Beyond Games: Systems Software for Your 6502 Personal Computer

Ken Skier
Beyond Games: Systems Software for Your 6502 Personal Computer

Copyright © 1981 BYTE Publications Inc. All Rights Reserved. No part of this book may be translated or reproduced in any form without the prior written consent of BYTE Publications Inc.

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 BYTE Publications Inc. however, makes any warranties concerning the programs and assumes no 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.

Library of Congress Cataloging in Publication Data

Skier, Ken.
Beyond games.

Includes index.
1. 6502 (Computer)—Programming. I. Title.
QA76.8.S63559 001.64'2 80-28512

Cover photo by Bob Hamor

Text set in Paladium by BYTE Publications
Edited by Blaise Lifick
Design and Production Supervision by Ellen Klempner
Copy Edited by Rich Friedman
Printed and bound using 50#MH Matte
Table of Contents

Introduction ................................................................. 1
Chapter 1: Your Computer ................................................. 3
Chapter 2: Introduction to Assembler ................................. 8
Chapter 3: Loops and Subroutines ..................................... 20
Chapter 4: Arithmetic and Logic ....................................... 32
Chapter 5: Screen Utilities ............................................... 44
Chapter 6: The Visible Monitor ......................................... 60
Chapter 7: Print Utilities ................................................ 84
Chapter 8: Two Hexdump Tools ......................................... 98
Chapter 9: A Table-Driven Disassembler ............................ 114
Chapter 10: A General MOVE Utility ................................. 134
Chapter 11: A Simple Text Editor ...................................... 145
Chapter 12: Extending the Visible Monitor .......................... 160
Chapter 13: Entering the Software into Your System ............. 163

Appendices:

A1: Hexadecimal Conversion Table ................................. 171
A2: ASCII Character Codes .............................................. 172
A3: 6502 Instruction Set — Mnemonic List .......................... 173
A4: 6502 Instruction Set — Opcode List ............................. 175
A5: Instruction Execution Times ....................................... 178
A6: 6502 Opcodes by Mnemonic and Addressing Mode .......... 181
B1: The Ohio Scientific Challenger 1-P ............................. 185
B2: The PET 2001 ............................................................ 188
B3: The Apple II ............................................................. 196
B4: The Atari 800 ........................................................... 204
C1: Screen Utilities ......................................................... 211
C2: Visible Monitor (Top Level and Display Subroutines) ....... 223
C3: Visible Monitor (Update Subroutine) ............................. 233
C4: Print Utilities ............................................................ 243
C5: Two Hexdump Tools .................................................. 257
C6: Table-Driven Disassembler (Top Level and Utility Subroutines) 275
C7: Table-Driven Disassembler (Addressing Mode Subroutines) 287
C8: Table-Driven Disassembler (Tables) 299
C9: Move Utilities 317
C10: Simple Text Editor (Top Level and Display Subroutines) 329
C11: Simple Text Editor (EDITIT Subroutines) 337
C12: Extending the Visible Monitor 349
C13: System Data Block for the Ohio Scientific C-1P 355
C14: System Data Block for the PET 2001 361
C15: System Data Block for the Apple II 367
C16: System Data Block for the Atari 800 373
D1: Screen Utilities 389
D2: Visible Monitor (Top Level and Display Subroutines) 390
D3: Visible Monitor (Update Subroutine) 391
D4: Print Utilities 392
D5: Two Hexdump Tools 393
D6: Table-Driven Disassembler (Top Level and Utility Subroutines) 395
D7: Table-Driven Disassembler (Addressing Mode Subroutines) 396
D8: Table-Driven Disassembler (Tables) 397
D9: Move Utilities 399
D10: Simple Text Editor 400
D11: Extending the Visible Monitor 401
E1: Screen Utilities 403
E2: Visible Monitor (Top Level and Display Subroutines) 405
E3: Visible Monitor (Update Subroutines) 407
E4: Print Utilities 409
E5: Two Hexdump Tools 411
E6: Table-Driven Disassembler (Top Level and Utility Subroutines) 413
E7: Table-Driven Disassembler (Addressing Mode Subroutine) 415
E8: Table-Driven Disassembler (Tables) 417
E9: Move Utilities 419
E10: Simple Text Editor 421
E11: Extending the Visible Monitor 423
E12: System Data Block for the Ohio Scientific C-1P 424
E13: System Data Block for the PET 2001 425
E14: System Data Block for the Apple II 426
E15: System Data Block for the Atari 800 427
Index 429
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 program-
ning 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 an Apple II, an Atari 400 or 800, an Ohio Scientific (OSI) Challenger I-P, or a PET 2001. 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 second 4 K bytes of memory (hexadecimal memory locations 1000 thru 1FFF), 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.

**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

The software in this book can run on a number of computers because it assumes very little about the host machine. Let's examine these assumptions and in so doing 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 (i.e. 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 ex-
cept 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.

![Diagram of 6502, 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, termi-
nals, burglar alarms — virtually anything that can generate an electrical signal.
The 6502 perceives these externally generated signals by reading the appropriate in-
put 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.

![Diagram of 6502 microprocessor access to memory and peripherals]

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 Apple, Atari OSI, and PET computers feature more than hardware. For example, all 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 I assume your system does have such routines.

There are many other subroutines in your computer’s operating system. Your system’s documentation should tell you what subroutines are available and provide their addresses. 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 the Apple, the Atari, the OSI, and the PET 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 should tell 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.

Unfortunately, not every dialect of BASIC is the same. A program written in BASIC that runs on machine A may not run on machine B. BASIC is a common language, but not a standard one. Is there any language that is standard from system to system?

6502 Code

The central processor is the computer's heart. The Apple, Atari, OSI, and PET computers all 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.

However, a 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 opera-
tions 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.

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>b7</th>
<th>b6</th>
<th>b5</th>
<th>b4</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Significance</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Hexadecimal Character</th>
<th>Decimal Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>0</td>
</tr>
<tr>
<td>$1$</td>
<td>1</td>
</tr>
<tr>
<td>$2$</td>
<td>2</td>
</tr>
<tr>
<td>$3$</td>
<td>3</td>
</tr>
<tr>
<td>$4$</td>
<td>4</td>
</tr>
<tr>
<td>$5$</td>
<td>5</td>
</tr>
<tr>
<td>$6$</td>
<td>6</td>
</tr>
<tr>
<td>$7$</td>
<td>7</td>
</tr>
<tr>
<td>$8$</td>
<td>8</td>
</tr>
<tr>
<td>$9$</td>
<td>9</td>
</tr>
<tr>
<td>$A$</td>
<td>10</td>
</tr>
<tr>
<td>$B$</td>
<td>11</td>
</tr>
<tr>
<td>$C$</td>
<td>12</td>
</tr>
<tr>
<td>$D$</td>
<td>13</td>
</tr>
<tr>
<td>$E$</td>
<td>14</td>
</tr>
<tr>
<td>$F$</td>
<td>15</td>
</tr>
</tbody>
</table>

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 American Standard Code for Information Interchange, 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.

![Diagram](image)

**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. It is likely that the source code presented in this book will assemble immediately or with only minor modification on other 6500 assemblers. (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:

<table>
<thead>
<tr>
<th>LABEL</th>
<th>MNEMONIC</th>
<th>OPERAND</th>
<th>COMMENT</th>
</tr>
</thead>
</table>

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 1**

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

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

ORIGIN = $0200
DEST = $0300
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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Carry Flag</td>
</tr>
<tr>
<td>1</td>
<td>Zero Flag</td>
</tr>
<tr>
<td>2</td>
<td>Interrupt Flag</td>
</tr>
<tr>
<td>3</td>
<td>Decimal Flag</td>
</tr>
<tr>
<td>4</td>
<td>Break Flag</td>
</tr>
<tr>
<td>5</td>
<td>Undefined</td>
</tr>
<tr>
<td>6</td>
<td>Overflow Flag</td>
</tr>
<tr>
<td>7</td>
<td>Negative Flag</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Carry Flag</th>
<th>Negative Flag</th>
<th>Zero Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare a register with an equal value and you</td>
<td>set C, clear N, and set Z.</td>
<td></td>
</tr>
<tr>
<td>Compare a register with a greater value and you</td>
<td>clear C, clear N, and clear Z.</td>
<td></td>
</tr>
<tr>
<td>Compare a register with a lesser value and you</td>
<td>set C, clear N, and clear Z.</td>
<td></td>
</tr>
</tbody>
</table>

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, (Branch on result EQual) and BNE (Branch on result Not Equal) 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.

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

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

\[
\begin{align*}
\text{ORIGIN} &= \$0200 \\
\text{DEST} &= \$0300
\end{align*}
\]

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

| 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 un-
conditionally branches to a specified address. A program may loop forever by JVM'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 xxxxxxxxxxxx some
       xxxxxxxxxxx
       xxxxxxxxxxx
       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:

\[
\begin{align*}
&\text{CLC} \\
&BCC \quad \text{PLACE}
\end{align*}
\]

rather than as:

\[
\begin{align*}
&\text{JMP} \quad \text{PLACE}
\end{align*}
\]

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 (Jump to SubRoutine) and the RTS (ReTurn from Subroutine). 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 do not affect the status flags.
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 (i.e. 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.](image)

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.](image)
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 (i.e., 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.](image-url)
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.](image)

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
ADC NUMBER
```

Clear the carry flag.
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
SBC OPERND
```

Set the carry flag.
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 sub-
tracting. 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 than 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.

<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OR</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>OR</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>OR</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>OR</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AND</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>AND</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>AND</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>AND</td>
<td>1</td>
</tr>
</tbody>
</table>
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 (i.e. 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).**

<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>XOR</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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
AND #$40
BEQ PLAN.B
```

Get flag byte.
Clear all bits but bit 6.
PLAN.A

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 in-
clude 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 Apple, Atari, OSI, and PET 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 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 thru B4 describe how display memory is mapped on the Apple, Atari, OSI, and PET 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 #$\$CD  A9  CD
STA POINTR  8D  00  10
LDA #$\$AB  A9  AB
STA POINTR+1  8D  01  10

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 TV.PTR must be in the zero page, let's place it at $0000$, meaning it will occupy the bytes at $0000$ and at $0001$. We can do that with the following assembler directive:

\[
\text{TV.PTR = } 0
\]

**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:

\[
\begin{align*}
\text{LDY} \#0 & \quad \text{A0 00} \\
\text{STA (TV.PTR),Y} & \quad 91 00
\end{align*}
\]

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" may have a different display code on your system. Instead of displaying an "A," we might display something else. Of the four computers considered in this book, only the Ohio Scientific Challenger I-P has a one-to-one correspondence between any character's ASCII code and that character's graphic code. The Atari, the PET, and the Apple computers lack such a one-to-one correspondence.

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
          LDY #0
          STA (TV.PTR), Y
          RTS
```

Convert ASCII character to your
system's display code for that character.
Put that graphic in the
current screen location.
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 cor-
ner). If you have an Ohio Scientific Challenger I-P, then you can consult your
system's documentation and determine that address $D062 in display memory cor-
responds to your desired screen location. $D0 is the high byte of this screen location;
$62 is the low byte of this screen location. Thus, you can set TV.PTR with the
following lines of code:

```
LDA #$62          A9  62  Set
STA TV.PTR        85  00  low byte.
LDA #$D0          A9  D0  Set
STA TV.PTR+1      85  01  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
LDY #4
JSR TVTOXY
```

