 00 08 08 0A 00 14 10 12 00 0E 0E 10 00
3DC0    04 16 00 00 06 06 06 00 12 04 12 00 0C 0C 0C 00
3DD0    14 18 00 00 00 08 08 00 12 10 00 00 00 0E 0E 00
3DE0    04 16 00 00 06 06 06 00 12 04 12 00 0C 0C 0C 00
3DF0    14 18 00 00 00 08 08 00 12 10 00 00 00 0E 0E 00

----------->  END OF APPENDIX D8  <----------
```

Appendix D9:

Move Utilities

APPENDIX D9: MOVE UTILITIES

SEE CHAPTER 10 OF TOP-DOWN ASSEMBLY LANGUAGE PROGRAMMING
FOR YOUR COMMODORE 64 OR VIC-20.

DUMPING $37B0-$38EE

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
37B0	00	00	00	00	20	08	34	20	E4	34	7F	0D	0A	20	20	20
37C0	20	20	4D	4F	56	45	20	54	4F	4F	4C	2E	0D	0A	0A	FF
37D0	20	E3	35	20	B9	38	AE	55	35	38	AD	54	35	ED	52	35
37E0	8D	B0	37	B0	02	CA	38	8A	ED	53	35	8D	B1	37	B0	03
37F0	A9	00	60	A0	03	B9	FB	00	48	88	10	F9	38	AD	53	35
3800	CD	B3	37	90	40	D0	18	AD	52	35	CD	B2	37	90	36	D0
3810	0E	A0	00	68	99	FB	00	C8	C0	04	D0	F7	A9	FF	60	20
3820	A4	38	A0	00	AE	B1	37	F0	0E	B1	FB	91	FD	C8	D0	F9
3830	E6	FC	E6	FE	CA	D0	F2	88	C8	B1	FB	91	FD	CC	B0	37
3840	D0	F6	4C	11	38	AD	B1	37	F0	48	AC	B1	37	AD	B0	37
3850	38	E9	FF	B0	01	88	AA	84	FE	8A	18	6D	52	35	85	FB
3860	90	01	C8	98	6D	53	35	85	FC	8A	18	6D	B2	37	85	FD
3870	90	02	E6	FE	A5	FE	6D	B3	37	85	FE	AE	B1	37	A0	FF
3880	B1	FB	91	FD	88	D0	F9	B1	FB	91	FD	C6	FC	C6	FE	CA
3890	D0	EC	20	A4	38	AC	B0	37	B1	FB	91	FD	88	C0	FF	D0
38A0	F7	4C	11	38	AD	52	35	85	FB	AD	53	35	85	FC	AD	B2
38B0	37	85	FD	AD	B3	37	85	FE	60	20	08	34	20	E4	34	7F
38C0	0D	0A	53	45	54	20	44	45	53	54	49	4E	41	54	49	4F
38D0	4E	20	41	4E	44	20	50	52	45	53	53	20	51	2E	FF	20
38E0	07	32	AD	05	32	8D	B2	37	AD	06	32	8D	B3	37	60	

----------> END OF APPENDIX D9 <----------

Appendix D10:

Simple Text Editor

APPENDIX D10:   A SIMPLE TEXT EDITOR

SEE CHAPTER 11 OF TOP-DOWN ASSEMBLY LANGUAGE PROGRAMMING
FOR YOUR COMMODORE 64 OR VIC-20.

DUMPING $3E00-$3FFF

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
3E00	00	00	20	0F	3E	20	37	3E	20	C8	3E	18	18	90	F6	20
3E10	08	34	20	E4	34	7F	0D	0A	0A	53	45	54	20	55	50	20
3E20	45	44	49	54	20	42	55	46	46	45	52	2E	0D	0A	0A	FF
3E30	20	E3	35	20	9A	37	60	20	C4	31	20	2B	31	AE	03	30
3E40	A0	03	20	13	31	20	2B	31	20	76	31	20	C4	31	20	5E
3E50	3E	20	D3	31	20	76	31	20	88	3E	20	D3	31	60	20	12
3E60	35	AD	03	30	4A	AA	CA	20	1A	33	CA	10	FA	AD	03	30
3E70	8D	00	3E	20	95	32	20	9B	31	20	7F	31	20	0D	33	CE
3E80	00	3E	10	EF	20	2B	35	60	AD	03	30	4A	E9	02	20	81
3E90	31	AD	01	3E	C9	01	D0	05	A9	49	18	90	02	A9	4F	20
3EA0	9B	31	A9	02	20	81	31	AD	07	30	20	9B	31	A9	02	20
3EB0	81	31	AD	06	32	20	A3	31	AD	05	32	20	A3	31	60	00
3EC0	93	94	1D	9D	10	14	51	00	20	E0	32	CD	C6	3E	D0	17
3ED0	48	20	E0	32	CD	C6	3E	D0	04	68	68	68	60	8D	C7	3E
3EE0	68	20	E7	3E	AD	C7	3E	CD	C1	3E	D0	0B	CE	01	3E	10
3EF0	05	A9	01	8D	01	3E	60	CD	C2	3E	D0	04	20	79	3F	60
3F00	CD	C3	3E	D0	04	20	87	3F	60	CD	C5	3E	D0	04	20	DD
3F10	3F	60	CD	C4	3E	D0	04	20	C5	3F	60	CD	C0	3E	D0	04
3F20	20	B4	3F	60	AE	01	3E	F0	04	20	34	3F	60	20	2D	33
3F30	20	7D	37	60	48	20	12	35	AD	53	35	48	AD	52	35	48
3F40	AD	55	35	48	AD	54	35	48	20	61	36	20	7D	37	30	11
3F50	20	E2	38	AD	54	35	D0	03	CE	55	35	CE	54	35	20	D6
3F60	37	68	8D	54	35	68	8D	55	35	68	8D	52	35	68	8D	53
3F70	35	20	2B	35	68	20	2D	3F	60	20	95	32	C9	FF	F0	04
3F80	20	7D	37	60	A9	FF	60	38	AD	53	35	CD	06	32	90	0C

```
3F90   D0 10 AD 52 35 CD 05 32 F0 17 B0 06 20 1A 33 A9
3FA0   00 60 AD 52 35 8D 05 32 AD 53 35 8D 06 32 A9 00
3FB0   60 A9 FF 60 20 9A 37 A9 FF 20 2D 33 20 7D 37 10
3FC0   F6 20 9A 37 60 20 9A 37 20 14 34 20 95 32 C9 FF
3FD0   F0 08 20 40 34 20 7D 37 10 F1 4C 1A 34 20 12 35
3FE0   AD 53 35 48 AD 52 35 48 20 E2 38 20 7D 37 20 61
3FF0   36 20 D6 37 68 8D 52 35 68 8D 53 35 20 2B 35 60
```

----------> END OF APPENDIX D10 <----------

Appendix D11:

Extending the Visible Monitor

APPENDIX D11:   EXTENDING THE VISIBLE MONITOR

SEE CHAPTER 12 OF TOP-DOWN ASSEMBLY LANGUAGE PROGRAMMING
FOR YOUR COMMODORE 64 OR VIC-20.

DUMPING $30B0-$30FC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
30B0	C9	50	D0	09	AD	00	34	49	FF	8D	00	34	60	C9	55	D0
30C0	09	AD	02	34	49	FF	8D	02	34	60	C9	48	D0	0D	AD	00
30D0	34	D0	04	20	57	35	60	20	A8	35	60	C9	4D	D0	04	20
30E0	B4	37	60	C9	3F	D0	0D	AD	00	34	D0	04	20	09	39	60
30F0	20	26	39	60	C9	54	D0	04	20	02	3E	60	60			

-----------> END OF APPENDIX D11 <----------

Appendix D12:

System Data Block for the VIC-20

APPENDIX D12: SYSTEM DATA BLOCK FOR THE VIC-20

SEE APPENDIX B1 OF TOP-DOWN ASSEMBLY LANGUAGE PROGRAMMING
FOR YOUR COMMODORE 64 OR VIC-20.

DUMPING $3000-$30AF

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
3000	00	10	16	15	18	11	20	1E	35	30	3C	30	41	30	10	30
3010	60	48	A5	FC	48	18	69	84	85	FC	A0	00	AD	86	02	91
3020	FB	68	85	FC	68	38	C9	40	90	0A	C9	60	90	03	E9	20
3030	60	38	E9	40	60	20	E4	FF	AA	F0	FA	60	A2	01	4C	43
3040	30	A2	02	48	20	C9	FF	68	20	D2	FF	60	01	00	00	00
3050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
3060	00	00	A9	03	AE	4C	30	A8	20	BA	FF	AD	4D	30	A2	4E
3070	A0	30	20	BD	FF	AD	52	35	85	FD	AD	53	35	85	FE	A9
3080	FD	AE	54	35	AC	55	35	E8	D0	01	C8	20	D8	FF	60	A9
3090	00	48	28	20	BD	FF	A9	01	A2	00	A0	FF	20	BA	FF	20
30A0	C0	FF	A9	03	8D	51	35	20	07	32	A9	01	20	C3	FF	60

-----------> END OF APPENDIX D12 <----------

Appendix D13:

System Data Block for the Commodore 64

```
APPENDIX D13:    SYSTEM DATA BLOCK FOR THE COMMODORE 64


SEE CHAPTER B2 OF TOP-DOWN ASSEMBLY LANGUAGE PROGRAMMING
                   FOR YOUR COMMODORE 64 OR VIC-20.