The leftmost column is column zero, so the third column is column two.
The topmost row is row zero, so the fifth row is row four.
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
ROWINC
```

The address of the character in the upper-left corner of the screen (i.e: the lowest address in screen memory).
ROW INCReement: 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:

\[
\begin{align*}
\text{HOME} & + \ X \ \text{Register} \\
& + \ (Y \ \text{Register}) \times \text{ROWINC} \\
\hline
\text{TV.PTR} & \quad \text{Address of screen location at column } X, \text{ row } Y.
\end{align*}
\]

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:

\[
\begin{align*}
\text{TVCOLS} & \quad \text{The number of columns on the display screen, counting from zero.} \\
\text{TVROWS} & \quad \text{The number of rows on the display screen, counting from zero.}
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>TVTOXY</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEC</td>
<td>CPX TVCOLS</td>
<td>Is X out of range?</td>
</tr>
<tr>
<td></td>
<td>BCC X.OK</td>
<td>If not, leave it alone.</td>
</tr>
<tr>
<td></td>
<td>LDX TVCOLS</td>
<td>If X is out of range, give it its maximum legal value. Now X is legal.</td>
</tr>
<tr>
<td>X.OK</td>
<td>SEC</td>
<td>Is Y out of range?</td>
</tr>
<tr>
<td></td>
<td>CPY TVROWS</td>
<td>If not, leave it alone.</td>
</tr>
<tr>
<td></td>
<td>BCC Y.OK</td>
<td>If Y is out of range, give it its maximum legal value. Now Y is legal.</td>
</tr>
<tr>
<td></td>
<td>LDY TVROWS</td>
<td></td>
</tr>
<tr>
<td>Y.OK</td>
<td>LDA HOME</td>
<td>Set TV.PTR = HOME.</td>
</tr>
<tr>
<td></td>
<td>STA TV.PTR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDA HOME+1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STA TV.PTR+1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TXA</td>
<td>Add X to TV.PTR.</td>
</tr>
<tr>
<td></td>
<td>CLC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADC TV.PTR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BCC COLSET</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INC TV.PTR+1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLC</td>
<td></td>
</tr>
<tr>
<td>COLSET</td>
<td>CPY #0</td>
<td>Add Y*ROWINC to TV.PTR.</td>
</tr>
<tr>
<td>LOOP</td>
<td>BEQ EXIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADC ROWINC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BCC NEXT</td>
<td></td>
</tr>
</tbody>
</table>
INC TV.PTR+1
DEY
BNE LOOP
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**

<table>
<thead>
<tr>
<th>TVDOWN</th>
<th>LDA ROWINC</th>
<th>Move TV.PTR down by one row.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CLC</td>
<td>Unconditionally branch.</td>
</tr>
<tr>
<td></td>
<td>BCC TVPLUS</td>
<td></td>
</tr>
<tr>
<td>TVSKIP</td>
<td>LDA #1</td>
<td>Skip one screen location by incrementing TV.PTR.</td>
</tr>
<tr>
<td>TVPLUS</td>
<td>CLC</td>
<td>Add the contents of the accumulator to the two zero-page bytes comprising the TV.PTR.</td>
</tr>
<tr>
<td></td>
<td>ADC TV.PTR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BCC NEXT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INC TV.PTR+1</td>
<td></td>
</tr>
<tr>
<td>NEXT</td>
<td>STA TV.PTR</td>
<td>Return to caller.</td>
</tr>
<tr>
<td></td>
<td>RTS</td>
<td></td>
</tr>
</tbody>
</table>

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 character, let's call it VUCHAR:

VUCHAR    JSR TV.PUT
          JSR TVSKIP
          RTS

Display, in the current screen location, the graphic for the character whose ASCII code is in the accumulator. Advance to the next screen location.

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: 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:

<table>
<thead>
<tr>
<th>VUBYTE</th>
<th>PHA</th>
<th>Save accumulator.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSR A</td>
<td>Move 4 MSB into positions</td>
<td></td>
</tr>
<tr>
<td>LSR A</td>
<td>occupied by 4 LSB.</td>
<td></td>
</tr>
<tr>
<td>JSR ASCII</td>
<td>Determine ASCII for accumulator’s 4 LSB (which were its 4 MSB).</td>
<td></td>
</tr>
<tr>
<td>JSR VUCHAR</td>
<td>Display the ASCII character in the current screen location and advance to next screen location.</td>
<td></td>
</tr>
<tr>
<td>PLA</td>
<td>Restore original value of accumulator.</td>
<td></td>
</tr>
<tr>
<td>JSR ASCII</td>
<td>Determine ASCII for accumulator’s 4 LSB (which were its 4 LSB).</td>
<td></td>
</tr>
<tr>
<td>JSR VUCHAR</td>
<td>Display this ASCII character just to the right of the other ASCII character and advance to next screen location.</td>
<td></td>
</tr>
<tr>
<td>RTS</td>
<td>Return to caller.</td>
<td></td>
</tr>
</tbody>
</table>
Of course, ASCII doesn’t exist yet. So let’s write it, and then VUBYTE should be complete.

**ASCII**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>AND #$0F</td>
</tr>
<tr>
<td></td>
<td>Clear the 4 MSB in accumulator.</td>
</tr>
<tr>
<td>CMP #$0A</td>
<td>Is accumulator greater than 9?</td>
</tr>
<tr>
<td>BMI DECIML</td>
<td>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.)</td>
</tr>
<tr>
<td>ADC #6</td>
<td>If accumulator is 0 thru 9, add $30 to it to convert it to corresponding ASCII character.</td>
</tr>
<tr>
<td>DECIML</td>
<td>ADC #$30</td>
</tr>
<tr>
<td>RTS</td>
<td>Return to caller, bearing the ASCII character corresponding to the hexadecimal value initially in the 4 LSB of the accumulator.</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TVHOME</td>
<td>LDX #0</td>
</tr>
<tr>
<td></td>
<td>Set TV.PTR to the leftmost column of the top row of the screen.</td>
</tr>
<tr>
<td></td>
<td>LDY #0</td>
</tr>
<tr>
<td></td>
<td>JSR TVTOXY</td>
</tr>
<tr>
<td>RTS</td>
<td>Return to caller.</td>
</tr>
</tbody>
</table>
LDA TVROWS    Load A with total rows.
LSR A         Divide it by two.
TAY           Y now holds the number of the central row on the screen.

LDA TVCOLS    Load A with total columns.
LSR A         Divide it by two.
TAX           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      Save it in X...
            PLA      ...and in Y.
            TAY

LDA TV.PTR+1 Get TV.PTR
PHA         and save
LDA TV.PTR  it on
PHA         the stack.

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

RTS         Then return to caller.
**TVPOP**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TV.POP</td>
<td>PLA</td>
</tr>
<tr>
<td></td>
<td>TAX</td>
</tr>
<tr>
<td></td>
<td>PLA</td>
</tr>
<tr>
<td></td>
<td>TAY</td>
</tr>
<tr>
<td></td>
<td>PLA</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TYA</td>
</tr>
<tr>
<td></td>
<td>PHA</td>
</tr>
<tr>
<td></td>
<td>TXA</td>
</tr>
<tr>
<td></td>
<td>PHA</td>
</tr>
<tr>
<td></td>
<td>RTS</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR.TV</td>
<td>JSR TVPUSH</td>
</tr>
<tr>
<td>JSR TVHOME</td>
<td></td>
</tr>
</tbody>
</table>
LDX TVCOLS
LDY TVROWS
JSR CLR.XY
JSR TV.POP
rts

CLR.XY STX COLS
TYA
TAX

CLRROW LDA BLANK
LDY COLS

CLRPOS STA (TV.PTR),Y
DEY
BPL CLRPOS
JSR TVDOWN
DEX
BPL CLRROW
RTS

COLS .BYTE 0

Load X,Y registers with X,Y dimensions of the screen.
Clear X columns, Y rows from current screen location.
Restore zero-page bytes that were changed.
Return to caller, with screen clear and with zero page preserved.
Set the number of columns to be cleared.
Now X holds the number of rows to be cleared.
Load accumulator with your system’s graphic code for a blank.
Load Y with number of columns to be cleared.
Clear a position by writing a blank into it.
Adjust index for next position in the row.
If not done with row, clear next position...
If done with row, move current screen location down by one row.
Done last row yet?
If not, clear next row...
If so, return to caller.
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.
TV.SKIP: Advance to next screen location.
TV.TOXY: 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 Apple and Ohio Scientific computers each feature a machine-language monitor in ROM (read-only memory). The Atari computers feature a machine-language monitor in a plug-in program cartridge. Your system's documentation should tell you how to use the features of your monitor, but let's take a closer look at one
monitor in particular, the Ohio Scientific 65V monitor. Because it is stored in read-only memory in the OSI Challenger I-P, I will refer to it as the OSI ROM monitor.

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.](image)

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 (i.e. increment the displayed address) by depressing the RETURN key. Unfortunately, this convenience is not available in address mode, and neither mode allows you to step backward through memory (i.e. 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 GREATER THAN (> ) and LESS THAN (< ) keys. Touching the GREATER THAN key will move the arrow one field to the right, and depressing the LESS THAN key will move the arrow one field to the left. (If my computer had a cursor pad, I would use the cursor-left and the cursor-right keys to move the arrow from field to field, but it doesn’t have a cursor pad, so GREATER THAN and LESS THAN have to fill the bill. You may assign the field-movement functions to any keys on your system, but GREATER THAN and LESS THAN are reasonable choices, because they look like arrows pointing right and left, respectively.)

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>.WORD 0</td>
<td>This points to the currently selected byte.</td>
</tr>
<tr>
<td>REG.A</td>
<td>.BYTE 0</td>
<td>REG.A holds the image of Register A (the Accumulator).</td>
</tr>
<tr>
<td>REG.X</td>
<td>.BYTE 0</td>
<td>REG.X holds the image of Register X.</td>
</tr>
<tr>
<td>REG.Y</td>
<td>.BYTE 0</td>
<td>REG.Y holds the image of Register Y.</td>
</tr>
<tr>
<td>REG.P</td>
<td>.BYTE 0</td>
<td>REG.P holds the image of the Processor Status Register.</td>
</tr>
<tr>
<td>FIELD</td>
<td>.BYTE 0</td>
<td>FIELD holds the number of the current field.</td>
</tr>
</tbody>
</table>

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.](image-url)
With this flowchart as a guide, we can now write the source code for the top level of the Visible Monitor:

```
| 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**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSR CLRMON</td>
<td>Clear monitor’s portion of screen.</td>
</tr>
<tr>
<td>JSR LINE.1</td>
<td>Display the Label Line.</td>
</tr>
<tr>
<td>JSR LINE.2</td>
<td>Display the Data Line.</td>
</tr>
<tr>
<td>JSR LINE.3</td>
<td>Display the Arrow Line.</td>
</tr>
<tr>
<td>RTS</td>
<td>Return to caller.</td>
</tr>
</tbody>
</table>

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
  LDY #2
  JSR TVTOXY
  LDX #25
  LDY #3
  JSR CLR.XY
  RTS

Set TV.PTR to column 2, row 2 of screen.
We'll clear 25 columns and 3 rows.
Here we clear them.
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

<table>
<thead>
<tr>
<th>LINE.1</th>
<th>LDX #13</th>
<th>X-coordinate of Label &quot;A&quot;.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LDY #2</td>
<td>Y-coordinate of Label &quot;A&quot;.</td>
</tr>
<tr>
<td></td>
<td>JSR TVTOXY</td>
<td>Place TV.PTR at coordinates given by X,Y registers.</td>
</tr>
<tr>
<td></td>
<td>LDY #0</td>
<td>Put labels on the screen:</td>
</tr>
<tr>
<td></td>
<td>STY LBLCOL</td>
<td>Initialize label column counter.</td>
</tr>
<tr>
<td>LBLOOP</td>
<td>LDA LABELS,Y</td>
<td>Get a character and put its graphic on the screen.</td>
</tr>
<tr>
<td></td>
<td>JSR VUCHAR</td>
<td>Prepare for next character.</td>
</tr>
<tr>
<td></td>
<td>INC LBLCOL</td>
<td>Use label column as an index.</td>
</tr>
<tr>
<td></td>
<td>LDY LBLCOL</td>
<td>Done last character?</td>
</tr>
<tr>
<td></td>
<td>CPY #10</td>
<td>If not, do next one.</td>
</tr>
<tr>
<td></td>
<td>BNE LBLOOP</td>
<td>Return to caller.</td>
</tr>
<tr>
<td></td>
<td>RTS</td>
<td>These are the characters to be copied to the screen.</td>
</tr>
<tr>
<td>LABELS</td>
<td>.BYTE 'A X'</td>
<td>This is a counter.</td>
</tr>
<tr>
<td></td>
<td>.BYTE 'Y P'</td>
<td></td>
</tr>
<tr>
<td>LBLCOL</td>
<td>.BYTE 0</td>
<td></td>
</tr>
</tbody>
</table>

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 dis-
play 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDX #2</td>
<td>Load X register with X-coordinate for start of data line.</td>
</tr>
<tr>
<td>LDY #3</td>
<td>Load Y register with Y-coordinate for data line.</td>
</tr>
<tr>
<td>JSR TVTOXY</td>
<td>Set TV.PTR to point to the start of the data line.</td>
</tr>
<tr>
<td>LDA SELECT+1</td>
<td>Display high byte of the currently selected address.</td>
</tr>
<tr>
<td>JSR VUBYTE</td>
<td>Display low byte of the currently selected address.</td>
</tr>
<tr>
<td>JSR TVSKIP</td>
<td>Skip one space after address field.</td>
</tr>
<tr>
<td>JSR GET.SL</td>
<td>Look up value of the currently selected byte.</td>
</tr>
<tr>
<td>PHA</td>
<td>Save it.</td>
</tr>
<tr>
<td>JSR VUBYTE</td>
<td>Display it, in hexadecimal format, in field 1.</td>
</tr>
<tr>
<td>JSR TVSKIP</td>
<td>Skip one space after field 1.</td>
</tr>
<tr>
<td>PLA</td>
<td>Restore value of currently selected byte.</td>
</tr>
<tr>
<td>JSR VUCHAR</td>
<td>Display that byte, in graphic form, in field 2.</td>
</tr>
<tr>
<td>JSR TVSKIP</td>
<td>Skip one space after field 2.</td>
</tr>
<tr>
<td>LDX #0</td>
<td>Display 6502 register images in fields 4 thru 7:</td>
</tr>
<tr>
<td>LDA REGS,X</td>
<td>Look up the register image.</td>
</tr>
<tr>
<td>JSR VUBYTE</td>
<td>Display it in hexadecimal format.</td>
</tr>
<tr>
<td>JSR TVSKIP</td>
<td>Skip one space after hexadecimal field.</td>
</tr>
<tr>
<td>INX</td>
<td>Get ready for next register...</td>
</tr>
<tr>
<td>CPX #4</td>
<td>Done 4 registers yet?</td>
</tr>
<tr>
<td>BNE VUREGS</td>
<td>If not, do next one...</td>
</tr>
<tr>
<td>RTS</td>
<td>If all registers displayed, return.</td>
</tr>
</tbody>
</table>
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
  LDA (SELECT),Y
  RTS

Get the zeroth byte above
the address pointed to by SELECT.
Return to caller.
```

However, SELECT is not a zero-page pointer; it's up in page $12. 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
  STA GETPTR
  LDA SELECT + 1
  STA GETPTR + 1
  LDY #0
  LDA (SELECT),Y
  RTS

Set GETPTR equal to
SELECT: first the low byte;
then the
high byte.
Get the zeroth byte above
the address pointed to by GETPTR.
Return to caller, with A bearing the contents 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 and from X register.
       STA GETPTR +1 Restore contents of current byte from
       TYA 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

LDX #2
LDY #4
JSR TVTOXY
LDY FIELD
SEC
CPY #7
BCC FLD.OK
LDY #0
STY FIELD

FLD.OK
LDA FIELDS,Y
TAY
LDA ARROW
STA (TV.PTR),Y
RTS

FIELDS
.BYTE 3,6,8
.BYTE $0B,$0E
.BYTE $11,$14

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 DISPLAY:

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

THE VISIBLE MONITOR 73
When DISPLAY 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 the display is quick!

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:

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>GREATER THAN</td>
<td>Move arrow one field to the right.</td>
</tr>
<tr>
<td>LESS THAN</td>
<td>Move arrow one field to the left.</td>
</tr>
<tr>
<td>SPACEBAR</td>
<td>Increment address being displayed. (Step forward through memory.)</td>
</tr>
<tr>
<td>RETURN</td>
<td>Decrement address being displayed. (Step backward through memory.)</td>
</tr>
</tbody>
</table>

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 (i.e: 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 CHARACTER FROM KEYBOARD

">" YES

NO

"<" YES

NO

SPACE KEY YES

NO

RETURN KEY YES

NO

"G" YES

NO

IS MONITOR IN CHARACTER MODE

P YES

NO

SAVE CHARACTER ON STACK

IS KEY 0-9 OR A-F

YES

NO

ROLL BINARY EQUIVALENT OF CHARACTER INTO CURRENT FIELD

POP CHARACTER FROM STACK

HERE ADDITIONAL FUNCTIONS MAY BE ADDED WITH CODE TO TEST THE KEYBOARD CHARACTER AND THEN CALL APPROPRIATE SUBROUTINES

RETURN

MOVE ARROW RIGHT BY ONE FIELD

MOVE ARROW LEFT BY ONE FIELD

INCREMENT DISPLAYED ADDRESS

DECREMENT DISPLAYED ADDRESS

CALL DISPLAYED ADDRESS

STORE CHARACTER IN DISPLAYED ADDRESS

POP CHARACTER FROM STACK
Get a Key

First we need a way to get a key from the keyboard. I assume that your system has a read-only memory routine to perform this function. Place the address of that routine (see the appropriate appendix for your system) into a pointer called ROMKEY located at address $1008. 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 #">" Is it the GREATER THAN key?
         BNE IF.LSR If not, perform next test.
NEXT.F   INC FIELD If so, select the next field.
         LDA FIELD If arrow was at the right-most field, place it underneath the left-most field.
         CMP #7
         BNE EXIT.1
         LDA #0
         STA FIELD
EXIT.1   RTS Then return.
         CMP #"< Is it the LESS THAN 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 current field. If arrow was at left-most field, place it under right-most field.
         LDA #6
         STA FIELD
      RTS Then return.
         CMP #$SPACE Is it the space bar?
         BNE IF.CR If not, perform next test.
INC.SL   INC SELECT If so, step forward through memory, by incrementing the pointer that specifies the displayed address.
         BNE EXIT.3
         INC SELECT +1
EXIT.3   RTS Then return.
         CMP #$CR Is it carriage return?
         BNE IF.CHR 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
      RTS  address to be displayed.
NEXT.1  DEC SELECT  Then return.
      RTS
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
      LDA TV.PTR  the selected address.
      PHA  Save zero-page pointer TV.PTR
      LDX TV.PTR+1  on stack and in X before we
                     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
      LDX #0  the selected address.
      STA (TV.PTR),Y  Store it in the
      STX TV.PTR+1  selected address.
      PLA  Restore TV.PTR to
      STA TV.PTR  its original value.
      RTS  Return to caller, with character origi-
      RTS  nally in A now in the selected address
      CMP #G  and with zero page unchanged.
      BNE IF.HEX  Then return.
      GO  Is it 'G' for GO?
      LDY REG.Y  If not, perform next test.
      LDX REG.X  If so, load the 6502 registers
      LDA REG,P  with their displayed images.
      PHA
      LDA REG.A
      PLA
      JSR CALLSL  Call the subroutine at the selected ad-
      PHP  dress.
      STA REG.A  When subroutine returns,
      STX REG.X  save register values in register
                      images.
THE VISIBLE MONITOR 77
STY REG.Y
PLA
STA REG.P
RTS
CALLSL
JMP (SELECT)

Then return to caller.

CALLSL
JMP (SELECT)

Call the subroutine at the selected address.

CALLSL
JMP (SELECT)

IF.HEX
PHA
JSR BINARY

Save keyboard character.

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
BNE NOTADR

Roll A into a hexadecimal field.

Is arrow underneath the address field (field 0)? If not, the arrow must be under another hexadecimal field.

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.

ADRFLD
LDX #3
CLC
ASL SELECT
ROL SELECT +1
DEX
BPL LOOP.1
TYA
ORA SELECT
STA SELECT
RTS

NOTADR
CPX #1
BNE REGFLD

Then return.

Is arrow underneath field 1?

If not, it must be underneath a register image.

ROL.SL
AND #$0F
PHA
JSR GET.SL
ASL A
ASL A
ASL A
ASL A
AND #$F0
STA TEMP

Roll A's 4 LSB into contents of currently selected byte.

Get the contents of the selected address and shift left 4 times.

Save it in a temporary variable.
PLA
ORA TEMP
JSR PUT.SL
RTS

Get original A's 4 LSB and
OR them with shifted contents of
selected address.
Store the result in the selected
address and return.

TEMP
.BYTE 0

This byte holds the temporary variable
used by ROL.SL.

REGFLD
DEX
DEX
DEX
LDY #3

The arrow must be underneath a
register image — field 3, 4, 5, or 6.

LOOP.2
CLC
ASL REGS,X
DEY
BPL LOOP.2
ORA REGS,X
STA REGS,X
RTS

Roll accumulator’s hexadecimal digit
into appropriate register image...

OTHER
PLA

...Then return.

Restore the raw keyboard character that
we saved on the stack.

CMP#Q
BNE NOT.Q
PLA
PLA
PLP
RTS

Is it ‘Q’ for Quit?
If not, perform next test.
If so, return to
the caller of

NOT.Q
JSR DUMMY

VISMON.
Replace this call to DUMMY with a call
to any other subroutine that extends the
functionality of the Visible Monitor.
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:
<table>
<thead>
<tr>
<th>If accumulator holds</th>
<th>BINARY will return</th>
</tr>
</thead>
<tbody>
<tr>
<td>$30 (ASCII &quot;0&quot;)</td>
<td>$00</td>
</tr>
<tr>
<td>$31 (ASCII &quot;1&quot;)</td>
<td>$01</td>
</tr>
<tr>
<td>$32 (ASCII &quot;2&quot;)</td>
<td>$02</td>
</tr>
<tr>
<td>$33 (ASCII &quot;3&quot;)</td>
<td>$03</td>
</tr>
<tr>
<td>$34 (ASCII &quot;4&quot;)</td>
<td>$04</td>
</tr>
<tr>
<td>$35 (ASCII &quot;5&quot;)</td>
<td>$05</td>
</tr>
<tr>
<td>$36 (ASCII &quot;6&quot;)</td>
<td>$06</td>
</tr>
<tr>
<td>$37 (ASCII &quot;7&quot;)</td>
<td>$07</td>
</tr>
<tr>
<td>$38 (ASCII &quot;8&quot;)</td>
<td>$08</td>
</tr>
<tr>
<td>$39 (ASCII &quot;9&quot;)</td>
<td>$09</td>
</tr>
<tr>
<td>$41 (ASCII &quot;A&quot;)</td>
<td>$0A</td>
</tr>
<tr>
<td>$42 (ASCII &quot;B&quot;)</td>
<td>$0B</td>
</tr>
<tr>
<td>$43 (ASCII &quot;C&quot;)</td>
<td>$0C</td>
</tr>
<tr>
<td>$44 (ASCII &quot;D&quot;)</td>
<td>$0D</td>
</tr>
<tr>
<td>$45 (ASCII &quot;E&quot;)</td>
<td>$0E</td>
</tr>
<tr>
<td>$46 (ASCII &quot;F&quot;)</td>
<td>$0F</td>
</tr>
<tr>
<td>Any other value</td>
<td>$FF</td>
</tr>
</tbody>
</table>

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
LDA BINTAB,Y
RTS

Use ASCII character as an index.
Look up entry in BINARY TABLe.
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 $36$ 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
        SBC #$30
        BCC BAD
        CMP #$0A
        BCC GOOD

        SBC #7
        CMP #$10
        BCS GOOD

BAD    LDA #$FF
       RTS
GOOD   LDX #0
       RTS
```

Prepare to subtract.
Subtract $30$ from character.
If character was originally less than $30$, it was bad, so return $FF$.
Was character in the range $30$ thru $39$?
If so, it was a good input, and we’ve already converted it to binary by subtracting $30$, so we’ll return now with the character’s binary equivalent in the accumulator.
Subtract 7.
Was character originally in the range $41$ thru $46$?
If so, it was a good input, and we’ve already converted it to binary by subtracting $37$, so we’ll return now with the character’s binary equivalent in the accumulator.
Indicate a bad input by returning minus, with $A$ holding $FF$.
Indicate a good input by returning plus, with $A$ holding the character’s binary equivalent.
Visible Monitor Utilities

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

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>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).</td>
</tr>
<tr>
<td>CALLSL</td>
<td>Call the currently selected address as a subroutine.</td>
</tr>
<tr>
<td>DEC.SL</td>
<td>Select previous address, by decrementing SELECT pointer.</td>
</tr>
<tr>
<td>GETKEY</td>
<td>Get a character from the keyboard by calling machine's read-only memory routine indirectly.</td>
</tr>
<tr>
<td>GET.SL</td>
<td>Get byte at currently selected address.</td>
</tr>
<tr>
<td>GO</td>
<td>Load registers from displayed images and call displayed address. Upon return, restore register images from registers.</td>
</tr>
<tr>
<td>INC.SL</td>
<td>Select next byte (increment SELECT pointer).</td>
</tr>
<tr>
<td>PUT.SL</td>
<td>Store accumulator at currently selected address.</td>
</tr>
<tr>
<td>VISMON</td>
<td>Let user give the Visible Monitor commands until user presses 'Q' to quit.</td>
</tr>
</tbody>
</table>

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

![Visible Monitor Hierarchy Diagram](image)

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

Use the minimal machine-language monitor on your computer to enter the Visible Monitor into memory; then have your monitor pass control to the Visible Monitor. The Visible Monitor display should appear in the upper portion of your video display. If it's not fully visible, adjust the value HOME in the screen parameters (HOME is the pointer at $1000). Use the GREATER THAN and LESS THAN character keys to move the arrow from field to field. Place the arrow under field 0 and roll hexadecimal characters into the address. Select an address in the lower portion of screen memory and use the Visible Monitor to place characters on the screen. Enter characters to the screen using both field 1 (the hexadecimal data field), and field 2 (the character field).

Select the address of the TVT routine in your system. Press G to call that subroutine. You should see the character in the accumulator print on the screen. Try exploring other memory locations. Try writing to a read-only memory address. Why doesn’t that work? Try writing to the upper portion of the screen. Why doesn’t that work?
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 Apple, Atari, OSI and PET computers each feature a routine to print a character on the screen, thus simulating a TVT (TeleVision Typewriter), 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:

\[
\text{JSR TVT.ON}
\]

This line will deselect the TVT:

\[
\text{JSR TVTOFF}
\]

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

<table>
<thead>
<tr>
<th>Flag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td>0</td>
</tr>
<tr>
<td>ON</td>
<td>$FF</td>
</tr>
<tr>
<td>TVT</td>
<td>.BYTE ON</td>
</tr>
</tbody>
</table>

- When a device flag = zero, that device is not selected.
- When a device flag = $FF, that device is selected.
- This flag is zero if TVT is not selected; nonzero otherwise. Initially, the TVT is selected.
<table>
<thead>
<tr>
<th></th>
<th>.BYTE OFF</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PRINTR</strong></td>
<td></td>
<td>This flag is zero if the PRINTR is not selected; nonzero otherwise. Initially, the printer is not selected.</td>
</tr>
<tr>
<td><strong>USER</strong></td>
<td></td>
<td>This flag is zero if the user-provided output subroutine is not selected; nonzero otherwise. Initially, the user-provided function is deselected.</td>
</tr>
</tbody>
</table>

**Select and Deselect Subroutines**

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TVT.ON</strong></td>
<td>LDA #ON STA TVT RTS</td>
<td>Select TVT as an output device by setting the flag that indicates the “select” state of the TVT.</td>
</tr>
<tr>
<td><strong>TVTOFF</strong></td>
<td>LDA #OFF STA TVT RTS</td>
<td>Deselect TVT as an output device by clearing the flag that indicates the “select” state of the TVT.</td>
</tr>
<tr>
<td><strong>PR.ON</strong></td>
<td>LDA #ON STA PRINTR RTS</td>
<td>Select printer as an output device by setting the flag that indicates the “select” state of the printer.</td>
</tr>
<tr>
<td><strong>PR.OFF</strong></td>
<td>LDA #OFF STA PRINTR RTS</td>
<td>Deselect printer as an output device by clearing the flag that indicates the “select” state of the printer.</td>
</tr>
<tr>
<td><strong>USR.ON</strong></td>
<td>LDA #ON STA USER RTS</td>
<td>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.</td>
</tr>
<tr>
<td><strong>USROFF</strong></td>
<td>LDA #OFF STA USER RTS</td>
<td>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.</td>
</tr>
<tr>
<td><strong>ALL.ON</strong></td>
<td>JSR TVT.ON JSR PR.ON JSR USR.ON RTS</td>
<td>Select all output devices by selecting each output device individually.</td>
</tr>
<tr>
<td><strong>ALLOFF</strong></td>
<td>JSR TVTOFF JSR PR.OFF JSR USROFF RTS</td>
<td>Deselect all output devices by deselecting each output device individually.</td>
</tr>
</tbody>
</table>
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.CHAR, because it will PRINT a CHARACTER.

All the software in this book that outputs characters will do so by calling PR.CHAR; 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.CHAR. 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.CHAR, means we can improve our character output in the future without difficulty.

When it is called, PR.CHAR 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.CHAR can return. Figure 7.1 is a flowchart for PR.CHAR.

Figure 7.1: To print a character to all currently selected output devices (PR.CHAR, 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, B2, B3, or B4 for your computer.

The source code of the PR.CHR routine follows:

```
PR.CHR

PR.CHR       STA CHAR
BEQ EXIT
LDA TVT     BEQ IF.PR
LDA CHAR    JSR SEND.1
           LDA PRINTR
           BEQ IF.USR
           LDA CHAR
           JSR SEND.2
           LDA USER
           BEQ EXIT
           LDA CHAR
           JSR SEND.3
           RTS
           .BYTE 0
```

Save the character.
If it's a null, return without printing it.
Is TVT selected?
If not, test next device.
If so, send character indirectly to
system's TVT output routine.
Is printer selected?
If not, test next device.
If so, send character indirectly
to system's printer driver.
Is user-written output subroutine
selected?
If not, test next device.
If so, send character indirectly
to user-written output subroutine.
Return to caller.
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
JSR PR.CHR
LDA #LF
JSR PR.CHR
RTS

Send a carriage return and a line feed to the currently selected device(s).

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-
LSR A
LSR A
LSR A
JSR ASCII
JSR PR.CHAR
PLA
JSR ASCII
JSR PR. CHR
RTS

significant bits) in the byte:

Print that ASCII character to the current device(s).
Determine ASCII for the 4 LSB (least-significant bits) in the byte that was passed to this subroutine.
Print that ASCII character to the current device(s).
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**

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

**Print X New Lines**

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

**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 End of TeXt. And for reasons which will become clear in Chapter 10, A Disassembler, we'll also start each message with another special character: TEX, for TeXt 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 byte 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
LDA #MSG.LO
STA MSGPTR
LDA #MSG.HI
STA MSGPTR +1
LDX #MSGPTR
JSR PR.MSG
```

Select TVT as an output device. (Any other currently selected output device will echo the screen output.)
Set MSGPTR
so it points
to the start
of the message.
Set X register so it points to MSGPTR.
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
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 so it points to the next character in the message.
BNE NEXT
INC 1,X
NEXT   JSR PR.CHK  Send the character to all currently selected output devices.
CLC  Get next character from message.
BCC LOOP
MSGEND  PLA
STA 0,X  Restore message pointer.
PLA
STA 1,X
RTS  Return to caller, with MSGPTR preserved.
TEMP.X  .BYTE 0  This data cell is used to preserve the initial 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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLA</td>
<td>Pull return address from stack and save it in registers X and Y.</td>
</tr>
<tr>
<td>TAX</td>
<td>Save the select pointer, because we're going to use it as a text pointer.</td>
</tr>
<tr>
<td>PLA</td>
<td>Set SELECT = return address.</td>
</tr>
<tr>
<td>TAY</td>
<td>Increment SELECT pointer so it points to TEX character.</td>
</tr>
<tr>
<td>JSR PUSHSL</td>
<td>Increment select pointer so it points to the next character in the message.</td>
</tr>
<tr>
<td>STX SELECT</td>
<td>Get character.</td>
</tr>
<tr>
<td>STY SELECT +1</td>
<td>Is it end of message indicator?</td>
</tr>
<tr>
<td>JSR INC.SL</td>
<td>If so, adjust return address and return.</td>
</tr>
<tr>
<td>JSR INC.SL</td>
<td>If not, print the character to all currently selected devices.</td>
</tr>
<tr>
<td>JSR GET.SL</td>
<td>Then loop to get next character...</td>
</tr>
<tr>
<td>CMP #$ETX</td>
<td></td>
</tr>
<tr>
<td>BEQ ENdit</td>
<td></td>
</tr>
<tr>
<td>JSR PR.CHR</td>
<td></td>
</tr>
<tr>
<td>CLC</td>
<td></td>
</tr>
<tr>
<td>BCC LOOP</td>
<td></td>
</tr>
<tr>
<td>LDX SELECT</td>
<td></td>
</tr>
<tr>
<td>LDY SELECT +1</td>
<td></td>
</tr>
</tbody>
</table>

ENDIT
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
STA RETURN
PLA
STA RETURN+1
LDA SELECT+1
PHA
LDA SELECT
PHA
LDA RETURN+1
PHA
LDA RETURN
PHA
RTS

Pull return address from stack and store it temporarily in RETURN.
Push select pointer onto stack.
Push return address back onto stack.
Return to caller. (Caller will find select pointer on top of the stack.)
**Restore Select Pointer**  
*(Preserving X,Y, and the Zero Page)*

<table>
<thead>
<tr>
<th>POP.SL</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PLA</td>
<td>STA RETURN</td>
<td>PLA</td>
<td></td>
</tr>
<tr>
<td>STA RETURN+1</td>
<td>PLA</td>
<td>STA SELECT</td>
<td>PLA</td>
</tr>
<tr>
<td>STA SELECT+1</td>
<td>LDA RETURN+1</td>
<td>PHA</td>
<td></td>
</tr>
<tr>
<td>LDA RETURN</td>
<td>PHA</td>
<td>RTS</td>
<td>.WORD 0</td>
</tr>
</tbody>
</table>

Save return address temporarily.

Restore select pointer from stack.

Place return address back on stack.

Return to caller.

This pointer is used by PUSHSL and POP.SL to preserve their return addresses.

**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 a dump that will be easily readable, even on the narrow confines of a twenty-five- 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 screen lines. See outputs A and B in figure 8.1.

Output A:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0200</td>
<td>HH HH HH HH HH HH HH HH HH</td>
</tr>
<tr>
<td>0208</td>
<td>HH HH HH HH HH HH HH HH HH</td>
</tr>
<tr>
<td>0210</td>
<td>HH HH HH HH HH HH HH HH HH</td>
</tr>
<tr>
<td>0218</td>
<td>HH HH HH HH HH HH HH HH HH</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>29 columns</td>
</tr>
</tbody>
</table>

Output B:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0200</td>
<td>HH HH HH HH HH HH HH HH</td>
</tr>
<tr>
<td>0208</td>
<td>HH HH HH HH HH HH HH HH</td>
</tr>
<tr>
<td>0210</td>
<td>HH HH HH HH HH HH HH HH</td>
</tr>
<tr>
<td>0218</td>
<td>HH HH HH HH HH HH HH HH</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>23 columns</td>
</tr>
</tbody>
</table>

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:

![Flowchart of the screen Hexdump Program.](image)

**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.
HEXLNS  .BYTE 4  Number of hexadecimal lines to be dumped by TVDUMP. (Set this to any number you like. To dump a single hexadecimal line [16 bytes], set HEXLNS = 1.)

TVDUMP

TVDUMP  JSR TVT.ON  Select TVT as an output device.
          (Other devices will echo the dump.)
          LDA HEXLNS  Set COUNTR to the number of lines to be dumped by TVDUMP.
          STA COUNTR  Set SELECT to beginning of a screen line (8 bytes)
          LDA SELECT  by zeroing 3 LSB in SELECT.
          AND #$F8    Skip two lines on the screen.
          STA SELECT
          LDX #2      
          JSR CR.LFS

DUMPLN  JSR PR.ADR  Print the selected address.
          JSR CR.LF   Advance to a new line on the screen.
          (This call to CR.LF may be replaced with a call to SPACE on systems with screens more than 27 columns wide.
          This will yield the Output A rather than
DMPBYT
  JSR SPACE
  JSR DUMPSSL
  JSR INC.SL
  LDA SELECT
  AND #07
  BNE DMPBYT
  JSR CR.LF
  LDA SELECT
  AND #$0F
  BNE IFDONE
  JSR CR.LF
  IFDONE
  DEC COUNTR
  BNE DUMPLN
  JSR TVTOFF
  RTS

Output B.)
Print a space.
Dump currently selected byte.
Select next address by incrementing select pointer.
Is it the beginning of a new screen line? (3 LSB = 0?)
If not, dump next byte...
If so, advance to a new line on the screen.
Does this address mark the beginning of a new hexadecimal line?
(4 LSB of SELECT = 0?)
If so, skip a line on the screen.
Dumped last line yet?
If not, dump next line.
Deselect TVT as an output device.
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.

DUMPSSL
  JSR GET.SL
  JSR PR.BYT
  RTS

Get currently selected byte.
Print it in hexadecimal format.
Return to caller.

PRINT ADDRESS

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

PR.ADR
  LDA SELECT+1
  JSR PR.BYT
  LDA SELECT
  JSR PR.BYT
  RTS

Get the high byte of SELECT...
...and print it in hexadecimal format.
Get the low byte of SELECT...
...and print it in hexadecimal format.
Then return to caller.
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 output that is easily read within the twenty-five 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 (e.g. “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:

![Flowchart](image)

**Figure 8.3:** To print a Hexdump.
| PRDUMP | JSR TITLE | Display the title. |
| PRDUMP | JSR SETADS | Let user set start address and end address 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 an 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. |
| TITLE | JSR TVT.ON | Select screen as an output device. |
| TITLE | 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    |             | ‘SET STARTING ADDRESS ’
| .BYTE    |             | ‘AND PRESS “Q”.’
| .BYTE    |             | Call the Visible Monitor, so user can
| .BYTE    |             | specify a given address.
| .BYTE    |             | Set starting address equal to address set
| .BYTE ETX|             | by the user.
| JSR VISMON |             | Calculate the difference between
| JSR SAHERE |             | the starting and ending addresses.

| SET.EA | JSR PRINT: | Put prompt on the screen:
|        | .BYTE    | ‘SET ENDING ADDRESS ’
|        | .BYTE    | ‘AND PRESS “Q”.’
|        | .BYTE ETX| Call the Visible Monitor, so user can
|        | JSR VISMON| 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.

| EAHERE | LDA SELECT +1 | Set EA=SELECT...
|        | STA EA+1      | ... and return.
|        | LDA SELECT   | Set SA=SELECT...
|        | STA SA+1      |
LDA SELECT
STA SA
RTS

TOOLOW
JSR PRINT:
.BYTE STX,
.BYTE CR,LF,LR
.BYTE
.BYTE
.BYTE
.BYTE ETX
JSR PR.SA
JMP SET.EA

SA
.WORD 0

EA
.WORD $FFFFFF

...and return.

Since user set ending address too low, print error message:

'ERROR! '

'END ADDRESS LESS '

'THAN START ADDRESS, '

'WHICH IS '

Print starting address. ...and let the user set

the ending address again.

Pointer to starting address of memory to be dumped.

Pointer to ending address of memory to be dumped.

Now that the user can set the starting address and the ending address for a hex-dump (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 #$
JSR PR.CHAR
LDA SA + 1
JSR PR.BYTE
LDA SA
JSR PR.BYTE
RTS

Print a dollar sign to indicate hexadecimal.

Print high byte of starting address.

Print low byte of starting address.

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.CHAR 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.CHAR
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 pro-
duced 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 hexa-
decimal 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
JSR SPACES
LDA #0
STA COLUMN
LDA COLUMN
JSR ASCII
JSR PR.CHAR
LDX #2
JSR SPACES
INC COLUMN
LDA COLUMN
AND #$F0
BEQ HXLOOP
LDX #2
JSR CR.LFS
RTS

PRINT HEXADECIMAL DIGITS (Version 1)
```