DUMPING $3000-$30AA

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

3000   00 04 28 27 18 07 20 1E 35 30 3C 30 41 30 10 30
3010   60 48 A5 FC 48 18 69 D4 85 FC A0 00 AD 86 02 91
3020   FB 68 85 FC 68 38 C9 40 90 0A C9 60 90 03 E9 20
3030   60 38 E9 40 60 20 E4 FF AA F0 FA 60 A2 01 4C 43
3040   30 A2 02 48 20 C9 FF 68 20 D2 FF 60 01 00 00 00
3050   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
3060   00 00 A9 03 AE 4C 30 A8 20 BA FF AD 4D 30 A2 4E
3070   A0 30 20 BD FF AD 52 35 85 FD AD 53 35 85 FE A9
3080   FD AE 54 35 AC 55 35 E8 D0 01 C8 20 D8 FF 60 A9
3090   00 48 28 20 BD FF A9 01 A2 00 A0 FF 20 BA FF 20
30A0   C0 FF 20 07 32 A9 01 20 C3 FF 60

------------>  END OF APPENDIX D13  <----------
```

Appendix E1:

Screen Utilities

```
APPENDIX E1:          SCREEN UTILITIES


THE FOLLOWING DATA STATEMENTS CONTAIN
DECIMAL OBJECT CODE AND CHECKSUMS
FOR MEMORY FROM 12544 TO 12769
SUITABLE FOR LOADING WITH THE
BASIC OBJECT CODE LOADER.


1000  DATA  12544, 32, 196, 49, 32, 43, 49, 174, 3, 13122
1001  DATA  12552, 48, 172, 4, 48, 32, 19, 49, 32, 12956
1002  DATA  12560, 211, 49, 96, 142, 42; 49, 152, 170, 13471
1003  DATA  12568, 173, 6, 48, 172, 42, 49, 145, 251, 13454
1004  DATA  12576, 136, 16, 251, 32, 118, 49, 202, 16, 13396
1005  DATA  12584, 239, 96, 39, 162, 0, 160, 0, 24, 13304
1006  DATA  12592, 144, 10, 173, 4, 48, 74, 168, 173, 13386
1007  DATA  12600, 3, 48, 74, 170, 56, 236, 3, 48, 13238
1008  DATA  12608, 144, 3, 174, 3, 48, 56, 204, 4, 13244
1009  DATA  12616, 48, 144, 3, 172, 4, 48, 173, 0, 13208
1010  DATA  12624, 48, 133, 251, 173, 1, 48, 133, 252, 13663
1011  DATA  12632, 8, 216, 138, 24, 101, 251, 144, 3, 13517
1012  DATA  12640, 230, 252, 24, 192, 0, 240, 11, 24, 13613
1013  DATA  12648, 109, 2, 48, 144, 2, 230, 252, 136, 13571
1014  DATA  12656, 208, 245, 133, 251, 40, 96, 173, 2, 13804
1015  DATA  12664, 48, 24, 144, 5, 32, 155, 49, 169, 13290
1016  DATA  12672, 1, 8, 216, 24, 101, 251, 144, 2, 13419
1017  DATA  12680, 230, 252, 133, 251, 56, 173, 5, 48, 13828
1018  DATA  12688, 197, 252, 176, 5, 173, 1, 48, 133, 13673
1019  DATA  12696, 252, 40, 96, 32, 17, 48, 160, 0, 13341
1020  DATA  12704, 145, 251, 96, 72, 74, 74, 74, 74, 13564
1021  DATA  12712, 32, 182, 49, 32, 124, 49, 104, 32, 13316
1022  DATA  12720, 182, 49, 32, 124, 49, 96, 8, 216, 13476
1023  DATA  12728, 41, 15, 201, 10, 48, 2, 105, 6, 13156
1024  DATA  12736, 105, 48, 40, 96, 104, 170, 104, 168, 13571
1025  DATA  12744, 165, 252, 72, 165, 251, 72, 152, 72, 13945
1026  DATA  12752, 138, 72, 96, 104, 170, 104, 168, 104, 13708
1027  DATA  12760, 133, 251, 104, 133, 252, 152, 72, 138, 13995
1028  DATA  12768, 72, 96, 0, 255, 0, 255, 0, 255, 13701

---------->  END OF APPENDIX E1  <----------
```

Appendix E2:

Visible Monitor (Top Level and Display Subroutines)

```
APPENDIX E2:          THE VISIBLE MONITOR
               (TOP LEVEL & DISPLAY SUBROUTINES)


THE FOLLOWING DATA STATEMENTS CONTAIN
DECIMAL OBJECT CODE AND CHECKSUMS
FOR MEMORY FROM 12800 TO 13011
SUITABLE FOR LOADING WITH THE
BASIC OBJECT CODE LOADER.


1100  DATA  12800, 0, 0, 0, 0, 0, 0, 0, 8, 12808
1101  DATA  12808, 216, 32, 18, 50, 32, 227, 50, 24, 13457
1102  DATA  12816, 144, 246, 32, 196, 49, 32, 37, 50, 13602
1103  DATA  12824, 32, 53, 50, 32, 93, 50, 32, 176, 13342
1104  DATA  12832, 50, 32, 211, 49, 96, 162, 0, 160, 13592
1105  DATA  12840, 0, 32, 60, 49, 174, 3, 48, 160, 13366
1106  DATA  12848, 3, 32, 19, 49, 96, 162, 11, 160, 13380
1107  DATA  12856, 0, 32, 60, 49, 160, 0, 140, 82, 13379
1108  DATA  12864, 50, 185, 83, 50, 32, 124, 49, 238, 13675
1109  DATA  12872, 82, 50, 172, 82, 50, 192, 10, 208, 13718
1110  DATA  12880, 240, 96, 10, 65, 32, 32, 88, 32, 13475
1111  DATA  12888, 32, 89, 32, 32, 80, 162, 0, 160, 13475
1112  DATA  12896, 1, 32, 60, 49, 173, 6, 50, 32, 13299
1113  DATA  12904, 163, 49, 173, 5, 50, 32, 163, 49, 13588
1114  DATA  12912, 32, 127, 49, 32, 149, 50, 72, 32, 13455
1115  DATA  12920, 163, 49, 32, 127, 49, 104, 32, 124, 13600
1116  DATA  12928, 49, 32, 127, 49, 162, 0, 189, 1, 13537
1117  DATA  12936, 50, 32, 163, 49, 32, 127, 49, 232, 13670
1118  DATA  12944, 224, 4, 208, 242, 96, 165, 251, 72, 14206
1119  DATA  12952, 166, 252, 173, 5, 50, 133, 251, 173, 14155
1120  DATA  12960, 6, 50, 133, 252, 160, 0, 177, 251, 13989
1121  DATA  12968, 168, 104, 133, 251, 134, 252, 152, 96, 14258
1122  DATA  12976, 172, 0, 50, 56, 192, 7, 144, 5, 13602
1123  DATA  12984, 160, 0, 140, 0, 50, 185, 205, 50, 13774
1124  DATA  12992, 170, 160, 2, 32, 60, 49, 173, 7, 13645
1125  DATA  13000, 48, 32, 124, 49, 96, 3, 6, 8, 13366
1126  DATA  13008, 11, 14, 17, 20, 0, 255, 0, 255, 13580

---------->   END OF APPENDIX E2  <----------
```

Appendix E3:

Visible Monitor (Update Subroutine)

```
APPENDIX E3:          THE VISIBLE MONITOR (UPDATE SUBROUTINE)


THE FOLLOWING DATA STATEMENTS CONTAIN
DECIMAL OBJECT CODE AND CHECKSUMS
FOR MEMORY FROM 13024 TO 13294
SUITABLE FOR LOADING WITH THE
BASIC OBJECT CODE LOADER.