- **Print seven spaces.**
- **Initialize column counter**
- **Convert column counter to**
- **ASCII character and**
- **print it.**
- **Space twice after the character.**
- **Increment the column counter.**
- **Loop if counter not greater**
- **than $0F.**
- **Otherwise, skip two lines**
- **after the header.**
- **Then return.**
- **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

---

108 BEYOND GAMES
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 '
.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 20 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 listings 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 cur-
rently 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

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

### 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**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEXTSL</td>
<td>SEC Prepare to compare.</td>
</tr>
<tr>
<td></td>
<td>LDA SELECT +1 Is high byte of SELECT less than high byte of end address (EA)?</td>
</tr>
<tr>
<td></td>
<td>CMP EA +1 If so, SELECT is less than EA, so it may be incremented.</td>
</tr>
<tr>
<td></td>
<td>BCC SL.OK If SELECT is greater than EA, don’t increment SELECT.</td>
</tr>
<tr>
<td></td>
<td>BNE NO.INC SELECT is in the same page as EA, prepare to compare low bytes:</td>
</tr>
<tr>
<td></td>
<td>SEC</td>
</tr>
<tr>
<td></td>
<td>LDA SELECT Is low byte of SELECT less than low byte of EA?</td>
</tr>
<tr>
<td></td>
<td>CMP EA</td>
</tr>
<tr>
<td></td>
<td>BCS NO.INC If not, don’t increment it.</td>
</tr>
<tr>
<td></td>
<td>SL.OK JSR INC.SL Since SELECT is less than EA, we may increment it.</td>
</tr>
<tr>
<td></td>
<td>LDA #0 Set “incremented” return code and return.</td>
</tr>
<tr>
<td></td>
<td>RTS</td>
</tr>
<tr>
<td></td>
<td>NO.INC LDA #$FF Set “not incremented” return code and return.</td>
</tr>
<tr>
<td></td>
<td>RTS</td>
</tr>
</tbody>
</table>

**Go to Start of Block**

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOTOSA</td>
<td>LDA SA Set SELECT equal to START ADDRESS of block.</td>
</tr>
<tr>
<td></td>
<td>STA SELECT</td>
</tr>
<tr>
<td></td>
<td>LDA SA +1</td>
</tr>
<tr>
<td></td>
<td>STA SELECT +1</td>
</tr>
<tr>
<td></td>
<td>RTS</td>
</tr>
</tbody>
</table>

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. Prob-
ably 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 (e.g., ‘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 0000 has the label "TV.PTR," but it can let you know that a given instruction operates on address 0000.

A disassembled line includes the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>1.</td>
<td>Mnemonic.</td>
</tr>
<tr>
<td>2.</td>
<td>Operand.</td>
</tr>
<tr>
<td>3.</td>
<td>Address of opcode.</td>
</tr>
<tr>
<td>4.</td>
<td>Opcode in hexadecimal.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>OP</th>
<th>Address</th>
<th>Value1</th>
<th>Value2</th>
<th>Value3</th>
<th>Value4</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSR</td>
<td>0400</td>
<td>1E00</td>
<td>20</td>
<td>00</td>
<td>04</td>
</tr>
<tr>
<td>JSR</td>
<td>04A0</td>
<td>1E03</td>
<td>20</td>
<td>A0</td>
<td>04</td>
</tr>
<tr>
<td>LDA</td>
<td>(0021),Y</td>
<td>1E06</td>
<td>B1</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>CLC</td>
<td></td>
<td>1E08</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCC</td>
<td>1E00</td>
<td>1E09</td>
<td>90</td>
<td>F5</td>
<td></td>
</tr>
</tbody>
</table>

HEXDUMP

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1E00</td>
<td>20</td>
<td>00</td>
<td>04</td>
<td>20</td>
<td>A0</td>
<td>04</td>
<td>B1</td>
<td>21</td>
<td>18</td>
<td>90</td>
<td>F5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

TO DISASSEMBLE ONE LINE:

1. GET OPCODE
2. WRITE DOWN ITS MNEMONIC
3. LOOK UP ITS ADDRESSING MODE
4. WRITE DOWN ITS OPERAND
5. FINISH THE LINE BY WRITING, IN HEX, THE BYTE(S) WE JUST DISASSEMBLED

RETURN

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**

DSLINESL 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:

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

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**

<table>
<thead>
<tr>
<th>OPERND</th>
<th>TAX</th>
<th>Look up addressing mode code for this opcode.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA MODES,X</td>
<td>TAX</td>
<td>X now indicates the addressing mode.</td>
</tr>
<tr>
<td>JSR MODE.X</td>
<td>RTS</td>
<td>Call the subroutine that handles addressing mode “X.”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Return to caller.</td>
</tr>
</tbody>
</table>

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”**

<table>
<thead>
<tr>
<th>MODE.X</th>
<th>LDA SUBS,X</th>
<th>Get low byte of Xth pointer in the table of subroutine pointers.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>STA SUBPTR</td>
<td>Set low byte of subroutine pointer.</td>
</tr>
<tr>
<td></td>
<td>INX</td>
<td>Adjust index to get next byte.</td>
</tr>
<tr>
<td></td>
<td>LDA SUBS,X</td>
<td>Get high byte of Xth pointer in the table of subroutine pointers.</td>
</tr>
<tr>
<td></td>
<td>STA SUBPTR+1</td>
<td>Set high byte of subroutine pointer.</td>
</tr>
<tr>
<td></td>
<td>JMP (SUBPTR)</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 give 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 DUMPSEL      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 DUMPSEL      Print it in hexadecimal format.
            PLA             Restore the operand’s first byte to the
            JSR PR.BYT       accumulator, and print it in hexa-
            RTS            decimal.

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
            BNE SENDIT       right parenthesis.
            Send it to all currently selected devices.
LPAREN       LDA #'('                  Load accumulator with ASCII code for
            Send it to all currently selected devices.
SENDIT       JSR PR.CHCHR          Return to caller.
Index with Register X: XINDEX

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

XINDEX      LDA #',
JSR PR.CHR  LDA #X
JSR PR.CHR  RTS

Load accumulator with ASCII code for a comma; then print it to all currently selected devices.
Load accumulator with ASCII code for the letter "X;" then print it to all currently selected devices.
Return to caller.

Index with Register Y: YINDEX

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

YINDEX      LDA #',
JSR PR.CHR  LDA #Y
JSR PR.CHR  RTS

Load accumulator with ASCII code for a comma; then print it to all currently selected devices.
Load accumulator with ASCII code for the letter "Y;" then print it to all currently selected devices.
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:

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Addressing Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSLUT</td>
<td>Absolute</td>
</tr>
</tbody>
</table>
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 eight characters in its operand, if leading zeros are included.

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 TOWOBYT
LDX #2 X holds number of bytes in operand.
LDA #4 A holds number of characters in operand.
RTS

A TABLE-DRIVEN DISASSEMBLER 123
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.

Absolute, 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
JSR PR.CHR
LDX #0
LDA #1
RTS

Load accumulator with ASCII code for the letter A.
Print it on all currently selected devices.
X holds number of bytes in operand.
A holds number of characters in operand.
Return to caller.

Implied Mode: IMPLID

Implied mode has no operand, so just return:

CLC 18

IMPLID
LDX #0
LDA #0
RTS

X holds number of bytes in operand.
A holds number of characters in operand.

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 #’#
JSR PR.CHR
LDA #$
JSR PR.CHR
JSR ONEBYTE
LDX #1
LDA #4
RTS

Print a ‘#’ sign.
Print a dollar sign.
Print 1-byte operand in hexadecimal format.
X holds number of bytes in operand.
A holds number of characters in operand.
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

<table>
<thead>
<tr>
<th>INDRCT</th>
<th>JSR LPAREN</th>
<th>Print left parenthesis.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JSR ABSLUT</td>
<td>Print the 2-byte operand.</td>
</tr>
<tr>
<td></td>
<td>JSR RPAREN</td>
<td>Print the right parenthesis.</td>
</tr>
<tr>
<td></td>
<td>LDX #2</td>
<td>X holds number of bytes in operand.</td>
</tr>
<tr>
<td></td>
<td>LDA #6</td>
<td>A holds number of characters in operand.</td>
</tr>
<tr>
<td></td>
<td>RTS</td>
<td>Return to caller.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>IND.X</th>
<th>JSR LPAREN</th>
<th>Print a left parenthesis.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JSR ZERO.X</td>
<td>Print a zero-page address, a comma, and the letter “X.”</td>
</tr>
<tr>
<td></td>
<td>JSR RPAREN</td>
<td>Print a right parenthesis.</td>
</tr>
<tr>
<td></td>
<td>LDX #1</td>
<td>X holds number of bytes in operand.</td>
</tr>
<tr>
<td></td>
<td>LDA #8</td>
<td>A holds number of characters in operand.</td>
</tr>
<tr>
<td></td>
<td>RTS</td>
<td>Return to caller.</td>
</tr>
</tbody>
</table>
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 #8 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 of the opcode following the branch instruction.

ADC SELECT
BCC RELEND
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 Restore SELECT pointer.
JSR POP.SL
LDX #1
LDA #4

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 could simply print a 1-byte operand. But I find listings more readable when all zero-page addresses are shown with the leading zeros (eg: “00FE” rather than “FE” to represent address $00FE). Therefore, let's print all zero-page operands with a leading zero. That simply requires us to print two ASCII zeroes and then to print the 1-byte operand. This will cause the bytes “85 2A” to be disassembled as:

STA 002A 85 2A

ZEROPG LDA #0 Print two ASCII zeroes to all currently selected devices.
JSR PR.BYT Print the 1-byte operand.
JSR ONEBYT
LDX #1
LDA #4

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 006C,X  B5 6C

and “B6 53” will disassemble as:

LDX 0053,Y  B6 53

<table>
<thead>
<tr>
<th>ZERO.X</th>
<th>JSR ZEROPG</th>
<th>JSR XINDEX</th>
<th>LDX #1</th>
<th>LDA #6</th>
<th>RTS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Print the zero-page address.</td>
<td>Print a comma and the letter “X.”</td>
<td>X holds number of bytes in operand.</td>
<td>A holds number of characters in operand.</td>
<td>Return to caller.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ZERO.Y</th>
<th>JSR ZEROPG</th>
<th>JSR YINDEX</th>
<th>LDX #1</th>
<th>LDA #6</th>
<th>RTS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Print the zero-page address.</td>
<td>Print a comma and the letter “Y.”</td>
<td>X holds number of bytes in operand.</td>
<td>A holds number of characters in operand.</td>
<td>Return to caller.</td>
</tr>
</tbody>
</table>

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

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

| SUBS     | .WORD ABSLUT     |
| SUBS     | .WORD ABS.X      |
| SUBS     | .WORD ABS.Y      |
| SUBS     | .WORD ACC        |
| SUBS     | .WORD IMPLID     |
| SUBS     | .WORD IMMEDT     |
| SUBS     | .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.

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FINISH</td>
<td>STA OPCHRS  Save the length of the operand, in characters and in bytes.</td>
</tr>
<tr>
<td></td>
<td>STX OPBYTS  If necessary, decrement the SELECT pointer so it points to the opcode.</td>
</tr>
<tr>
<td></td>
<td>DEX</td>
</tr>
<tr>
<td></td>
<td>BMI SEL.OK</td>
</tr>
<tr>
<td>LOOP.1</td>
<td>JSR DEC.SL</td>
</tr>
<tr>
<td></td>
<td>DEX</td>
</tr>
<tr>
<td></td>
<td>BPL LOOP.1</td>
</tr>
<tr>
<td>SEL.OK</td>
<td>SEC</td>
</tr>
<tr>
<td></td>
<td>LDA ADRCOL</td>
</tr>
<tr>
<td></td>
<td>SBC #4</td>
</tr>
<tr>
<td></td>
<td>SBC OPCHRS</td>
</tr>
<tr>
<td></td>
<td>TAX</td>
</tr>
<tr>
<td></td>
<td>JSR SPACES</td>
</tr>
<tr>
<td>LOOP.2</td>
<td>JSR PR.ADR</td>
</tr>
<tr>
<td></td>
<td>JSR SPACE</td>
</tr>
<tr>
<td></td>
<td>JSR DUMPSL</td>
</tr>
<tr>
<td></td>
<td>JSR INC.SL</td>
</tr>
</tbody>
</table>

A TABLE-DRIVEN DISASSEMBLER 131
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 disas- 
semble 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 “PRINT- 
ING 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     |                  |
|             | .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
MOVNUM address (EA), and its destination address (DEST).

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.](image)

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
PUTPTR = GETPTR + 2

This is the input-page pointer.
This is the output-page pointer.
Set NUM = EA - SA

MOV.EA
SEC
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
RTS

MOVNUM
LDY #3
LDA GETPTR, Y
PHA
DEY
BPL SAVE
SEC
LDA SA + 1
CMP DEST + 1
BCC MOVEUP
BNE MOVEDN
LDA SA

Is DEST less than START?

If so, we'll move down.
If not, we'll move up.
SA, destination are in the same page.
If SA more than destination, we'll move down. If SA less than destination,
we'll move up. If they are equal, we'll return bearing okay code.

OK.RTN
LDY #0
PLA
STA GETPTR, Y
INY
CMP #4
BNE RESTOR
RTS

Restore 4 zero-page bytes that were used by the move code.