1200  DATA  13024, 108, 8, 48, 32, 224, 50, 201, 29, 13724
1201  DATA  13032, 208, 16, 238, 0, 50, 173, 0, 50, 13767
1202  DATA  13040, 201, 7, 208, 5, 169, 0, 141, 0, 13771
1203  DATA  13048, 50, 96, 201, 157, 208, 11, 206, 0, 13977
1204  DATA  13056, 50, 16, 5, 169, 6, 141, 0, 50, 13493
1205  DATA  13064, 96, 201, 32, 208, 9, 238, 5, 50, 13903
1206  DATA  13072, 208, 3, 238, 6, 50, 96, 201, 13, 13887
1207  DATA  13080, 208, 12, 173, 5, 50, 208, 3, 206, 13945
1208  DATA  13088, 6, 50, 206, 5, 50, 96, 174, 0, 13675
1209  DATA  13096, 50, 224, 2, 208, 27, 168, 165, 251, 14191
1210  DATA  13104, 72, 166, 252, 173, 5, 50, 133, 251, 14206
1211  DATA  13112, 173, 6, 50, 133, 252, 152, 160, 0, 14038
1212  DATA  13120, 145, 251, 134, 252, 104, 133, 251, 96, 14486
1213  DATA  13128, 201, 71, 208, 35, 172, 3, 50, 174, 14042
1214  DATA  13136, 2, 50, 173, 4, 50, 72, 173, 1, 13661
1215  DATA  13144, 50, 40, 32, 108, 51, 8, 141, 1, 13575
1216  DATA  13152, 50, 142, 2, 50, 140, 3, 50, 104, 13693
1217  DATA  13160, 141, 4, 50, 96, 108, 5, 50, 72, 13686
1218  DATA  13168, 32, 213, 51, 48, 75, 168, 104, 152, 14011
1219  DATA  13176, 174, 0, 50, 208, 20, 162, 3, 24, 13817
1220  DATA  13184, 14, 5, 50, 46, 6, 50, 202, 16, 13573
1221  DATA  13192, 246, 152, 13, 5, 50, 141, 5, 50, 13854
1222  DATA  13200, 96, 224, 1, 208, 24, 41, 15, 72, 13881
1223  DATA  13208, 32, 149, 50, 10, 10, 10, 10, 41, 13520
1224  DATA  13216, 240, 141, 172, 51, 104, 13, 172, 51, 14160
1225  DATA  13224, 32, 45, 51, 96, 0, 202, 202, 202, 14054
1226  DATA  13232, 160, 3, 24, 30, 1, 50, 136, 16, 13652
1227  DATA  13240, 249, 29, 1, 50, 157, 1, 50, 96, 13873
```

```
1228  DATA  13248, 104, 201, 147, 208, 4, 32, 0, 49, 13993
1229  DATA  13256, 96, 201, 81, 208, 4, 104, 104, 40, 14094
1230  DATA  13264, 96, 32, 16, 48, 96, 56, 233, 48, 13889
1231  DATA  13272, 144, 15, 201, 10, 144, 14, 233, 7, 14040
1232  DATA  13280, 201, 16, 176, 5, 56, 201, 10, 176, 14121
1233  DATA  13288, 3, 169, 255, 96, 162, 0, 96, 255, 14324
```

----------> END OF APPENDIX E3 <----------

Appendix E4:

Print Utilities

```
APPENDIX E4:          PRINT UTILITIES


THE FOLLOWING DATA STATEMENTS CONTAIN
DECIMAL OBJECT CODE AND CHECKSUMS
FOR MEMORY FROM 13312 TO 13635
SUITABLE FOR LOADING WITH THE
BASIC OBJECT CODE LOADER.


1300  DATA  13312, 0, 255, 0, 0, 0, 0, 0, 0, 13567
1301  DATA  13320, 169, 255, 141, 1, 52, 96, 169, 0, 14203
1302  DATA  13328, 141, 1, 52, 96, 169, 255, 141, 0, 14183
1303  DATA  13336, 52, 96, 169, 0, 141, 0, 52, 96, 13942
1304  DATA  13344, 169, 255, 141, 2, 52, 96, 169, 0, 14228
1305  DATA  13352, 141, 2, 52, 96, 32, 8, 52, 32, 13767
1306  DATA  13360, 20, 52, 32, 32, 52, 96, 32, 14, 13690
1307  DATA  13368, 52, 32, 26, 52, 32, 38, 52, 96, 13748
1308  DATA  13376, 201, 0, 240, 36, 141, 3, 52, 173, 14222
1309  DATA  13384, 1, 52, 240, 6, 173, 3, 52, 32, 13943
1310  DATA  13392, 105, 52, 173, 0, 52, 240, 6, 173, 14193
1311  DATA  13400, 3, 52, 32, 108, 52, 173, 2, 52, 13874
1312  DATA  13408, 240, 6, 173, 3, 52, 32, 111, 52, 14077
1313  DATA  13416, 96, 108, 10, 48, 108, 12, 48, 108, 13954
1314  DATA  13424, 14, 48, 169, 13, 32, 64, 52, 169, 13985
1315  DATA  13432, 10, 32, 64, 52, 96, 169, 32, 32, 13919
1316  DATA  13440, 64, 52, 96, 72, 74, 74, 74, 74, 14020
1317  DATA  13448, 32, 182, 49, 32, 64, 52, 104, 32, 13995
1318  DATA  13456, 182, 49, 32, 64, 52, 96, 169, 32, 14132
1319  DATA  13464, 142, 4, 52, 72, 174, 4, 52, 240, 14204
1320  DATA  13472, 10, 206, 4, 52, 32, 64, 52, 104, 13996
1321  DATA  13480, 24, 144, 240, 104, 96, 142, 4, 52, 14286
1322  DATA  13488, 174, 4, 52, 240, 9, 206, 4, 52, 14229
1323  DATA  13496, 32, 114, 52, 24, 144, 242, 96, 142, 14342
1324  DATA  13504, 5, 52, 181, 1, 72, 181, 0, 72, 14068
1325  DATA  13512, 174, 5, 52, 161, 0, 201, 255, 240, 14600
1326  DATA  13520, 12, 246, 0, 208, 2, 246, 1, 32, 14267
1327  DATA  13528, 64, 52, 24, 144, 235, 104, 149, 0, 14300
```

```
1328  DATA  13536, 104, 149, 1, 96, 104, 170, 104, 168, 14432
1329  DATA  13544, 32, 18, 53, 142, 5, 50, 140, 6, 13990
1330  DATA  13552, 50, 32, 13, 51, 32, 13, 51, 32, 13826
1331  DATA  13560, 149, 50, 201, 255, 240, 6, 32, 64, 14557
1332  DATA  13568, 52, 24, 144, 240, 174, 5, 50, 172, 14429
1333  DATA  13576, 6, 50, 32, 43, 53, 152, 72, 138, 14122
1334  DATA  13584, 72, 96, 104, 141, 6, 52, 104, 141, 14300
1335  DATA  13592, 7, 52, 173, 6, 50, 72, 173, 5, 14130
1336  DATA  13600, 50, 72, 173, 7, 52, 72, 173, 6, 14205
1337  DATA  13608, 52, 72, 96, 104, 141, 6, 52, 104, 14235
1338  DATA  13616, 141, 7, 52, 104, 141, 5, 50, 104, 14220
1339  DATA  13624, 141, 6, 50, 173, 7, 52, 72, 173, 14298
1340  DATA  13632, 6, 52, 72, 96, 0, 255, 0, 255, 14368

----------> END OF APPENDIX E4 <----------
```

Appendix E5:

Two Hexdump Tools

```
APPENDIX E5:          TWO HEXDUMP TOOLS


THE FOLLOWING DATA STATEMENTS CONTAIN
DECIMAL OBJECT CODE AND CHECKSUMS
FOR MEMORY FROM 13648 TO 14246
SUITABLE FOR LOADING WITH THE
BASIC OBJECT CODE LOADER.

1400  DATA  13648, 0, 7, 0, 0, 255, 255, 0, 32, 14197
1401  DATA  13656, 8, 52, 169, 4, 141, 80, 53, 173, 14336
1402  DATA  13664, 5, 50, 41, 240, 141, 5, 50, 32, 14228
1403  DATA  13672, 155, 53, 32, 125, 52, 32, 125, 52, 14298
1404  DATA  13680, 32, 148, 53, 32, 13, 51, 173, 5, 14187
1405  DATA  13688, 50, 45, 81, 53, 208, 239, 32, 114, 14510
1406  DATA  13696, 52, 173, 5, 50, 41, 15, 208, 3, 14243
1407  DATA  13704, 32, 114, 52, 206, 80, 53, 208, 215, 14664
1408  DATA  13712, 32, 14, 52, 96, 32, 149, 50, 32, 14169
1409  DATA  13720, 131, 52, 96, 173, 6, 50, 32, 131, 14391
1410  DATA  13728, 52, 173, 5, 50, 32, 131, 52, 96, 14319
1411  DATA  13736, 32, 195, 53, 32, 227, 53, 32, 154, 14514
1412  DATA  13744, 55, 32, 20, 52, 32, 229, 54, 32, 14250
1413  DATA  13752, 60, 55, 16, 251, 32, 114, 52, 32, 14364
1414  DATA  13760, 26, 52, 96, 32, 8, 52, 32, 26, 14084
1415  DATA  13768, 52, 32, 228, 52, 127, 13, 80, 82, 14434
1416  DATA  13776, 73, 78, 84, 73, 78, 71, 32, 72, 14337
1417  DATA  13784, 69, 88, 68, 85, 77, 80, 13, 10, 14274
1418  DATA  13792, 10, 255, 96, 32, 8, 52, 32, 228, 14505
1419  DATA  13800, 52, 127, 13, 10, 83, 69, 84, 32, 14270
1420  DATA  13808, 83, 84, 65, 82, 84, 73, 78, 71, 14428
1421  DATA  13816, 32, 65, 68, 68, 82, 69, 83, 83, 14366
1422  DATA  13824, 32, 65, 78, 68, 32, 80, 82, 69, 14330
1423  DATA  13832, 83, 83, 32, 34, 81, 34, 46, 255, 14480
1424  DATA  13840, 32, 7, 50, 32, 97, 54, 32, 8, 14152
1425  DATA  13848, 52, 32, 228, 52, 127, 13, 10, 83, 14445
1426  DATA  13856, 69, 84, 32, 69, 78, 68, 32, 65, 14353
1427  DATA  13864, 68, 68, 82, 69, 83, 83, 32, 65, 14414
```

```
1428  DATA  13872, 78, 68, 32, 80, 82, 69, 83, 83, 14447
1429  DATA  13880, 32, 34, 81, 34, 46, 255, 32, 7, 14401
1430  DATA  13888, 50, 56, 173, 6, 50, 205, 83, 53, 14564
1431  DATA  13896, 144, 36, 208, 8, 173, 5, 50, 205, 14725
1432  DATA  13904, 82, 53, 144, 26, 173, 6, 50, 141, 14579
1433  DATA  13912, 85, 53, 173, 5, 50, 141, 84, 53, 14556
1434  DATA  13920, 96, 173, 6, 50, 141, 83, 53, 173, 14695
1435  DATA  13928, 5, 50, 141, 82, 53, 96, 32, 228, 14615
1436  DATA  13936, 52, 127, 13, 10, 10, 10, 32, 69, 14259
1437  DATA  13944, 82, 82, 79, 82, 33, 33, 33, 32, 14400
1438  DATA  13952, 69, 78, 68, 32, 65, 68, 68, 82, 14482
1439  DATA  13960, 69, 83, 83, 32, 76, 69, 83, 83, 14538
1440  DATA  13968, 32, 84, 72, 65, 78, 32, 83, 84, 14498
1441  DATA  13976, 65, 82, 84, 32, 65, 68, 68, 82, 14522
1442  DATA  13984, 69, 83, 83, 44, 32, 87, 72, 73, 14527
1443  DATA  13992, 67, 72, 32, 73, 83, 32, 255, 32, 14638
1444  DATA  14000, 181, 54, 76, 22, 54, 169, 36, 32, 14624
1445  DATA  14008, 64, 52, 173, 83, 53, 32, 131, 52, 14648
1446  DATA  14016, 173, 82, 53, 32, 131, 52, 96, 169, 14804
1447  DATA  14024, 36, 32, 64, 52, 173, 85, 53, 32, 14551
1448  DATA  14032, 131, 52, 173, 84, 53, 32, 131, 52, 14740
1449  DATA  14040, 96, 32, 181, 54, 169, 45, 32, 64, 14713
1450  DATA  14048, 52, 32, 199, 54, 96, 32, 228, 52, 14793
1451  DATA  14056, 127, 13, 10, 10, 68, 85, 77, 80, 14526
1452  DATA  14064, 73, 78, 71, 32, 255, 32, 217, 54, 14876
1453  DATA  14072, 32, 114, 52, 32, 228, 52, 127, 10, 14719
1454  DATA  14080, 10, 32, 32, 32, 32, 32, 32, 32, 14314
1455  DATA  14088, 32, 48, 32, 32, 49, 32, 32, 50, 14395
1456  DATA  14096, 32, 32, 51, 32, 32, 52, 32, 32, 14391
1457  DATA  14104, 53, 32, 32, 54, 32, 32, 55, 32, 14426
1458  DATA  14112, 32, 56, 32, 32, 57, 32, 32, 65, 14450
1459  DATA  14120, 32, 32, 66, 32, 32, 67, 32, 32, 14445
1460  DATA  14128, 68, 32, 32, 69, 32, 32, 70, 13, 14476
1461  DATA  14136, 10, 10, 255, 96, 32, 114, 52, 173, 14878
1462  DATA  14144, 5, 50, 72, 41, 15, 141, 86, 53, 14607
1463  DATA  14152, 104, 41, 240, 141, 5, 50, 32, 155, 14920
1464  DATA  14160, 53, 162, 3, 32, 150, 52, 173, 86, 14871
1465  DATA  14168, 53, 240, 13, 162, 3, 32, 150, 52, 14873
1466  DATA  14176, 32, 13, 51, 206, 86, 53, 208, 243, 15068
1467  DATA  14184, 32, 148, 53, 32, 125, 52, 32, 125, 14783
1468  DATA  14192, 55, 48, 9, 173, 5, 50, 41, 15, 14588
1469  DATA  14200, 201, 0, 208, 236, 96, 56, 173, 6, 15176
1470  DATA  14208, 50, 205, 85, 53, 144, 11, 208, 15, 14979
1471  DATA  14216, 56, 173, 5, 50, 205, 84, 53, 176, 15018
1472  DATA  14224, 6, 32, 13, 51, 169, 0, 96, 169, 14760
1473  DATA  14232, 255, 96, 173, 82, 53, 141, 5, 50, 15087
1474  DATA  14240, 173, 83, 53, 141, 6. 50, 96, 255, 15097

---------->  END OF APPENDIX E5  <----------
```

Appendix E6:

Table-Driven Disassembler (Top Level and Utility Subroutines)

```
APPENDIX E6:           DISASSEMBLER (TOP LEVEL & UTILITY SUBS)



THE FOLLOWING DATA STATEMENTS CONTAIN
DECIMAL OBJECT CODE AND CHECKSUMS
FOR MEMORY FROM 14592 TO 14914
SUITABLE FOR LOADING WITH THE
BASIC OBJECT CODE LOADER.


1500  DATA  14592, 5, 0, 0, 0, 0, 0, 0, 0, 14597
1501  DATA  14600, 11, 32, 8, 52, 173, 0, 57, 141, 15074
1502  DATA  14608, 1, 57, 169, 255, 141, 84, 53, 141, 15509
1503  DATA  14616, 85, 53, 32, 114, 52, 32, 125, 57, 15166
1504  DATA  14624, 206, 1, 57, 208, 248, 96, 32, 26, 15498
1505  DATA  14632, 52, 32, 8, 52, 32, 228, 52, 127, 15215
1506  DATA  14640, 13, 10, 32, 32, 32, 32, 32, 80, 14903
1507  DATA  14648, 82, 73, 78, 84, 73, 78, 71, 32, 15219
1508  DATA  14656, 68, 73, 83, 65, 83, 83, 69, 77, 15257
1509  DATA  14664, 66, 76, 69, 82, 46, 13, 10, 255, 15281
1510  DATA  14672, 32, 227, 53, 32, 20, 52, 32, 228, 15348
1511  DATA  14680, 52, 127, 13, 10, 68, 73, 83, 65, 15171
1512  DATA  14688, 83, 83, 69, 77, 66, 76, 73, 78, 15293
1513  DATA  14696, 71, 32, 255, 32, 217, 54, 32, 154, 15543
1514  DATA  14704, 55, 32, 114, 52, 32, 125, 57, 16, 15187
1515  DATA  14712, 251, 32, 26, 52, 96, 32, 149, 50, 15400
1516  DATA  14720, 72, 32, 146, 57, 32, 125, 52, 104, 15340
1517  DATA  14728, 32, 175, 57, 32, 1, 58, 32, 125, 15240
1518  DATA  14736, 55, 96, 162, 3, 142, 2, 57, 170, 15423
1519  DATA  14744, 189, 0, 60, 170, 189, 80, 59, 142, 15633
1520  DATA  14752, 3, 57, 32, 64, 52, 174, 3, 57, 15194
1521  DATA  14760, 232, 206, 2, 57, 208, 238, 96, 170, 15969
1522  DATA  14768, 189, 0, 61, 170, 32, 184, 57, 96, 15557
1523  DATA  14776, 189, 38, 59, 141, 4, 57, 232, 189, 15685
1524  DATA  14784, 38, 59, 141, 5, 57, 108, 4, 57, 15253
1525  DATA  14792, 32, 13, 51, 32, 148, 53, 96, 32, 15249
1526  DATA  14800, 13, 51, 32, 149, 50, 72, 32, 13, 15212
1527  DATA  14808, 51, 32, 148, 53, 104, 32, 131, 52, 15411
```