RESTOR

Restored last byte yet?
If not, restore next one. If so, return, with move complete and zero page preserved.
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
BEQ LESSUP

More than one page to move?
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
LDA NUM
SEC
SBC #$FF.

Now Y is high byte of block size.
Now A is low byte of block size.
Prepare to subtract.
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
ADC SA
STA GETPTR
BCC NEXT.2
INY

NEXT.2
TYA
ADC SA+1
STA GETPTR+1

Prepare to add.

TXA
CLC
ADC DEST
STA PUTPTR
BCC NEXT.3
INC PUTPTR+1

NEXT.3
LDA PUTPTR+1
ADC DEST+1
STA PUTPTR+1

Now GETPTR = SA + NUM - $FF
(the last page in the origin block).

Prepare to add.

Now PUTPTR = DEST + NUM - $FF
(the last page in the destination block).
Now the page pointers (GETPTR and
PUTPTR) point to the last page in, respec-
tively, the origin and destination blocks.
PAGEUP
LDX NUM+1 Load X with number of pages to move.
LDY #$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
DEC PUTPTR+1
DEX Still more than a page to move?
BNE PAGEUP If so, move up another page.
LED R PAGEUP
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
CPY #$FF About to move last byte?
BNE SOMEUP If not, move another.
JMP OK.RTN If so, return bearing "OK" code.
LOPAGE
LDA SA Set page pointers to the bottom of the origin and destination blocks.
STA GETPTR
LDA SA+1
STA GETPTR+1
LDA DEST
LDA DEST+1
STA PUTPTR
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
LDY #0
LDX NUM+1
BEQ LESSDN
PAGEDN LDA (GETPTR),Y
STA (PUTPTR),Y
INY

Set page pointers to bottom of origin and destination blocks.
Y must equal zero whether we move more or less than a page.
More than one page to move?
If not, move less than a page down.
Move a page down.
Get a byte from origin block and put it in destination block.
Moved last byte in page?
BNE PAGEDN
INC GETPTR + 1
INC PUTPTR + 1
DEX
BNE PAGEDN
LDY #0

LESSDN
LDA (GETPTR), Y
STA (PUTPTR), Y
INY
SEC
CPY NUM
BCC LESSDN
JMP OK.RTN

Increment page pointers.
Still more than a page to move?
If so, move another page down.
Move less than a page down starting at the bottom.
Get a byte from origin...
and put it in destination block.
Adjust index for next byte.
Moved last byte yet?
If not, move another.
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

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:

![Flowchart of MOVER routine](image)

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.
.BLUE TEX,CR
.BLUE "MOVE TOOL"
.BLUE CR,LF,LF
.BLUE ETX
JSR SETADS Get starting address,
JSET DA ending address, and
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
JSR PRINT: other selected output devices will echo
.BLUE TEX the screen output.
.BLUE CR,LF,LF
.BLUE "SET DESTINATION ADDRESS"
.BYTE "AND PRESS Q."
.BLUE ETX
JSR VISMON Call the Visible Monitor, so user can
Set destination address equal to
address set by the user.
DHERE LDA SELECT Return to caller.
STA DEST .WORD 0 Pointer to destination of block to be
LDA SELECT+1 moved.
STA DEST+1
RTS
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 II:

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 “greater than” 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:

```
INITIALIZE POINTERS

DISPLAY CURRENT TEXT

GET A KEYSTROKE AND HANDLE IT
```

Figure 11.1: Structure of simple text editor.

<table>
<thead>
<tr>
<th>EDITOR</th>
<th>JSR SETBUF</th>
<th>Initialize pointers and variables required by the editor.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDLOOP</td>
<td>JSR SHOWIT</td>
<td>Show the user a portion of the text buffer.</td>
</tr>
<tr>
<td></td>
<td>JSR EDITIT</td>
<td>Let the user edit the buffer or move about within it.</td>
</tr>
<tr>
<td>CLC</td>
<td>BCC EDLOOP</td>
<td>Loop back to show the current text.</td>
</tr>
</tbody>
</table>

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 stor-
ing 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETBUF</td>
<td>JSR TVT.ON</td>
</tr>
<tr>
<td></td>
<td>JSR PRINT:</td>
</tr>
<tr>
<td></td>
<td>.BYTE TEX, CR, LF, LF</td>
</tr>
<tr>
<td></td>
<td>.BYTE 'SET UP EDIT BUFFER'</td>
</tr>
<tr>
<td></td>
<td>.BYTE CR, LF, LF, ETX</td>
</tr>
<tr>
<td>GETADS</td>
<td>JSR SETADS</td>
</tr>
<tr>
<td></td>
<td>JSR GOTOSA</td>
</tr>
<tr>
<td></td>
<td>RTS</td>
</tr>
</tbody>
</table>

Select TVT.

Display “SET UP EDIT BUFFER.”

Let user set starting address and end address of edit buffer.

Now SELECT = starting address of edit buffer.

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 (i.e., from the top of the screen to the bottom of the screen), you need only change the home position, which is set by SHOWIT.

| LINE 1: | X |
| LINE 2: | SOME CHARACTERS FROM TEXT BUFFER GO HERE |
| LINE 3: | M HHHH |

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, INSRCH 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 | JSR TVHOME | LDX TVCOLS | LDY #3 | JSR CLR.XY | JSR TVHOME | JSR TVDOWN | JSR TVPUSH | JSR LINE.2 | JSR TV.POP | JSR TVDOWN | JSR LINE.3 | JSR TV.POP | RTS |
|--------|------------|------------|------------|--------|------------|------------|------------|------------|------------|------------|------------|------------|------------|    |
|        | Save the zero-page bytes we’ll use. | Set home position of the edit display. | Clear 3 rows for the edit display. | Restore TV.PTR to home position of edit display. | Set TV.PTR to beginning of line 2 and save it. | Display text in line 2. | Set TV.PTR to beginning of line 3. | Display line 3. | Restore zero-page bytes used. | 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**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LINE.2</td>
<td>JSR</td>
<td>PUSHSL</td>
</tr>
<tr>
<td></td>
<td>LDA</td>
<td>TVCOLS</td>
</tr>
<tr>
<td></td>
<td>LSR</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>TAX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DEX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DEX</td>
<td></td>
</tr>
<tr>
<td>LOOP.1</td>
<td>JSR</td>
<td>DEC.SL</td>
</tr>
<tr>
<td></td>
<td>DEX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BPL</td>
<td>LOOP.1</td>
</tr>
<tr>
<td></td>
<td>LDA</td>
<td>TVCOLS</td>
</tr>
<tr>
<td></td>
<td>STA</td>
<td>COUNTR</td>
</tr>
<tr>
<td>LOOP.2</td>
<td>JSR</td>
<td>GET.SL</td>
</tr>
<tr>
<td></td>
<td>JSR</td>
<td>TV.PUT</td>
</tr>
<tr>
<td></td>
<td>JSR</td>
<td>TVSKIP</td>
</tr>
<tr>
<td></td>
<td>JSR</td>
<td>INC.SL</td>
</tr>
<tr>
<td></td>
<td>DEC</td>
<td>COUNTR</td>
</tr>
<tr>
<td></td>
<td>BPL</td>
<td>LOOP.2</td>
</tr>
<tr>
<td></td>
<td>JSR</td>
<td>POP.SL</td>
</tr>
<tr>
<td></td>
<td>RTS</td>
<td></td>
</tr>
</tbody>
</table>

Save SELECT pointer.  
Set X equal to half the width of the screen.  
Decrement SELECT X times.  
Initialize COUNTR. (We’re going to display TVCOLS characters.)  
Get a character from buffer.  
Put it on screen.  
Go to next screen position.  
Advance to next byte in buffer.  
Done last character in row?  
If not, do next character.  
Restore SELECT from stack.  
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

LDA TVCOLS
LSR A
SBC #2
JSR TVPLUS

A = TVCOLS/2
A = (TVCOLS/2) - 2
Now TV.PTR is pointing 2 characters to
the left of center of line 3 of the edit
display.

LDA EDMODE
CMP #1
BNE OVMODE
LDA #INSCHR
CLC
BCC TVMODE

WHAT IS CURRENT MODE?
IS IT INSERT MODE?
IF NOT, IT MUST BE OVERSTRIKE MODE.
IF SO, LOAD A WITH THE INSERT GRAPHIC.

OVMODE
LDA #OVCHR
TVMODE
JSR TV.PUT
LDA #2
JSR TVPLUS

LOAD A WITH THE OVERSTRIKE GRAPHIC.
PUT MODE GRAPHIC ON SCREEN.
Now TVPTR is pointing at the center of
line 3 of the edit display.

LDA ARROW
JSR TV.PUT
LDA #2
JSR TVPLUS

DISPLAY AN UP-ARROW HERE,
POINTING UP AT THE CURRENT CHARACTER.
Now TV.PTR is pointing at the position
reserved for the address of the current
character.

LDA SELECT+1
JSR VBYTE
LDA SELECT
JSR VBYTE
RTS

DISPLAY ADDRESS OF CURRENT
CHARACTER.
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**

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

DO.KEY  CMP MODEKY
         BNE IFNEXT
         DEC EDMODE
         BPL DO.END
         LDA #1
         STA EDMODE

DO.END  RTS

IFNEXT  CMP NEXTKY
         BNE IFPREV
         JSR NEXTCH

         RTS

IFPREV  CMP PREVKY
         BNE IF.RUB
         JSR PREVCH

         RTS

IF.RUB  CMP RUBKEY
         BNE IF.PRT
         JSR DELETE

         RTS

IF.PRT  CMP PRTKEY
         BNE IFFLSH
         JSR PRTBUF

         RTS

IFFLSH  CMP FLSHKY
         BNE CHARKY
         JSR FLUSH

         RTS

OK. It's not an editor function key, so it must be a regular character key. Therefore, 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
         BEQ STRIKE
         JSR INSERT
         RTS

STRIKE  JSR PUT.SL

Are we in overstrike mode?
If so, overstrike the character.
If not, insert the character...
and return.
the current character.
Advance to the next character position, and return to caller.
Save the character to be inserted, while we make space for it in the edit buffer...
PUSH the address of the current character onto the stack.
Push starting address of the buffer onto stack.
PUSH ending address of the buffer onto stack.
Set $SA = \text{SELECT}$, so current character will be the start of the block we'll move.
Advance to next character position in the text buffer.
If we're at the end of the buffer, we'll overwrite instead of inserting.
Set $\text{DEST} = \text{SELECT}$, so destination of block move will be 1 byte above block's start address (ie, we'll move a block up by 1 byte).
Decrement end address so we won't move text beyond the end of 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.
Open up 1 byte of space at the current character's location, by moving to $\text{DEST}$ the block specified by $SA$ and $EA$.
Restore $EA$ so it points to the last byte in the edit buffer.
Restore $SA$ so it points to the first byte in the edit buffer.
PLA
STA SA+1
JSR POP.SL

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
CMP #ETX
BEQ AN.ETX

Get currently selected character.
Is it an ETX?
If so, return to caller, bearing a negative return code.
JSR NEXTSL
RTS

If not, select next byte in the buffer, and return positive if we incremented SELECT; negative if SELECT already equaled EA.

AN.ETX  LDA $FF
RTS

Since we are on an ETX, we won’t increment 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
LDA SA+1
CMP SELECT+1
BCC SL.OK
BNE NOT.OK

Prepare to compare.
Is SELECT in a higher page than SA?

LDA SA
CMP SELECT
BEQ NO.DEC
BNE NOT.OK

If so, SELECT may be decremented.
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.
Is SELECT greater than SA?

SL.OK  JSR DEC.SL

If SELECT = SA, don’t decrement it.
If SELECT is less than SA, it’s not okay, so we’ll have to fix it.
SELECT is OK, because it’s greater than SA. Thus, we may decrement it and it will remain in the edit buffer.

LDA #0
RTS

Set a positive return code...
and return.

NOT.OK  LDA SA
STA SELECT
LDA SA+1

Since SELECT is less than SA, it is not even in the edit buffer. So give SELECT a legal value, by setting it = SA.

156 BEYOND GAMES
STA SELECT +1
LDA #0
RTS.

Set a positive return code...

NO.DEC  LDA #$FF
RTS

SELECT = SA, so change nothing. Set
a negative return code and return.

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

**FLUSH**

<table>
<thead>
<tr>
<th>FLUSH</th>
<th>JSR GOTOSA</th>
<th>Set SELECT to the first character position in the buffer.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOOP</td>
<td>LDA #ETX</td>
<td>Load accumulator with an ETX character...</td>
</tr>
<tr>
<td></td>
<td>JSR PUT.SL</td>
<td>and put it into the buffer.</td>
</tr>
<tr>
<td></td>
<td>JSR NEXTSL</td>
<td>Advance to next byte.</td>
</tr>
<tr>
<td></td>
<td>BPL FLOOP</td>
<td>If we haven't reached the last byte in the buffer, let's repeat the operation for this byte.</td>
</tr>
<tr>
<td></td>
<td>JSR GOTOSA</td>
<td>If we have reached the last byte in the buffer, let's set SELECT to the beginning of the buffer...</td>
</tr>
<tr>
<td></td>
<td>JSR RTS</td>
<td>and return.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>PRTBUF</th>
<th>JSR GOTOSA</th>
<th>Set SELECT to the start of the buffer.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRLOOP</td>
<td>JSR GET.SL</td>
<td>Get the currently selected character.</td>
</tr>
<tr>
<td></td>
<td>CMP #ETX</td>
<td>Is it an ETX character?</td>
</tr>
<tr>
<td></td>
<td>BEQ ENDPRT</td>
<td>If so, stop printing and return.</td>
</tr>
</tbody>
</table>
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 buffer, let’s get the next character from the buffer, and handle it.

ENDPRT  
Since we reached the end of the buffer, let’s return.

RTS  
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 |                                 |

| 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 current 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 is the start address of the text buffer, not of the block we just moved). |

| STA SA |
| STA SA+1 |
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 an Ohio Scientific C-IP. If you have a different system or 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; and Rubout or 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
  CMP #P
  BNE IF.U
  LDA PRINTR
  EOR #$FF
  STA PRINTR
  RTS

IF.U
  CMP #U
  BNE IF.H
  LDA USR.FN
  EOR #$FF
  STA USR.FN
  RTS

IF.H
  CMP #H
  BNE IF.M
  LDA PRINTR
  BNE NEXT.1
  JSR TVDUMP
  RTS

NEXT.1
  JSR PRDUMP
  RTS

IF.M
  CMP #M
  BNE IF.DIS
  JSR MOVER
  RTS

IF.DIS
  CMP #?
  BNE IF.T
  LDA PRINTR
  BNE NEXT.2
  JSR TV.DIS
  RTS

NEXT.2
  JSR PR.DIS
  RTS

IF.T
  CMP #T
  BNE EXIT
```

When EXTEND is called by the Visible Monitor's UPDATE routine, a character from the keyboard is in the accumulator.

- Is it the "P" key?
  - If not, perform the next test.
  - If so, toggle the printer flag...
- and return to caller.

- Is it the "U" key?
  - If not, perform the next test.
  - If so, toggle the user-output flag...
- and return.

- Is it the "H" key?
  - If not, perform the next test.
  - If so, print a hexdump.
- If not, dump to screen...
- and return.

- Print a hexdump...
- and return.

- Is it the "M" key?
  - If not, perform the next test.
  - If so, call the move tool.
- ...and return.

- Is it the "?" key?
  - If not, perform the next test.
  - If so, print a disassembly.
- If not, dump to screen...
- and return.

- Print a disassembly...
- and return.

- Is it the "T" key?
  - If not, return.
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 $10 at addresses $13D1, $13D2, and $13D3, respectively. To make the Visible Monitor’s UPDATE routine call EXTEND (instead of DUMMY), you must change $13D2 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 unextended Visible Monitor. Once you have changed $13D2, you may strike any key, but while you are changing $13D2, 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 $13D2, try out your new extensions of the Visible Monitor by pressing the now legal keys: “H,” “M,” “P,” “U,” “t,” 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. How you do that will depend on the system you have.

If you have an Apple II, you have an extended machine-language monitor built into your system. If the monitor doesn't come up on RESET, you can invoke it from BASIC with the following BASIC command:

\[ \text{POKE 0,0:CALL 0 [RETURN]} \]

(The string "[RETURN]" means press the carriage return key.)

This writes a 6502 BRK instruction into location $0000, and then executes a call to a machine-language subroutine at location $0000. The 6502, upon encountering the BRK instruction, will pass control to the Apple II ROM monitor. You'll know you're in the Apple II monitor because you'll see an asterisk (*) on the screen. Your Apple II documentation should tell you how to use this monitor to enter data into memory, dump memory, etc.

The Ohio Scientific C-IP has a much simpler monitor than the Apple II built into its ROM (read-only memory). Press BREAK on the Ohio Scientific C-IP and then press "M." You'll get the ROM monitor display and can use the ROM monitor to enter hexadecimal object code into memory. Unfortunately, although the Ohio Scientific ROM monitor lets you enter a machine-language program into memory by hand, or even from a cassette file in the proper format, it provides no facility for...
recording a machine-language program onto a cassette. So unless you plan to key
the Visible Monitor into memory and then leave your computer on forever, you’re
out of luck. However, you can SAVE a BASIC program on cassette, and then
LOAD it from cassette. And that’s the key: we’ll use the OSI C-1P’s ROM BASIC in-
terpreter to help get machine-language programs into memory.

And what if you have an Atari or a PET Computer? Each of these systems fea-
tures a BASIC interpreter in ROM (read-only memory), but lacks a machine-lan-
guage 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 ob-
ject code into memory, how can we save that object code onto a cassette? 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 com-
puter, but the BASIC OBJECT CODE LOADER is designed to let you know if
you’ve made a mistake. It won’t catch any 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 the
OBJECT CODE LOADER detects a checksum error, 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:
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.

Here is the object code loader:

```
100 REM
110 REM
120 DIM BYTE(8)
130 READ FIRST
140 REM
150 READ LAST
160 REM
170 FOR LINE=FIRST TO LAST
180 GOSUB 300
190 NEXT LINE
200 PRINT "LOADED LINES"",FIRST,"THROUGH",LAST,"SUCCESSFULLY."
210 END
220 REM
230 REM
240 REM
300 READ A
310 SUM=A
320 FOR J=1 TO 8
321 REM
330 READ TEMP: BYTE(J)=TEMP
340 SUM=SUM+BYTE(J)
341 REM
350 NEXT J
360 REM
370 READ CHECK
380 IF SUM <> CHECK THEN 500
```

**OBJECT CODE LOADER** by Ken Skier

:REM Initialize BYTE array.
:REM Get the line number of the first DATA statement containing object code.
:REM Get the line number of the last DATA statement containing object code.
:REM Read the specified DATA lines.
:REM Load next data line into memory.
:REM If not done, read next DATA line.
:REM If done, say so.

Subroutine at 300 handles one DATA statement.
:REM Get address for object code.
:REM Initialize calculated sum of data.
:REM Get 8 bytes of object code from data.
:REM Put them in the byte array, and
:REM add them to the calculated sum of data.
:REM Now we have the 8 bytes, and we have calculated the sum of the data.
:REM Get checksum from data line.
:REM If checksum error, handle it.
390 FOR J=1 TO 8
400 POKE A+J-1, BYTE(J)
410 NEXT J
420 RETURN
430 REM
440 REM

500 PRINT "CHECKSUM ERROR IN DATA LINE", LINE
510 PRINT "START ADDRESS GIVEN IN BAD DATA LINE IS", A
520 END
530 REM
540 REM
550 REM
570 REM
600 DATA ????
610 REM
611 REM
612 REM
620 DATA ????
630 REM
631 REM

:REM Since there is no checksum error, :REM poke the data into the specified :REM portion of memory,
:REM and return to caller.

Checksum error-handling code follows.

The next two DATA statements specify the range of DATA statements that contain object code.

:REM This should be the number of the first DATA statement containing object code.

:REM This should be the number of the last DATA statement containing object 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 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 $0FFF (decimal 4095). You can insure that you don’t run over this limit by setting 4095 as the top of memory available to your system’s BASIC interpreter. How do you set the top of memory available to the BASIC interpreter? That varies from system to system, so consult the B Appendix for your system.

Before you can append to the OBJECT CODE LOADER all the DATA statements from Appendix E1, your BASIC interpreter may give you an OUT OF MEMORY error (MEMORY FULL). When 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
START ADDRESS GIVEN IN BAD DATA LINE IS
```

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 4442. 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 E10, as well as the DATA statements in the E Appendix containing system data for your computer (one of the Appendices E11 thru E14). 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 $1207.

Using the Ohio Scientific C-IP ROM monitor, you can activate the Visible Monitor simply by typing:

1207G

Using the Apple II ROM monitor, you can call the Visible Monitor with the command:

G1207 [RETURN]

Using the Atari 400 or 800 with its BASIC cartridge plugged in, you can invoke the Visible Monitor with the BASIC command:
\[ X = \text{USR}(4615) \] [RETURN]

In Atari BASIC, you can call a machine-language subroutine by passing the address of that subroutine as a parameter to the USR function. Since $1207$ is 4615 in decimal, the command \( X = \text{USR}(4615) \) causes Atari BASIC to call the subroutine at $1207$. (The value returned by that subroutine will then be stored in the BASIC variable \( X \) — not in the 6502’s X register. But that doesn’t concern us because the Visible Monitor isn’t designed to return a value to its caller.)

Using the PET 2001, you can invoke the Visible Monitor from BASIC in the immediate mode with the following BASIC command:

\[ \text{SYS (4615)} \]

When you press (RETURN), you’ll see the Visible Monitor display, because \( \text{SYS (4615)} \) causes BASIC to call the subroutine at address 4615 decimal, which is $1207$—the entry point for the Visible Monitor.

If and when you press “Q” to quit the Visible Monitor, the Visible Monitor will return to its caller — PET BASIC. (The Visible Monitor doesn’t leave much room for a PET BASIC program, since your BASIC program and its arrays, variables, etc cannot require memory beyond $0\text{FFFF}$, but the Visible Monitor should work very well with a small PET BASIC program. In any case, it’s reassuring to have a new program such as the Visible Monitor return to a familiar one such as the PET BASIC interpreter.)

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 failed to enter the display code properly.

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 screen memory and place new data into that address. If you picked a place in display memory that is not cleared by the Visible Monitor (ie: a place not in the top five rows of the screen), then you should be able to place arbitrary characters on the screen just by using the Visible Monitor to store arbitrary values in the selected 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.

To extend the Visible Monitor as described in Chapter 12, store a $BO in address $13D2. To disable the features described in Chapter 12, store a $10 in address $13D2. 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.

NOTE:

The author intends to provide the software in this book for sale on cassettes compatible with the Apple II, Atari, Ohio Scientific, and PET computers. If you prefer to load your software from cassette, rather than enter it in by hand, contact the author through BYTE Books.
Appendices
## Appendix A1:
### Hexadecimal Conversion Table

<table>
<thead>
<tr>
<th>HEX</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>00</th>
<th>000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>256</td>
<td>4096</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>512</td>
<td>8192</td>
</tr>
<tr>
<td>3</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>768</td>
<td>12288</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>1024</td>
<td>16384</td>
</tr>
<tr>
<td>5</td>
<td>80</td>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
<td>1280</td>
<td>20480</td>
</tr>
<tr>
<td>6</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
<td>108</td>
<td>109</td>
<td>110</td>
<td>111</td>
<td>1536</td>
<td>24576</td>
</tr>
<tr>
<td>7</td>
<td>112</td>
<td>113</td>
<td>114</td>
<td>115</td>
<td>116</td>
<td>117</td>
<td>118</td>
<td>119</td>
<td>120</td>
<td>121</td>
<td>122</td>
<td>123</td>
<td>124</td>
<td>125</td>
<td>126</td>
<td>127</td>
<td>1792</td>
<td>28672</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>129</td>
<td>130</td>
<td>131</td>
<td>132</td>
<td>133</td>
<td>134</td>
<td>135</td>
<td>136</td>
<td>137</td>
<td>138</td>
<td>139</td>
<td>140</td>
<td>141</td>
<td>142</td>
<td>143</td>
<td>2048</td>
<td>32768</td>
</tr>
<tr>
<td>9</td>
<td>144</td>
<td>145</td>
<td>146</td>
<td>147</td>
<td>148</td>
<td>149</td>
<td>150</td>
<td>151</td>
<td>152</td>
<td>153</td>
<td>154</td>
<td>155</td>
<td>156</td>
<td>157</td>
<td>158</td>
<td>159</td>
<td>2304</td>
<td>36864</td>
</tr>
<tr>
<td>A</td>
<td>160</td>
<td>161</td>
<td>162</td>
<td>163</td>
<td>164</td>
<td>165</td>
<td>166</td>
<td>167</td>
<td>168</td>
<td>169</td>
<td>170</td>
<td>171</td>
<td>172</td>
<td>173</td>
<td>174</td>
<td>175</td>
<td>2560</td>
<td>40960</td>
</tr>
<tr>
<td>B</td>
<td>176</td>
<td>177</td>
<td>178</td>
<td>179</td>
<td>180</td>
<td>181</td>
<td>182</td>
<td>183</td>
<td>184</td>
<td>185</td>
<td>186</td>
<td>187</td>
<td>188</td>
<td>189</td>
<td>190</td>
<td>191</td>
<td>2816</td>
<td>45056</td>
</tr>
<tr>
<td>C</td>
<td>192</td>
<td>193</td>
<td>194</td>
<td>195</td>
<td>196</td>
<td>197</td>
<td>198</td>
<td>199</td>
<td>200</td>
<td>201</td>
<td>202</td>
<td>203</td>
<td>204</td>
<td>205</td>
<td>206</td>
<td>207</td>
<td>3072</td>
<td>49152</td>
</tr>
<tr>
<td>D</td>
<td>208</td>
<td>209</td>
<td>210</td>
<td>211</td>
<td>212</td>
<td>213</td>
<td>214</td>
<td>215</td>
<td>216</td>
<td>217</td>
<td>218</td>
<td>219</td>
<td>220</td>
<td>221</td>
<td>222</td>
<td>223</td>
<td>3328</td>
<td>53248</td>
</tr>
<tr>
<td>E</td>
<td>224</td>
<td>225</td>
<td>226</td>
<td>227</td>
<td>228</td>
<td>229</td>
<td>230</td>
<td>231</td>
<td>232</td>
<td>233</td>
<td>234</td>
<td>235</td>
<td>236</td>
<td>237</td>
<td>238</td>
<td>239</td>
<td>3584</td>
<td>57344</td>
</tr>
<tr>
<td>F</td>
<td>240</td>
<td>241</td>
<td>242</td>
<td>243</td>
<td>244</td>
<td>245</td>
<td>246</td>
<td>247</td>
<td>248</td>
<td>249</td>
<td>250</td>
<td>251</td>
<td>252</td>
<td>253</td>
<td>254</td>
<td>255</td>
<td>3840</td>
<td>61440</td>
</tr>
</tbody>
</table>
## Appendix A2:
### ASCII Character Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Char</th>
<th>Code</th>
<th>Char</th>
<th>Code</th>
<th>Char</th>
<th>Code</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>NUL</td>
<td>20</td>
<td>SP</td>
<td>40</td>
<td>@</td>
<td>60</td>
<td>`</td>
</tr>
<tr>
<td>01</td>
<td>SOH</td>
<td>21</td>
<td>!</td>
<td>41</td>
<td>A</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>02</td>
<td>STX</td>
<td>22</td>
<td>&quot;</td>
<td>42</td>
<td>B</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>03</td>
<td>ETX</td>
<td>23</td>
<td>#</td>
<td>43</td>
<td>C</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>04</td>
<td>EOT</td>
<td>24</td>
<td>$</td>
<td>44</td>
<td>D</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>05</td>
<td>ENQ</td>
<td>25</td>
<td>%</td>
<td>45</td>
<td>E</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>06</td>
<td>ACK</td>
<td>26</td>
<td>&amp;</td>
<td>46</td>
<td>F</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>07</td>
<td>BEL</td>
<td>27</td>
<td>'</td>
<td>47</td>
<td>G</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>08</td>
<td>BS</td>
<td>28</td>
<td>(</td>
<td>48</td>
<td>H</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>09</td>
<td>HT</td>
<td>29</td>
<td>)</td>
<td>49</td>
<td>I</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>0A</td>
<td>LF</td>
<td>2A</td>
<td>+</td>
<td>4A</td>
<td>J</td>
<td>6A</td>
<td>j</td>
</tr>
<tr>
<td>0B</td>
<td>VT</td>
<td>2B</td>
<td></td>
<td>4B</td>
<td>K</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>0C</td>
<td>FF</td>
<td>2C</td>
<td></td>
<td>4C</td>
<td>L</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>0D</td>
<td>CR</td>
<td>2D</td>
<td></td>
<td>4D</td>
<td>M</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>0E</td>
<td>SO</td>
<td>2E</td>
<td></td>
<td>4E</td>
<td>N</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>0F</td>
<td>SI</td>
<td>2F</td>
<td></td>
<td>4F</td>
<td>O</td>
<td>6F</td>
<td>o</td>
</tr>
<tr>
<td>10</td>
<td>DLE</td>
<td>30</td>
<td>0</td>
<td>50</td>
<td>P</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>11</td>
<td>DC1</td>
<td>31</td>
<td>1</td>
<td>51</td>
<td>Q</td>
<td>71</td>
<td>q</td>
</tr>
<tr>
<td>12</td>
<td>DC2</td>
<td>32</td>
<td>2</td>
<td>52</td>
<td>R</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>13</td>
<td>DC3</td>
<td>33</td>
<td>3</td>
<td>53</td>
<td>S</td>
<td>73</td>
<td>s</td>
</tr>
<tr>
<td>14</td>
<td>DC4</td>
<td>34</td>
<td>4</td>
<td>54</td>
<td>T</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>15</td>
<td>NAK</td>
<td>35</td>
<td>5</td>
<td>55</td>
<td>U</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>16</td>
<td>SYN</td>
<td>36</td>
<td>6</td>
<td>56</td>
<td>V</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>17</td>
<td>ETB</td>
<td>37</td>
<td>7</td>
<td>57</td>
<td>W</td>
<td>77</td>
<td>w</td>
</tr>
<tr>
<td>18</td>
<td>CAN</td>
<td>38</td>
<td>8</td>
<td>58</td>
<td>X</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>19</td>
<td>EM</td>
<td>39</td>
<td>9</td>
<td>59</td>
<td>Y</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>1A</td>
<td>SUB</td>
<td>3A</td>
<td>:</td>
<td>5A</td>
<td>Z</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>1B</td>
<td>ESC</td>
<td>3B</td>
<td>;</td>
<td>5B</td>
<td>[</td>
<td>7B</td>
<td>{</td>
</tr>
<tr>
<td>1C</td>
<td>FS</td>
<td>3C</td>
<td>&lt;</td>
<td>5C</td>
<td>\</td>
<td>7C</td>
<td></td>
</tr>
<tr>
<td>1D</td>
<td>GS</td>
<td>3D</td>
<td>=</td>
<td>5D</td>
<td>]</td>
<td>7D</td>
<td>}</td>
</tr>
<tr>
<td>1E</td>
<td>RS</td>
<td>3E</td>
<td>&gt;</td>
<td>5E</td>
<td>^</td>
<td>7E</td>
<td>~</td>
</tr>
<tr>
<td>1F</td>
<td>US</td>
<td>3F</td>
<td>?</td>
<td>5F</td>
<td>—</td>
<td>7F</td>
<td>DEL</td>
</tr>
</tbody>
</table>
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
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
<td>Jump to New Location</td>
</tr>
<tr>
<td>JSR</td>
<td>Jump to New Location Saving Return Address</td>
</tr>
<tr>
<td>LDA</td>
<td>Load Accumulator with Memory</td>
</tr>
<tr>
<td>LDX</td>
<td>Load Register X with Memory</td>
</tr>
<tr>
<td>LDY</td>
<td>Load Register Y with Memory</td>
</tr>
<tr>
<td>LSR</td>
<td>Shift Right One Bit (Memory or Accumulator)</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
</tr>
<tr>
<td>ORA</td>
<td>&quot;OR&quot; Memory with Accumulator</td>
</tr>
<tr>
<td>PHA</td>
<td>Push Accumulator on Stack</td>
</tr>
<tr>
<td>PHP</td>
<td>Push Processor Status on Stack</td>
</tr>
<tr>
<td>PLA</td>
<td>Pull Accumulator from Stack</td>
</tr>
<tr>
<td>PLP</td>
<td>Pull Processor Status from Stack</td>
</tr>
<tr>
<td>ROL</td>
<td>Rotate One Bit Left (Memory or Accumulator)</td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate One Bit Right (Memory or Accumulator)</td>
</tr>
<tr>
<td>RTI</td>
<td>Return from Interrupt</td>
</tr>
<tr>
<td>RTS</td>
<td>Return from Subroutine</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract Memory from Accumulator with Borrow</td>
</tr>
<tr>
<td>SEC</td>
<td>Set Carry Flag</td>
</tr>
<tr>
<td>SED</td>
<td>Set Decimal Mode</td>
</tr>
<tr>
<td>SEI</td>
<td>Set Interrupt Disable Status</td>
</tr>
<tr>
<td>STA</td>
<td>Store Accumulator in Memory</td>
</tr>
<tr>
<td>STX</td>
<td>Store Register X in Memory</td>
</tr>
<tr>
<td>STY</td>
<td>Store Register Y in Memory</td>
</tr>
<tr>
<td>TAX</td>
<td>Transfer Accumulator to Register X</td>
</tr>
<tr>
<td>TAY</td>
<td>Transfer Accumulator to Register Y</td>
</tr>
<tr>
<td>TSX</td>
<td>Transfer Stack Pointer to Register X</td>
</tr>
<tr>
<td>TXA</td>
<td>Transfer Register X to Accumulator</td>
</tr>
<tr>
<td>TXS</td>
<td>Transfer Register X to Stack Pointer</td>
</tr>
<tr>
<td>TYA</td>
<td>Transfer Register Y to Accumulator</td>
</tr>
</tbody>
</table>
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
67 — 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 | A8 — TAY |
| 81 — STA — (Indirect,X) | A9 — LDA — Immediate |
| 82 — Future Expansion | AA — TAX |
| 83 — Future Expansion | AB — Future Expansion |
| 84 — STY — Zero Page | AC — LDY — Absolute |
| 85 — STA — Zero Page | AD — LDA — Absolute |
| 86 — STX — Zero Page | AE — LDX — Absolute |
| 87 — Future Expansion | AF — Future Expansion |
| 88 — DEY | B0 — BCS |
| 89 — Future Expansion | B1 — LDA — (Indirect),Y |
| 8A — TXA | B2 — Future Expansion |
| 8B — Future Expansion | B3 — Future Expansion |
| 8C — STY — Absolute | B4 — LDY — Zero Page,X |
| 8D — STA — Absolute | B5 — LDA — Zero Page,X |
| 8E — STX — Absolute | B6 — LDX — Zero Page,Y |
| 8F — Future Expansion | B7 — Future Expansion |
| 90 — BCC | B8 — CLV |
| 91 — STA — (Indirect),Y | B9 — LDA — Absolute,Y |
| 92 — Future Expansion | BA — TSX |
| 93 — Future Expansion | BB — Future Expansion |
| 94 — STY — Zero Page,X | BC — LDY — Absolute,X |
| 95 — STA — Zero Page,X | BD — LDA — Absolute,X |
| 96 — STX — Zero Page,Y | BE — LDX — Absolute,Y |
| 97 — Future Expansion | BF — Future Expansion |
| 98 — TYA | C0 — CPY — Immediate |
| 99 — STA — Absolute,Y | C1 — CMP — (Indirect,X) |
| 9A — TXS | C2 — Future Expansion |
| 9B — Future Expansion | C3 — Future Expansion |
| 9C — Future Expansion | C4 — CPY — Zero Page |
| 9D — STA — Absolute,X | C5 — CMP — Zero Page |
| 9E — Future Expansion | C6 — DEC — Zero Page |
| 9F — Future Expansion | C7 — Future Expansion |
| A0 — LDY — Immediate | C8 — INY |
| A1 — LDA — (Indirect,X) | C9 — CMP — Immediate |
| A2 — LDX — Immediate | CA — DEX |
| A3 — Future Expansion | CB — Future Expansion |
| A4 — LDY — Zero Page | CC — CPY — Absolute |
| A5 — LDA — Zero Page | CD — CMP — Absolute |
| A6 — LDX — Zero Page | CE — DEC — Absolute |
| A7 — Future Expansion | 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)