```
1528  DATA  14816, 96, 169, 40, 208, 2, 169, 41, 32, 15573
1529  DATA  14824, 64, 52, 96, 169, 44, 32, 64, 52, 15397
1530  DATA  14832, 169, 88, 32, 64, 52, 96, 169, 44, 15546
1531  DATA  14840, 32, 64, 52, 169, 89, 32, 64, 52, 15394
1532  DATA  14848, 96, 141, 7, 57, 142, 6, 57, 202, 15556
1533  DATA  14856, 48, 6, 32, 26, 51, 202, 16, 250, 15487
1534  DATA  14864, 8, 216, 56, 173, 8, 57, 233, 4, 15619
1535  DATA  14872, 237, 7, 57, 40, 170, 32, 150, 52, 15617
1536  DATA  14880, 32, 155, 53, 32, 125, 52, 32, 148, 15509
1537  DATA  14888, 53, 173, 3, 48, 56, 201, 24, 144, 15590
1538  DATA  14896, 3, 32, 125, 52, 32, 13, 51, 206, 15410
1539  DATA  14904, 6, 57, 16, 234, 32, 26, 51, 32, 15358
1540  DATA  14912, 114, 52, 96, 91, 0, 255, 0, 255, 15775

----------->  END OF APPENDIX E6  <----------
```

Appendix E7:

Table-Driven Disassembler (Addressing Mode Subroutines)

```
APPENDIX E7:      DISASSEMBLER (ADDRESSING MODE SUBROUTINES)


THE FOLLOWING DATA STATEMENTS CONTAIN
DECIMAL OBJECT CODE AND CHECKSUMS
FOR MEMORY FROM 14928 TO 15171
SUITABLE FOR LOADING WITH THE
BASIC OBJECT CODE LOADER.

1600  DATA  14928, 32, 207, 57, 162, 2, 169, 4, 96, 15657
1601  DATA  14936, 32, 80, 58, 32, 235, 57, 162, 2, 15594
1602  DATA  14944, 169, 6, 96, 32, 80, 58, 32, 246, 15663
1603  DATA  14952, 57, 162, 2, 169, 6, 96, 169, 65, 15678
1604  DATA  14960, 32, 64, 52, 162, 0, 169, 1, 96, 15536
1605  DATA  14968, 162, 0, 169, 0, 96, 169, 35, 32, 15631
1606  DATA  14976, 64, 52, 169, 36, 32, 64, 52, 32, 15477
1607  DATA  14984, 200, 57, 162, 1, 169, 4, 96, 32, 15705
1608  DATA  14992, 225, 57, 32, 80, 58, 32, 229, 57, 15762
1609  DATA  15000, 169, 6, 162, 2, 96, 32, 225, 57, 15749
1610  DATA  15008, 32, 243, 58, 32, 229, 57, 162, 1, 15822
1611  DATA  15016, 169, 6, 96, 32, 225, 57, 32, 235, 15868
1612  DATA  15024, 58, 32, 229, 57, 32, 246, 57, 162, 15897
1613  DATA  15032, 1, 169, 6, 96, 32, 13, 51, 32, 15432
1614  DATA  15040, 18, 53, 32, 149, 50, 72, 32, 13, 15459
1615  DATA  15048, 51, 104, 201, 0, 16, 3, 206, 6, 15635
1616  DATA  15056, 50, 8, 216, 24, 109, 5, 50, 144, 15662
1617  DATA  15064, 3, 238, 6, 50, 141, 5, 50, 40, 15597
1618  DATA  15072, 32, 155, 53, 32, 43, 53, 162, 1, 15603
1619  DATA  15080, 169, 4, 96, 32, 200, 57, 162, 1, 15801
1620  DATA  15088, 169, 2, 96, 32, 235, 58, 32, 235, 15947
1621  DATA  15096, 57, 162, 1, 169, 4, 96, 32, 235, 15852
1622  DATA  15104, 58, 32, 246, 57, 162, 1, 169, 4, 15833
1623  DATA  15112, 96, 104, 104, 104, 104, 32, 125, 55, 15836
1624  DATA  15120, 48, 13, 32, 149, 50, 201, 255, 240, 16108
1625  DATA  15128, 6, 32, 64, 52, 24, 144, 238, 32, 15720
1626  DATA  15136, 114, 52, 32, 125, 55, 96, 120, 58, 15788
1627  DATA  15144, 110, 58, 125, 58, 235, 58, 243, 58, 16089
```

```
1628  DATA  15152, 254, 58, 80, 58, 88, 58, 99, 58, 15905
1629  DATA  15160, 120, 58, 188, 58, 157, 58, 171, 58, 16028
1630  DATA  15168, 143, 58, 9, 59, 0, 255, 0, 255, 15947
```

```
---------->  END OF APPENDIX E7  <----------
```

Appendix E8:

Table-Driven Disassembler (Tables)

```
APPENDIX E8:          DISASSEMBLER (TABLES)


THE FOLLOWING DATA STATEMENTS CONTAIN
DECIMAL OBJECT CODE AND CHECKSUMS
FOR MEMORY FROM 15184 TO 15871
SUITABLE FOR LOADING WITH THE
BASIC OBJECT CODE LOADER.


1700  DATA  15184, 127, 66, 65, 68, 65, 68, 67, 65, 15775
1701  DATA  15192, 78, 68, 65, 83, 76, 66, 67, 67, 15762
1702  DATA  15200, 66, 67, 83, 66, 69, 81, 66, 73, 15771
1703  DATA  15208, 84, 66, 77, 73, 66, 78, 69, 66, 15787
1704  DATA  15216, 80, 76, 66, 82, 75, 66, 86, 67, 15814
1705  DATA  15224, 66, 86, 83, 67, 76, 67, 67, 76, 15812
1706  DATA  15232, 68, 67, 76, 73, 67, 76, 86, 67, 15812
1707  DATA  15240, 77, 80, 67, 80, 88, 67, 80, 89, 15868
1708  DATA  15248, 68, 69, 67, 68, 69, 88, 68, 69, 15814
1709  DATA  15256, 89, 69, 79, 82, 73, 78, 67, 73, 15866
1710  DATA  15264, 78, 88, 73, 78, 89, 74, 77, 80, 15901
1711  DATA  15272, 74, 83, 82, 76, 68, 65, 76, 68, 15864
1712  DATA  15280, 88, 76, 68, 89, 76, 83, 82, 78, 15920
1713  DATA  15288, 79, 80, 79, 82, 65, 80, 72, 65, 15890
1714  DATA  15296, 80, 72, 80, 80, 76, 65, 80, 76, 15905
1715  DATA  15304, 80, 82, 79, 76, 82, 79, 82, 82, 15946
1716  DATA  15312, 84, 73, 82, 84, 83, 83, 66, 67, 15934
1717  DATA  15320, 83, 69, 67, 83, 69, 68, 83, 69, 15911
1718  DATA  15328, 73, 83, 84, 65, 83, 84, 88, 83, 15971
1719  DATA  15336, 84, 89, 84, 65, 88, 84, 65, 89, 15984
1720  DATA  15344, 84, 83, 88, 84, 88, 65, 84, 88, 16008
1721  DATA  15352, 83, 84, 89, 65, 84, 69, 88, 255, 16169
1722  DATA  15360, 34, 106, 1, 1, 1, 106, 10, 1, 15620
1723  DATA  15368, 112, 106, 10, 1, 1, 106, 10, 1, 15715
1724  DATA  15376, 31, 106, 1, 1, 1, 106, 10, 1, 15633
1725  DATA  15384, 43, 106, 1, 1, 1, 106, 10, 1, 15653
1726  DATA  15392, 88, 7, 1, 1, 22, 7, 121, 1, 15640
1727  DATA  15400, 118, 7, 121, 1, 22, 7, 121, 1, 15796
```

```
1728 DATA 15408, 25, 7, 1, 1, 1, 7, 121, 1, 15572
1729 DATA 15416, 136, 7, 1, 1, 1, 7, 121, 1, 15691
1730 DATA 15424, 127, 73, 1, 1, 1, 73, 100, 1, 15801
1731 DATA 15432, 109, 73, 100, 1, 85, 73, 100, 1, 15974
1732 DATA 15440, 37, 73, 1, 1, 1, 73, 100, 1, 15727
1733 DATA 15448, 49, 73, 1, 1, 1, 73, 100, 1, 15747
1734 DATA 15456, 130, 4, 1, 1, 1, 4, 124, 1, 15722
1735 DATA 15464, 115, 4, 124, 1, 85, 4, 124, 1, 15922
1736 DATA 15472, 40, 4, 1, 1, 1, 4, 124, 1, 15648
1737 DATA 15480, 142, 4, 1, 1, 1, 4, 124, 172, 15929
1738 DATA 15488, 1, 145, 1, 1, 151, 145, 148, 1, 16081
1739 DATA 15496, 70, 1, 163, 1, 151, 145, 148, 1, 16176
1740 DATA 15504, 13, 145, 1, 1, 151, 145, 148, 1, 16109
1741 DATA 15512, 169, 145, 163, 1, 1, 145, 1, 1, 16138
1742 DATA 15520, 97, 91, 94, 1, 97, 91, 94, 1, 16086
1743 DATA 15528, 157, 91, 154, 1, 97, 91, 94, 1, 16214
1744 DATA 15536, 16, 91, 1, 1, 97, 91, 94, 1, 15928
1745 DATA 15544, 52, 91, 158, 1, 97, 91, 94, 1, 16129
1746 DATA 15552, 61, 55, 1, 1, 61, 55, 64, 1, 15851
1747 DATA 15560, 82, 55, 67, 1, 61, 55, 64, 1, 15946
1748 DATA 15568, 28, 55, 1, 1, 1, 55, 64, 1, 15774
1749 DATA 15576, 46, 55, 1, 1, 1, 55, 64, 1, 15800
1750 DATA 15584, 58, 133, 1, 1, 58, 133, 76, 1, 16045
1751 DATA 15592, 79, 133, 103, 1, 58, 133, 76, 1, 16176
1752 DATA 15600, 19, 133, 1, 1, 1, 133, 76, 1, 15965
1753 DATA 15608, 139, 133, 1, 1, 1, 133, 76, 1, 16093
1754 DATA 15616, 18, 22, 0, 0, 0, 6, 6, 0, 15668
1755 DATA 15624, 18, 4, 2, 0, 0, 12, 12, 0, 15672
1756 DATA 15632, 20, 24, 0, 0, 0, 14, 14, 0, 15704
1757 DATA 15640, 18, 16, 0, 0, 0, 22, 22, 0, 15718
1758 DATA 15648, 12, 22, 0, 0, 6, 6, 6, 0, 15700
1759 DATA 15656, 18, 4, 2, 0, 12, 12, 12, 0, 15716
1760 DATA 15664, 20, 24, 0, 0, 0, 8, 8, 0, 15724
1761 DATA 15672, 18, 16, 0, 0, 0, 14, 14, 0, 15734
1762 DATA 15680, 18, 22, 0, 0, 0, 6, 6, 0, 15732
1763 DATA 15688, 18, 12, 2, 0, 12, 12, 12, 0, 15756
1764 DATA 15696, 20, 24, 0, 0, 0, 8, 8, 0, 15756
1765 DATA 15704, 18, 16, 0, 0, 0, 14, 14, 0, 15766
1766 DATA 15712, 18, 22, 0, 0, 0, 6, 6, 0, 15764
1767 DATA 15720, 18, 4, 2, 0, 26, 12, 12, 0, 15794
1768 DATA 15728, 20, 24, 0, 0, 0, 8, 8, 0, 15788
1769 DATA 15736, 18, 16, 0, 0, 0, 14, 14, 28, 15826
1770 DATA 15744, 0, 22, 0, 0, 6, 6, 6, 0, 15784
1771 DATA 15752, 18, 0, 18, 0, 12, 12, 12, 0, 15824
1772 DATA 15760, 20, 24, 0, 0, 8, 8, 10, 0, 15830
1773 DATA 15768, 18, 16, 18, 0, 0, 14, 0, 0, 15834
1774 DATA 15776, 4, 22, 4, 0, 6, 6, 6, 0, 15824
1775 DATA 15784, 18, 4, 18, 0, 12, 12, 12, 0, 15860
1776 DATA 15792, 20, 24, 0, 0, 8, 8, 10, 0, 15862
1777 DATA 15800, 20, 16, 18, 0, 14, 14, 16, 0, 15898
1778 DATA 15808, 4, 22, 0, 0, 6, 6, 6, 0, 15852
1779 DATA 15816, 18, 4, 18, 0, 12, 12, 12, 0, 15892
1780 DATA 15824, 20, 24, 0, 0, 0, 8, 8, 0, 15884
1781 DATA 15832, 18, 16, 0, 0, 0, 14, 14, 0, 15894
```

```
1782  DATA  15840, 4, 22, 0, 0, 6, 6, 6, 0, 15884
1783  DATA  15848, 18, 4, 18, 0, 12, 12, 12, 0, 15924
1784  DATA  15856, 20, 24, 0, 0, 0, 8, 8, 0, 15916
1785  DATA  15864, 18, 16, 0, 0, 0, 14, 14, 0, 15926
```

-----------> END OF APPENDIX E8 <----------

Appendix E9:

Move Utilities

```
APPENDIX E9:          MOVE UTILITIES


THE FOLLOWING DATA STATEMENTS CONTAIN
DECIMAL OBJECT CODE AND CHECKSUMS
FOR MEMORY FROM 14256 TO 14574
SUITABLE FOR LOADING WITH THE
BASIC OBJECT CODE LOADER.


1800  DATA  14256, 0, 0, 0, 0, 32, 8, 52, 32, 14380
1801  DATA  14264, 228, 52, 127, 13, 10, 32, 32, 32, 14790
1802  DATA  14272, 32, 32, 77, 79, 86, 69, 32, 84, 14763
1803  DATA  14280, 79, 79, 76, 46, 13, 10, 10, 255, 14848
1804  DATA  14288, 32, 227, 53, 32, 185, 56, 174, 85, 15132
1805  DATA  14296, 53, 56, 173, 84, 53, 237, 82, 53, 15087
1806  DATA  14304, 141, 176, 55, 176, 2, 202, 56, 138, 15250
1807  DATA  14312, 237, 83, 53, 141, 177, 55, 176, 3, 15237
1808  DATA  14320, 169, 0, 96, 160, 3, 185, 251, 0, 15184
1809  DATA  14328, 72, 136, 16, 249, 56, 173, 83, 53, 15166
1810  DATA  14336, 205, 179, 55, 144, 64, 208, 24, 173, 15388
1811  DATA  14344, 82, 53, 205, 178, 55, 144, 54, 208, 15323
1812  DATA  14352, 14, 160, 0, 104, 153, 251, 0, 200, 15234
1813  DATA  14360, 192, 4, 208, 247, 169, 255, 96, 32, 15563
1814  DATA  14368, 164, 56, 160, 0, 174, 177, 55, 240, 15394
1815  DATA  14376, 14, 177, 251, 145, 253, 200, 208, 249, 15873
1816  DATA  14384, 230, 252, 230, 254, 202, 208, 242, 136, 16138
1817  DATA  14392, 200, 177, 251, 145, 253, 204, 176, 55, 15853
1818  DATA  14400, 208, 246, 76, 17, 56, 173, 177, 55, 15408
1819  DATA  14408, 240, 72, 172, 177, 55, 173, 176, 55, 15528
1820  DATA  14416, 56, 233, 255, 176, 1, 136, 170, 132, 15575
1821  DATA  14424, 254, 138, 24, 109, 82, 53, 133, 251, 15468
1822  DATA  14432, 144, 1, 200, 152, 109, 83, 53, 133, 15307
1823  DATA  14440, 252, 138, 24, 109, 178, 55, 133, 253, 15582
1824  DATA  14448, 144, 2, 230, 254, 165, 254, 109, 179, 15785
1825  DATA  14456, 55, 133, 254, 174, 177, 55, 160, 255, 15719
1826  DATA  14464, 177, 251, 145, 253, 136, 208, 249, 177, 16060
1827  DATA  14472, 251, 145, 253, 198, 252, 198, 254, 202, 16225
```

```
1828  DATA  14480, 208, 236, 32, 164, 56, 172, 176, 55, 15579
1829  DATA  14488, 177, 251, 145, 253, 136, 192, 255, 208, 16105
1830  DATA  14496, 247, 76, 17, 56, 173, 82, 53, 133, 15333
1831  DATA  14504, 251, 173, 83, 53, 133, 252, 173, 178, 15800
1832  DATA  14512, 55, 133, 253, 173, 179, 55, 133, 254, 15747
1833  DATA  14520, 96, 32, 8, 52, 32, 228, 52, 127, 15147
1834  DATA  14528, 13, 10, 83, 69, 84, 32, 68, 69, 14956
1835  DATA  14536, 83, 84, 73, 78, 65, 84, 73, 79, 15155
1836  DATA  14544, 78, 32, 65, 78, 68, 32, 80, 82, 15059
1837  DATA  14552, 69, 83, 83, 32, 81, 46, 255, 32, 15233
1838  DATA  14560, 7, 50, 173, 5, 50, 141, 178, 55, 15219
1839  DATA  14568, 173, 6, 50, 141, 179, 55, 96, 255, 15523

---------->  END OF APPENDIX E9  <----------
```

Appendix E10:

Simple Text Editor

```
APPENDIX E10:          A SIMPLE TEXT EDITOR



THE FOLLOWING DATA STATEMENTS CONTAIN
DECIMAL OBJECT CODE AND CHECKSUMS
FOR MEMORY FROM 15872 TO 16383
SUITABLE FOR LOADING WITH THE
BASIC OBJECT CODE LOADER.