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Immediate</th>
<th>Zero Page</th>
<th>Zero Page, X</th>
<th>Zero Page, Y</th>
<th>Absolute</th>
<th>Absolute, X</th>
<th>Absolute, Y</th>
<th>Implied</th>
<th>Relative</th>
<th>(Indirect), X</th>
<th>(Indirect), Y</th>
<th>Absolute Indirect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>2 3 4</td>
<td>4 4* 4*</td>
<td>4*</td>
<td>6 5*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>2 3 4</td>
<td>4 4* 4*</td>
<td>4*</td>
<td>6 5*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASL</td>
<td>2 5 6</td>
<td>6 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIT</td>
<td>3 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BVC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BVS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>2 3 4</td>
<td>4 4* 4*</td>
<td>4*</td>
<td>6 5*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPX</td>
<td>2 3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPY</td>
<td>2 3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>5 6</td>
<td>6 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EOR</td>
<td>2 3 4</td>
<td>4 4* 4*</td>
<td>4*</td>
<td>6 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accumulator</td>
<td>Immediate</td>
<td>Zero Page</td>
<td>Zero Page, X</td>
<td>Zero Page, Y</td>
<td>Absolute</td>
<td>Absolute, X</td>
<td>Absolute, Y</td>
<td>Implied</td>
<td>Relative</td>
<td>(Indirect), X</td>
<td>(Indirect), Y</td>
<td>Absolute, Indirect</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>-----------</td>
<td>--------------</td>
<td>--------------</td>
<td>----------</td>
<td>-------------</td>
<td>-------------</td>
<td>---------</td>
<td>----------</td>
<td>--------------</td>
<td>----------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>INC</td>
<td>.</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>INX</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>INY</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>JMP</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>3</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>5</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>JSR</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>6</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>LDA</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4*</td>
<td>4*</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>6</td>
</tr>
<tr>
<td>LDX</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4*</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>5*</td>
</tr>
<tr>
<td>LDY</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4*</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>LSR</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>NOP</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>ORA</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4*</td>
<td>4*</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>6</td>
</tr>
<tr>
<td>PHA</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>3</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>PHP</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>3</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>PLA</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>4</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>PLP</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>4</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>ROL</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>ROR</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>RTI</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>6</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>RTS</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>6</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>SBC</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4*</td>
<td>4*</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>6</td>
</tr>
<tr>
<td>SEC</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>SED</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>SEI</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>STA</td>
<td>.</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>6</td>
</tr>
<tr>
<td>STX*</td>
<td>.</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>STY**</td>
<td>.</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>TAX</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>TAY</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>TSX</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>TXA</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>TXS</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>TYA</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

* 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

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Absolute</th>
<th>Absolute X</th>
<th>Absolute Y</th>
<th>Accumulator</th>
<th>Immediate</th>
<th>Implied</th>
<th>Indirect</th>
<th>Indirect X</th>
<th>Indirect Y</th>
<th>Relative</th>
<th>Zero Page</th>
<th>Zero Page X</th>
<th>Zero Page Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>6D</td>
<td>7D</td>
<td>79</td>
<td>69</td>
<td></td>
<td>61</td>
<td>71</td>
<td>65</td>
<td>75</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>2D</td>
<td>3D</td>
<td>39</td>
<td>29</td>
<td></td>
<td>21</td>
<td>31</td>
<td>25</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASL</td>
<td>0E</td>
<td>1E</td>
<td>0A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>06</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>90</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIT</td>
<td>2C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BVC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BVS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>70</td>
</tr>
<tr>
<td>CLC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>58</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Addressing Modes

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Absolute</th>
<th>Absolute X</th>
<th>Absolute Y</th>
<th>Accumulator</th>
<th>Immediate</th>
<th>Implied</th>
<th>Indirect</th>
<th>Indirect X</th>
<th>Indirect Y</th>
<th>Relative</th>
<th>Zero Page</th>
<th>Zero Page X</th>
<th>Zero Page Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>CD</td>
<td>DD</td>
<td>D9</td>
<td>C9</td>
<td></td>
<td>C1</td>
<td>D1</td>
<td>C5</td>
<td>D5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPX</td>
<td>EC</td>
<td></td>
<td></td>
<td>E0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPY</td>
<td>CC</td>
<td></td>
<td></td>
<td>C0</td>
<td></td>
<td></td>
<td></td>
<td>C4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>CE</td>
<td>DE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C6</td>
<td>D6</td>
</tr>
<tr>
<td>DEX</td>
<td></td>
<td></td>
<td></td>
<td>CA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEY</td>
<td></td>
<td></td>
<td></td>
<td>88</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EOR</td>
<td>4D</td>
<td>5D</td>
<td>59</td>
<td>49</td>
<td></td>
<td>41</td>
<td>51</td>
<td>45</td>
<td>55</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC</td>
<td>EE</td>
<td>FE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E6</td>
<td>F6</td>
</tr>
<tr>
<td>INX</td>
<td></td>
<td></td>
<td></td>
<td>E8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INY</td>
<td></td>
<td></td>
<td></td>
<td>C8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td>4C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSR</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>AD</td>
<td>BD</td>
<td>B9</td>
<td>A9</td>
<td></td>
<td>A1</td>
<td>B1</td>
<td>A5</td>
<td>B5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDX</td>
<td>AE</td>
<td>BE</td>
<td></td>
<td>A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDY</td>
<td>AC</td>
<td>BC</td>
<td></td>
<td>A0</td>
<td></td>
<td></td>
<td></td>
<td>A4</td>
<td>B4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSR</td>
<td>4E</td>
<td>5E</td>
<td>4A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>46</td>
<td>56</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td>EA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORA</td>
<td>0D</td>
<td>1D</td>
<td>19</td>
<td>09</td>
<td></td>
<td>01</td>
<td>11</td>
<td>05</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PHA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>68</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROL</td>
<td>2E</td>
<td>3E</td>
<td>2A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26</td>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROR</td>
<td>6E</td>
<td>7E</td>
<td>6A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>66</td>
<td>76</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

182 BEYOND GAMES
<table>
<thead>
<tr>
<th>Memonics</th>
<th>Addressing Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS</td>
<td>ABSOLUTE</td>
</tr>
<tr>
<td>SBC</td>
<td>ABSOLUTE,X</td>
</tr>
<tr>
<td>ED</td>
<td>ABSOLUTE,Y</td>
</tr>
<tr>
<td>FD</td>
<td>ACCUMULATOR</td>
</tr>
<tr>
<td>F9</td>
<td>IMMEDIATE</td>
</tr>
<tr>
<td>60</td>
<td>IMPLIED</td>
</tr>
<tr>
<td>8A</td>
<td>INDIRECT</td>
</tr>
<tr>
<td>A8</td>
<td>INDIRECT,X</td>
</tr>
<tr>
<td>9A</td>
<td>INDIRECT,Y</td>
</tr>
<tr>
<td>98</td>
<td>RELATIVE</td>
</tr>
<tr>
<td>38</td>
<td>ZERO PAGE</td>
</tr>
<tr>
<td>78</td>
<td>ZERO PAGE,X</td>
</tr>
<tr>
<td>81</td>
<td>ZERO PAGE,Y</td>
</tr>
</tbody>
</table>
Appendix B1:
The Ohio Scientific Challenger I-P

The Ohio Scientific Challenger I-P is the simplest of the systems considered in this book. Its screen is mapped in the manner described in Chapter 5: the lowest screen address is in the upper left corner, and the screen addresses increase uniformly as you move to the right and down the screen. Any ASCII character stored in screen memory will be displayed properly on the video screen; it is not necessary to replace the ASCII character with a system-specific display code. Therefore, the system data block may be initialized as shown in Appendices C13 and E12.

Incidentally, the OSI C-IP’s screen TVT subroutine at $BF2D stores the relative location of the cursor in $0200. Modify $0200 and you change the next location at which a character will be printed to the screen.

If you have an Ohio Scientific BASIC-in-ROM system other than the Challenger I-P, it may have different character input/output routines. If so, examine the following locations:

<table>
<thead>
<tr>
<th>BASIN</th>
<th>$FFEB</th>
<th>General character-input routine for OSI BASIC-in-ROM.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASOUT</td>
<td>$FFEE</td>
<td>General character-output routine for OSI BASIC-in-ROM.</td>
</tr>
</tbody>
</table>

For example, in the OSI C-IP you can get a character from the keyboard by calling $FEED, or you may call OSI’s general character-input routine at $FFEB. This routine gets a character from the keyboard unless the SAVE flag is set, in which case it gets a character from the cassette input port. Similarly, in the OSI C-IP you can print a character to the screen by calling $BF2D, or send a character to the cassette output port by calling $FCB1. Or, you can simply call OSI’s general character-output routine at $FFEE, which outputs the accumulator to the screen and, if the SAVE flag is set, echoes to the serial port as well.

Thus, even if you don’t know the addresses of your OSI system’s specific I/O routines, you can set ROMKEY =$FFEB and ROMTVT =$FFEE. When you RESET
your system, the Ohio Scientific Operating System will automatically "hook" those routines to your keyboard for input and to your screen for output.

Setting the Top of Memory

If you wish to load object code using the BASIC OBJECT CODE LOADER (see Chapter 13) you must first set the top of memory available to your BASIC interpreter to $0FFF. Do this as part of cold-starting BASIC. To cold-start BASIC, turn on your OSI computer, press the (BREAK) key, and then press 'C'. The screen will prompt, "Memory Size?" Type "4095" and then press (RETURN). Now BASIC will use the lowest 4K of RAM, leaving memory from $1000 and up available to machine-language programs.

With the top of memory set to $0FFF, you may enter and RUN the BASIC programs that load object code into your computer's memory.

Calling Machine-Language Code from BASIC

To call a machine-language subroutine from BASIC, first set the pointer at $000B, 000C so it points to the subroutine, and then call that subroutine with BASIC's USR function, either in the immediate mode or from within a BASIC program. For example, let's say you wish to call the Visible Monitor from BASIC. The Visible Monitor's entry point is at $1207, so we must make $000B,000C point to $1207. This means storing 07 in $000B, and storing 12 (decimal 18) in $000C. The following line will do that for us:

POKE 11,7:POKE 12,18

Now we may invoke the Visible Monitor with the line:

X = USR(X)

or with any other line that uses the USR function.

Note that the USR function does not set a BASIC variable equal to the contents of some register in the 6502; in fact, the line X = USR(X) will not change the value of the BASIC variable X at all. Thus, the USR function lets you activate any desired machine-language subroutine, but it doesn't let you capture a value returned by such
a subroutine. If you want a machine-language subroutine to return some value which you can then use in a BASIC program, you'll have to make the machine-language subroutine store its value or values somewhere in memory, and then have the BASIC program PEEK that memory location after it has called the machine-language subroutine via the USR function.
Appendix B2:
The PET 2001

Display Memory

The PET screen is mapped conventionally, with the HOME address at $8000 (32,768 decimal). It has 25 rows, each consisting of 40 characters. The address of each screen location is 40 ($28) greater than the address of the screen location directly above it. Thus, the screen parameters for the PET 2001 are:

```
HOME .WORD $8000,
ROWINC .BYTE $28
TVCOLS .BYTE 39
TVROWS .BYTE 24
```

(We count columns from zero.)

(We count rows from zero.)

PET Character Set

However, although the PET screen buffer is mapped conventionally, you cannot simply store an ASCII character in screen memory if you wish to see that ASCII character on the screen. The PET character generator introduces a few wrinkles and you must compensate carefully if you are to display ASCII characters properly on the screen.

For example, if you store $31 (the code for an ASCII “1”) in the PET’s display memory, then you will see a “1” displayed on the screen. So far, so good. The same is true for all ASCII digits and for some ASCII punctuation marks. But if you store $45 (ASCII code for an upper case “E”) in screen memory, then you won’t see an “E” on the screen; you’ll see either a lowercase “e” or else a horizontal line segment much longer than a hyphen. What’s happening?