1900  DATA  15872, 0, 0, 32, 15, 62, 32, 55, 62, 16130
1901  DATA  15880, 32, 200, 62, 24, 24, 144, 246, 32, 16644
1902  DATA  15888, 8, 52, 32, 228, 52, 127, 13, 10, 16410
1903  DATA  15896, 10, 83, 69, 84, 32, 85, 80, 32, 16371
1904  DATA  15904, 69, 68, 73, 84, 32, 66, 85, 70, 16451
1905  DATA  15912, 70, 69, 82, 46, 13, 10, 10, 255, 16467
1906  DATA  15920, 32, 227, 53, 32, 154, 55, 96, 32, 16601
1907  DATA  15928, 196, 49, 32, 43, 49, 174, 3, 48, 16522
1908  DATA  15936, 160, 3, 32, 19, 49, 32, 43, 49, 16323
1909  DATA  15944, 32, 118, 49, 32, 196, 49, 32, 94, 16546
1910  DATA  15952, 62, 32, 211, 49, 32, 118, 49, 32, 16537
1911  DATA  15960, 136, 62, 32, 211, 49, 96, 32, 18, 16596
1912  DATA  15968, 53, 173, 3, 48, 74, 170, 202, 32, 16723
1913  DATA  15976, 26, 51, 202, 16, 250, 173, 3, 48, 16745
1914  DATA  15984, 141, 0, 62, 32, 149, 50, 32, 155, 16605
1915  DATA  15992, 49, 32, 127, 49, 32, 13, 51, 206, 16551
1916  DATA  16000, 0, 62, 16, 239, 32, 43, 53, 96, 16541
1917  DATA  16008, 173, 3, 48, 74, 233, 2, 32, 129, 16702
1918  DATA  16016, 49, 173, 1, 62, 201, 1, 208, 5, 16716
1919  DATA  16024, 169, 73, 24, 144, 2, 169, 79, 32, 16716
1920  DATA  16032, 155, 49, 169, 2, 32, 129, 49, 173, 16790
1921  DATA  16040, 7, 48, 32, 155, 49, 169, 2, 32, 16534
1922  DATA  16048, 129, 49, 173, 6, 50, 32, 163, 49, 16699
1923  DATA  16056, 173, 5, 50, 32, 163, 49, 96, 255, 16879
1924  DATA  16064, 147, 148, 29, 157, 16, 20, 81, 0, 16662
1925  DATA  16072, 32, 224, 50, 205, 198, 62, 208, 23, 17074
1926  DATA  16080, 72, 32, 224, 50, 205, 198, 62, 208, 17131
1927  DATA  16088, 4, 104, 104, 104, 96, 141, 199, 62, 16902
```

```
1928  DATA  16096, 104, 32, 231, 62, 173, 199, 62, 205, 17164
1929  DATA  16104, 193, 62, 208, 11, 206, 1, 62, 16, 16863
1930  DATA  16112, 5, 169, 1, 141, 1, 62, 96, 205, 16792
1931  DATA  16120, 194, 62, 208, 4, 32, 121, 63, 96, 16900
1932  DATA  16128, 205, 195, 62, 208, 4, 32, 135, 63, 17032
1933  DATA  16136, 96, 205, 197, 62, 208, 4, 32, 221, 17161
1934  DATA  16144, 63, 96, 205, 196, 62, 208, 4, 32, 17010
1935  DATA  16152, 197, 63, 96, 205, 192, 62, 208, 4, 17179
1936  DATA  16160, 32, 180, 63, 96, 174, 1, 62, 240, 17008
1937  DATA  16168, 4, 32, 52, 63, 96, 32, 45, 51, 16543
1938  DATA  16176, 32, 125, 55, 96, 72, 32, 18, 53, 16659
1939  DATA  16184, 173, 83, 53, 72, 173, 82, 53, 72, 16945
1940  DATA  16192, 173, 85, 53, 72, 173, 84, 53, 72, 16957
1941  DATA  16200, 32, 97, 54, 32, 125, 55, 48, 17, 16660
1942  DATA  16208, 32, 226, 56, 173, 84, 53, 208, 3, 17043
1943  DATA  16216, 206, 85, 53, 206, 84, 53, 32, 214, 17149
1944  DATA  16224, 55, 104, 141, 84, 53, 104, 141, 85, 16991
1945  DATA  16232, 53, 104, 141, 82, 53, 104, 141, 83, 16993
1946  DATA  16240, 53, 32, 43, 53, 104, 32, 45, 63, 16665
1947  DATA  16248, 96, 32, 149, 50, 201, 255, 240, 4, 17275
1948  DATA  16256, 32, 125, 55, 96, 169, 255, 96, 56, 17140
1949  DATA  16264, 173, 83, 53, 205, 6, 50, 144, 12, 16990
1950  DATA  16272, 208, 16, 173, 82, 53, 205, 5, 50, 17064
1951  DATA  16280, 240, 23, 176, 6, 32, 26, 51, 169, 17003
1952  DATA  16288, 0, 96, 173, 82, 53, 141, 5, 50, 16888
1953  DATA  16296, 173, 83, 53, 141, 6, 50, 169, 0, 16971
1954  DATA  16304, 96, 169, 255, 96, 32, 154, 55, 169, 17330
1955  DATA  16312, 255, 32, 45, 51, 32, 125, 55, 16, 16923
1956  DATA  16320, 246, 32, 154, 55, 96, 32, 154, 55, 17144
1957  DATA  16328, 32, 20, 52, 32, 149, 50, 201, 255, 17119
1958  DATA  16336, 240, 8, 32, 64, 52, 32, 125, 55, 16944
1959  DATA  16344, 16, 241, 76, 26, 52, 32, 18, 53, 16858
1960  DATA  16352, 173, 83, 53, 72, 173, 82, 53, 72, 17113
1961  DATA  16360, 32, 226, 56, 32, 125, 55, 32, 97, 17015
1962  DATA  16368, 54, 32, 214, 55, 104, 141, 82, 53, 17103
1963  DATA  16376, 104, 141, 83, 53, 32, 43, 53, 96, 16981
```

----------> END OF APPENDIX E10 <----------

Appendix E11:

Extending the Visible Monitor

```
APPENDIX E11:          EXTENDING THE VISIBLE MONITOR


THE FOLLOWING DATA STATEMENTS CONTAIN
DECIMAL OBJECT CODE AND CHECKSUMS
FOR MEMORY FROM 12464 TO 12540
SUITABLE FOR LOADING WITH THE
BASIC OBJECT CODE LOADER.


2000  DATA  12464, 201, 80, 208, 9, 173, 0, 52, 73, 13260
2001  DATA  12472, 255, 141, 0, 52, 96, 201, 85, 208, 13510
2002  DATA  12480, 9, 173, 2, 52, 73, 255, 141, 2, 13187
2003  DATA  12488, 52, 96, 201, 72, 208, 13, 173, 0, 13303
2004  DATA  12496, 52, 208, 4, 32, 87, 53, 96, 32, 13060
2005  DATA  12504, 168, 53, 96, 201, 77, 208, 4, 32, 13343
2006  DATA  12512, 180, 55, 96, 201, 63, 208, 13, 173, 13501
2007  DATA  12520, 0, 52, 208, 4, 32, 9, 57, 96, 12978
2008  DATA  12528, 32, 38, 57, 96, 201, 84, 208, 4, 13248
2009  DATA  12536, 32, 2, 62, 96, 96, 126, 145, 154, 13249

---------->  END OF APPENDIX E11  <----------
```

Appendix E12:

System Data Block for the VIC-20

```
APPENDIX E12:          SYSTEM DATA BLOCK FOR THE VIC-20



THE FOLLOWING DATA STATEMENTS CONTAIN
DECIMAL OBJECT CODE AND CHECKSUMS
FOR MEMORY FROM 12288 TO 12458
SUITABLE FOR LOADING WITH THE
BASIC OBJECT CODE LOADER.


2100  DATA  12288, 0, 16, 22, 21, 24, 17, 32, 30, 12450
2101  DATA  12296, 53, 48, 60, 48, 65, 48, 16, 48, 12682
2102  DATA  12304, 96, 72, 165, 252, 72, 24, 105, 132, 13222
2103  DATA  12312, 133, 252, 160, 0, 173, 134, 2, 145, 13311
2104  DATA  12320, 251, 104, 133, 252, 104, 56, 201, 64, 13485
2105  DATA  12328, 144, 10, 201, 96, 144, 3, 233, 32, 13191
2106  DATA  12336, 96, 56, 233, 64, 96, 32, 228, 255, 13396
2107  DATA  12344, 170, 240, 250, 96, 162, 1, 76, 67, 13406
2108  DATA  12352, 48, 162, 2, 72, 32, 201, 255, 104, 13228
2109  DATA  12360, 32, 210, 255, 96, 1, 0, 0, 0, 12954
2110  DATA  12368, 0, 0, 0, 0, 0, 0, 0, 0, 12368
2111  DATA  12376, 0, 0, 0, 0, 0, 0, 0, 0, 12376
2112  DATA  12384, 0, 0, 169, 3, 174, 76, 48, 168, 13022
2113  DATA  12392, 32, 186, 255, 173, 77, 48, 162, 78, 13403
2114  DATA  12400, 160, 48, 32, 189, 255, 173, 82, 53, 13392
2115  DATA  12408, 133, 253, 173, 83, 53, 133, 254, 169, 13659
2116  DATA  12416, 253, 174, 84, 53, 172, 85, 53, 232, 13522
2117  DATA  12424, 208, 1, 200, 32, 216, 255, 96, 169, 13601
2118  DATA  12432, 0, 72, 40, 32, 189, 255, 169, 1, 13190
2119  DATA  12440, 162, 0, 160, 255, 32, 186, 255, 32, 13522
2120  DATA  12448, 192, 255, 169, 3, 141, 81, 53, 32, 13374
2121  DATA  12456, 7, 50, 169, 1, 32, 195, 255, 96, 13261

---------->  END OF APPENDIX E12  <----------
```

Appendix E13:

System Data Block for the Commodore 64

```
APPENDIX E13:        SYSTEM DATA BLOCK FOR THE COMMODORE 64


THE FOLLOWING DATA STATEMENTS CONTAIN
DECIMAL OBJECT CODE AND CHECKSUMS
FOR MEMORY FROM 12288 TO 12458
SUITABLE FOR LOADING WITH THE
BASIC OBJECT CODE LOADER.

2100  DATA  12288, 0, 4, 40, 39, 24, 7, 32, 30, 12464
2101  DATA  12296, 53, 48, 60, 48, 65, 48, 16, 48, 12682
2102  DATA  12304, 96, 72, 165, 252, 72, 24, 105, 212, 13302
2103  DATA  12312, 133, 252, 160, 0, 173, 134, 2, 145, 13311
2104  DATA  12320, 251, 104, 133, 252, 104, 56, 201, 64, 13485
2105  DATA  12328, 144, 10, 201, 96, 144, 3, 233, 32, 13191
2106  DATA  12336, 96, 56, 233, 64, 96, 32, 228, 255, 13396
2107  DATA  12344, 170, 240, 250, 96, 162, 1, 76, 67, 13406
2108  DATA  12352, 48, 162, 2, 72, 32, 201, 255, 104, 13228
2109  DATA  12360, 32, 210, 255, 96, 1, 0, 0, 0, 12954
2110  DATA  12368, 0, 0, 0, 0, 0, 0, 0, 0, 12368
2111  DATA  12376, 0, 0, 0, 0, 0, 0, 0, 0, 12376
2112  DATA  12384, 0, 0, 169, 3, 174, 76, 48, 168, 13022
2113  DATA  12392, 32, 186, 255, 173, 77, 48, 162, 78, 13403
2114  DATA  12400, 160, 48, 32, 189, 255, 173, 82, 53, 13392
2115  DATA  12408, 133, 253, 173, 83, 53, 133, 254, 169, 13659
2116  DATA  12416, 253, 174, 84, 53, 172, 85, 53, 232, 13522
2117  DATA  12424, 208, 1, 200, 32, 216, 255, 96, 169, 13601
2118  DATA  12432, 0, 72, 40, 32, 189, 255, 169, 1, 13190
2119  DATA  12440, 162, 0, 160, 255, 32, 186, 255, 32, 13522
2120  DATA  12448, 192, 255, 32, 7, 50, 169, 1, 32, 13186
2121  DATA  12456, 195, 255, 96, 255, 0, 255, 0, 255, 13767

---------->  END OF APPENDIX E13  <----------
```

Appendix F1:

BASIC Program to Load Object Code for the Visible Monitor into Memory

```
        Appendix F1:    BASIC Program to Load Object Code
                        for the Visible Monitor into Memory

                        (An "E" appendix must be appended to
                        this program.  See Chapter 13.)

100 REM     OBJECT CODE LOADER
110 :
120 DIM BYTE(8)
130 READ FIRST
140 :
150 READ LAST
160 :
170 FOR LINE=FIRST TO LAST
180 GOSUB 300
190 NEXT LINE
200 PRINT "LOADED LINES",FIRST,"THROUGH",LAST,"SUCCESSFULLY."
210 END
300 READ A
310 SUM=A
320 FOR J=1 TO 8
330 READ BYTE(J)
340 SUM=SUM+BYTE(J)
350 NEXT J
370 READ CHECK
380 IF SUM<>CHECK THEN 500
390 FOR J=1 TO 8
400 POKE A+J-1,BYTE(J)
410 NEXT J
420 RETURN
500 PRINT "CHECKSUM ERROR IN DATA LINE",LINE
510 PRINT"START ADDRESS IN BAD DATA LINE IS",A
520 END
600 DATA ???  : REM  Number of first data line from an "E" appendix
620 DATA ???  : REM  Number of last data line from that "E" appendix.
690 :
700 REM  An "E" appendix must follow...
```

Appendix F2:

BASIC Program to SAVE a Machine Language Program to Tape or Disk

```
Appendix F2:     BASIC Program to SAVE
                 a Machine Language Program
                 to Tape or Disk

                 (Requires Extended
                 Visible Monitor)


10 DEVICE=12364
20 LNGTH =12365
30 NAME  =12366
40 MLSAV =12386
50 SETADS=13795
60 :
100 PRINT "SAVE A MACHINE LANGUAGE PROGRAM"
110 PRINT
120 INPUT "FILE NAME";NAME$
125 IF LEN(NAME$)>19 THEN NAME$=LEFT$(NAME$,19)
130 POKE LNGTH,LEN(NAME$)
140 IF LEN(NAME$)=0 THEN 200
150 :
160 FOR J=1 TO LEN(NAME$)
170 :   POKE NAME+J-1,ASC(MID$(NAME$,J))
180 NEXT J
190 :
200 PRINT "SAVE TO (T)APE OR (D)ISK?"
210 GET A$:IF LEN(A$)=0 THEN 210
220 IF A$="T" THEN POKE DEVICE,1:GOTO 300
230 IF A$="D" THEN POKE DEVICE,8:GOTO 300
240 PRINT "KEYSTROKE IGNORED.":PRINT:GOTO 200
250 :
300 SYS (SETADS)  : REM GET START, END ADDRESSES
310 SYS (MLSAV)   : REM SAVE THE PROGRAM ON DISK OR TAPE.
```

Index

General Interest
Computers

TOP-DOWN ASSEMBLY LANGUAGE PROGRAMMING *for your* VIC-20™ AND COMMODORE 64™

KEN SKIER

Now you can learn about assembly language from the top down! Learn how it works and how to make it work for you. This book, for VIC-20™ and Commodore 64™ computer owners who know little or nothing about bits, bytes, hardware, and software, presents a guided tour of your computer. Beginning with basic concepts such as *what is memory?* and *what is a program?*, **Top-Down Assembly Language** moves through a fast but surprisingly complete course in assembly language programming. Having mastered these fundamentals, the reader is introduced to many useful subroutines and programming tools, such as screen utilities, print utilities, a machine language monitor, a hexadecimal dump tool, a move tool, a disassembler, and a simple, screen-based text editor.

About the Author

KEN SKIER, the President of SkiSoft, Inc., of Cambridge, Massachusetts, develops software and documentation for personal computers. He created SkiWriter,™ the word processor bundled with the Epson HX-20 portable computer. Before founding SkiSoft, he designed word processing software for Wang Laboratories and co-founded the Writing Program at MIT. Articles of his appear regularly in *Popular Computing.* His other books include *Top-Down Assembly Language Programming for the 6502 Personal Computer* (Byte Books, 1983); *Top-Down BASIC for the TRS-80 Color Computer* (Byte Books, 1983); and the Operations and BASIC Manuals for the Epson HX-20 portable computer. Mr. Skier lives in Lexington, Massachusetts, with his wife and daughter.

Cover: Bill Graef

McGraw-Hill Book Company
Serving the Need for Knowledge
1221 Avenue of the Americas
New York, NY 10020

ISBN 0-07-057864-8