The PET 2001 features a memory location, $E84C (59468) which has a special effect on the video-display circuitry. The value stored in that address selects for the video display one character set or another.
To see how the choice of character set affects the display, enter the following BASIC program into your PET:

```
100 REM DISPLAY PET CHARACTER SET
110 REM IN 16 BY 16 MATRIX
120 REM
130 HOME=32768
140 CHAR=0
150 FOR ROW=0 TO 15
160 FOR COL=0 TO 15
170 POKE (HOME+COL)+(40*ROW),CHAR
180 CHAR=CHAR+1
190 NEXT COL
200 NEXT ROW
210 END
```

Before running this program, clear the screen by holding down the PET's SHIFT key at the same time that you depress the CLR/HOME key. When the screen is clear, use the CRSR SOUTH key to move the cursor down seventeen rows. Then type RUN and press RETURN. You'll see one PET character set appear in a 16 by 16 matrix in the upper left portion of your PET's screen.

What you'll see on your screen will look like table B2.1 (without the labeled axes).

**Table B2.1: The PET character set.**

<table>
<thead>
<tr>
<th>LEFT NYBBLE OF CHARACTER</th>
<th>RIGHT NYBBLE OF CHARACTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-</td>
<td>@</td>
</tr>
<tr>
<td>1-</td>
<td>P</td>
</tr>
<tr>
<td>2-</td>
<td>!</td>
</tr>
<tr>
<td>3-</td>
<td>0</td>
</tr>
<tr>
<td>4-</td>
<td>a</td>
</tr>
<tr>
<td>5-</td>
<td>p</td>
</tr>
<tr>
<td>6-</td>
<td>_</td>
</tr>
<tr>
<td>7-</td>
<td>_</td>
</tr>
<tr>
<td>8-</td>
<td>@</td>
</tr>
<tr>
<td>9-</td>
<td>P</td>
</tr>
<tr>
<td>A-</td>
<td>!</td>
</tr>
<tr>
<td>B-</td>
<td>0</td>
</tr>
<tr>
<td>C-</td>
<td>a</td>
</tr>
<tr>
<td>D-</td>
<td>p</td>
</tr>
<tr>
<td>E-</td>
<td>_</td>
</tr>
<tr>
<td>F-</td>
<td>_</td>
</tr>
</tbody>
</table>
In this chart, special graphic characters are indicated by an underline. Look at your PET screen to see those special graphics in all their glorious detail.

Note that the characters for $80$ thru $\$FF$ are the same as for $00$ thru $7F$, but in reverse intensity. The low 128 characters ($00$ thru $7F$) are “normal” — that is, white characters on a dark background; whereas the high 128 characters ($80$ thru $\$FF$) are in reverse video — dark characters in a white background. An “A” in normal intensity may be displayed by storing an $01$ somewhere in the screen memory; a reverse intensity “A” may be displayed by storing an $81$ somewhere in screen memory. From this pattern we can derive a handy corollary: to reverse the intensity of any character on the screen, simply reverse its bit 7. You don’t even have to know what the character represents; just toggle bit 7 and you change its intensity.

The chart in figure B2.1 (and on your PET screen) shows one complete character set because the BASIC program stores every 8-bit value, from $00$ thru $\$FF$, into the screen buffer. But I mentioned two character sets. What must you do to see the second character set?

If the cursor is within three rows of the bottom of the screen, move it up so that it is at least three rows above the bottom of the screen. This will insure that you don’t scroll part of the character set up off the screen when you execute the following BASIC command in the immediate mode:

\[
\text{POKE 59468,12}
\]

Did that change the display? If not, then execute the following BASIC command in the immediate mode (again being sure that the cursor is at least three rows from the bottom of the screen):

\[
\text{POKE 59468,14}
\]

Depending on the value stored in 59468 ($\$E84C$), one or another character set will be displayed. The values of the bytes stored in screen memory will not change when you change the contents of $\$E84C$, but in some cases the displayed characters will change. In the ranges 00 thru $\$3F$ and $\$80$ thru $\$BF$, the two character sets are identical. But in the ranges $\$40$ thru $\$7F$ and $\$C0$ thru $\$FF$, they differ.

Both character sets include numbers, uppercase letters, and certain punctuation marks; but only one character set includes lowercase letters and the remaining punctuation marks. The second character set lacks lowercase letters and these punctuation marks, offering instead a set of special graphics characters, including playing-card suits. POKE 59468,14 to select the former character set (thereby making possible the display of all printable ASCII characters); POKE 59468,12 to select the latter character set (thereby making possible the display of the gaming graphics).
FIXCHR

Note that neither character set corresponds directly to ASCII. If you have an
ASCII character in the accumulator and you want to display the appropriate graphic
character on the screen, you must first call FIXCHR (as TV.PUT does, in Chapter 5).
When an ASCII character is passed in the accumulator, FIXCHR must return in the
accumulator the proper PET display code for that character. FIXCHR’s caller may
then store this display code in memory, thereby placing on the screen an appropriate
image of the original ASCII character.

How will FIXCHR work? By examining the PET character set and comparing it
to Appendix A2, ASCII codes, we can see a solution in the form of the following
algorithm:

- If a character is in the range $40$ thru $5F$, subtract $40$ and return.
- If a character is in the range $20$ thru $3F$, return.
- If a character is in the range $60$ thru $7A$, store a decimal $14$ in $59468$ to select
  the character set that has lowercase letters; and return.
- All other input characters are either ASCII control codes, for which there are
  no agreed-upon graphics, or else PET special graphics characters, so just
  return.

Examine the tables yourself to see if this algorithm will work.

\[
\begin{align*}
\text{FIXCHR} & \quad \text{AND} \ #7F \\
\text{SEC} & \\
\text{CMP} \ #40 & \\
\text{BCC} \ \text{FIXEND} & \\
\text{CMP} \ #60 & \\
\text{BCS} \ \text{LOWERC} & \\
\text{SBC} \ #40 & \\
\text{RTS} & \\
\text{LOWERC} & \quad \text{LDX} \ #14 \\
\text{STX} \ 59468 & \\
\text{FIXEND} & \quad \text{RTS}
\end{align*}
\]

Clear bit 7, so the character will be in
the legal ASCII range.
Prepare to compare.
If it’s less than $40$, return.
Okay. The character is greater than $40$.
Is it greater than $5F$?
If so, handle it as lowercase.
Okay. The character is in the range
$40$-$5F$.
Subtract $40$ for proper display code.
Since we have a lowercase letter, let’s
select the character set that
has lowercase letters.
Return, bearing PET display code for
character originally in accumulator.
Call FIXCHR with an ASCII character in the accumulator. FIXCHR will return with the corresponding PET display code in the accumulator. When it returns, its caller may store the accumulator anywhere in screen memory, thus displaying an image of the original ASCII character.

PET Keyboard Input Routine

To get an ASCII character from the PET keyboard, call the following subroutine:

```
PETKEY    JSR $FFE4
          CMP #0
          BEQ PETKEY

          AND #$7F
          RTS
```

Call PET ROM key scan routine. Zero means no key. If no key, scan again. A new key is in the accumulator. If the shift key was down, bit 7 is set. So clear bit 7, just to be sure we've got a legal ASCII character. Return with ASCII character in the accumulator.

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.

PET TVT Routine

To print an ASCII character to the screen, call $FFD2, a PET ROM routine I will refer to as PETTVT.

Any printable ASCII character passed to $FFD2 (or, apparently, to $E3EA or $F230) will be printed properly to the screen at the PET's current TVT screen location. You may change the PET's current TVT screen location (which is not the same as the current location used by the screen utilities in Chapter 5) by calling PETTVT with the accumulator holding any of the control codes from Table B2.1.
Table B2.1: Control codes that affect the next character to be printed by PETTVT.

<table>
<thead>
<tr>
<th>Character Name</th>
<th>Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURSOR NORTH</td>
<td>$91</td>
<td>Move current location up by one row.</td>
</tr>
<tr>
<td>CURSOR EAST</td>
<td>$1D</td>
<td>Move current location one column to the right.</td>
</tr>
<tr>
<td>CURSOR SOUTH</td>
<td>$11</td>
<td>Move current location down by one row.</td>
</tr>
<tr>
<td>CURSOR WEST</td>
<td>$9D</td>
<td>Move current location left by one column.</td>
</tr>
<tr>
<td>INSERT</td>
<td>$94</td>
<td>Move current character, and all characters to its right, one column to the right.</td>
</tr>
<tr>
<td>DELETE</td>
<td>$14</td>
<td>Move current character, and all characters to its right, one column to the left.</td>
</tr>
<tr>
<td>HOME</td>
<td>$13</td>
<td>Set current location to upper left of screen.</td>
</tr>
<tr>
<td>CLEAR</td>
<td>$93</td>
<td>Set current location to the upper left corner and clear the screen.</td>
</tr>
<tr>
<td>REVERSE</td>
<td>$12</td>
<td>Select reverse video for following characters.</td>
</tr>
<tr>
<td>REVERSE-OFF</td>
<td>$92</td>
<td>Select normal video mode for following characters.</td>
</tr>
</tbody>
</table>

These control codes may be passed directly to PETTVT, 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 the PETTVT routine. The PETTVT 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 PETTVT control codes, but it’s good to know they’re available, should you ever want your PET’s display screen to perform as something more than a glass teletype.

System Data Block

To run on a PET 2001, the software in this book requires the system data block shown in Appendices C14 and E13.
Setting the Top of Memory

Before you can use the BASIC OBJECT CODE LOADER (presented in Chapter 12) to load object code into your PET’s memory, you must insure that your PET’s BASIC interpreter leaves undisturbed all memory above $0FFF (4095 decimal). The PET BASIC interpreter will do as we wish if we set its top-of-memory pointer appropriately. The top-of-memory pointer specifies the highest address that may be used for the storage of BASIC program lines, variables, and strings. Memory above that address is off-limits to BASIC.

As you may know, there is more than one version of the PET 2001 by Commodore. Some PET’s have software in “old” ROMS (REV 2 ROMS), and others have software in “new” ROMS (REV 3 ROMS). As far as the software in this book is concerned, old ROM PETS and new ROM PETS are the same, since the ROM routines we care about are accessible from the same addresses in both old and new ROM PETS. Therefore, until now I haven’t even mentioned that the PET 2001 comes in two flavors. But now you must discover whether you have an old ROM or a new ROM PET, because otherwise you won’t be able to set the top of memory.

Old ROM and new ROM PETS each contain a machine-language subroutine to clear the screen, but in new ROM PETS that subroutine is at $E229 (57897 decimal), and in old ROM PETS that subroutine is as $E236 (57910 decimal). To see what ROMS are in your PET, use the PET’s screen editor to place some characters on the screen, and then type:

```
SYS (57897)
```

and press (RETURN). Does the screen clear? If so, you’ve got a new ROM PET. If not, turn off your PET, turn it on, place some characters on the screen, and then type:

```
SYS (57910)
```

and press (RETURN). Does the screen clear? If so, you’ve got an old ROM PET. If not, then your PET contains neither Rev 2 ROMS nor Rev 3 ROMS, and you’ll have to consult your system’s documentation carefully to discover the address of the top-of-memory pointer.

On old ROM PETS, the top-of-memory pointer is at 134 and 135 ($86,87). On new ROM PETS, the top-of-memory pointer is at 52 and 53 ($34,35). Regardless of the location of the top-of-memory pointer, we want to set the low byte of that pointer equal to $FF (255 decimal), and the high byte of that pointer equal to $0F (15 decimal), so that the pointer itself points to $0FFF. That will leave memory from
$1000 and up available to machine-language programs.

Thus, we set the top of memory on an old ROM PET with:

POKE 134,255:POKE 135,15

Similarly, we set the top of memory on a new ROM PET with:

POKE 34,255:POKE 35,15

Once you have set the top of memory available to your PET's BASIC interpreter, you may enter the BASIC OBJECT CODE LOADER and the DATA statements from Appendices E1 thru E11, and from Appendix E13. Remember to set the top of memory not only when typing in these DATA statements, but when RUNning the OBJECT CODE LOADER, as well.
Appendix B3:
The Apple II

Apple Display

The display memory of the Apple II is mapped in a manner that is much more complex than the Ohio Scientific or PET computers. On each of these other systems, only one portion of memory is mapped to the screen. The screen cannot display the contents of any other bank of memory (unless, of course, you copy the contents of another bank of memory into the display memory). But the Apple II may display the contents of any of four banks of memory: Low-Resolution Graphics and Text Page 1, Low-Resolution Graphics and Text Page 2, High-Resolution Graphics Page 1, and High-Resolution Graphics Page 2. Table B3.1 summarizes the locations of these pages in memory.

Table B3.1: Banks of display memory in the Apple II.

<table>
<thead>
<tr>
<th></th>
<th>Hexadecimal</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-Resolution Graphics and Text Page 1:</td>
<td>$0400-$07FF</td>
<td>1024-2043</td>
</tr>
<tr>
<td>Low-Resolution Graphics and Text Page 2:</td>
<td>$0800-$0BFF</td>
<td>2048-3071</td>
</tr>
<tr>
<td>Hi-Resolution Graphics Page 1:</td>
<td>$2000-$3FFF</td>
<td>8192-16383</td>
</tr>
<tr>
<td>Hi-Resolution Graphics Page 2:</td>
<td>$4000-$5FFF</td>
<td>16384-24575</td>
</tr>
</tbody>
</table>

Note that each of these display pages takes up much more than one hexadecimal page (256 bytes). A display page is simply an area of any size memory, whose contents may be displayed on the screen. Each low-res display page occupies four hexadecimal pages, and each hi-res display page occupies 32 hexadecimal pages. Why are the hi-res display pages bigger than the low-res display pages? Hi-res means high-resolution, and higher resolution requires more information.
How do you make the video screen show the contents of a given display page? You need only store a zero in a particular address. Certain addresses in the Apple II signal the video-display circuitry whenever data are written to them. The video-display circuitry responds to these signals by displaying the contents of a given bank of memory. These special addresses, or display selectors, are given in Table B3.2.

Table B3.2: Addresses that affect the APPLE II Display.

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Label</th>
<th>Purpose of Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C050</td>
<td>-16304</td>
<td>TXTCLR</td>
<td>Store a 0 here to set graphics mode.</td>
</tr>
<tr>
<td>$C051</td>
<td>-16303</td>
<td>TXTSET</td>
<td>Store a 0 here to set text mode.</td>
</tr>
<tr>
<td>$C052</td>
<td>-16302</td>
<td>MIXCLR</td>
<td>Store a 0 here to set bottom four lines to graphics.</td>
</tr>
<tr>
<td>$C053</td>
<td>-16301</td>
<td>MIXSET</td>
<td>Store a 0 here to select text/graphics mix (bottom four lines text).</td>
</tr>
<tr>
<td>$C055</td>
<td>-16299</td>
<td>HISCR</td>
<td>Store a 0 here to select Page 2.</td>
</tr>
<tr>
<td>$C056</td>
<td>-16298</td>
<td>LORES</td>
<td>Store a 0 here to select low-resolution graphics and text page.</td>
</tr>
<tr>
<td>$C057</td>
<td>-16297</td>
<td>HIRES</td>
<td>Store a 0 here to select high-resolution graphics.</td>
</tr>
</tbody>
</table>

Space limitations prohibit a discussion in this book of the power of high-resolution graphics. The Apple II documentation, however, provides an excellent step-by-step guide to the design, display, saving, and loading of high-resolution images. I must stress, however, that the software in this book expects the host system to have low-resolution graphics, so you'd better tell your Apple II to have low-resolution graphics. The software in this book uses the Apple's low-resolution graphics with text page 1 as the screen memory. To select this display page, simply press the RESET button on your Apple. If, on the other hand, you wish to select this display page under software control, you can do it by calling the subroutine LORES1:

```
LORES1      PHP
            PHA
            LDA # 0
            STA LOWSCR
            STA LORES
            PLA
            PLP
            RTS
```

Save processor flags.
Save accumulator.
Store a 0 in LOWSCR to select Page 1, and in LORES to select low-resolution graphics.
Restore accumulator.
Restore processor flags.
Return to caller.
This subroutine will select low-resolution graphics and text page 1. It preserves all flags and registers, and is completely relocatable.

Even when you’ve configured your Apple II to low-resolution graphics, your job isn’t done. The low-res display of the Apple II is mapped in an unusual manner. For any other system you can assume that the address of a given location on the screen is simply the address of the location above it, plus some row increment. On the Apple II this is not always true. See Table B3.3, Apple II low-res display memory map.

Table B3.3: Apple II low-resolution display.

<table>
<thead>
<tr>
<th>Row Number</th>
<th>Address of Leftmost Column</th>
<th>Address of Rightmost Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00</td>
<td>$400</td>
<td>$427</td>
</tr>
<tr>
<td>$01</td>
<td>$480</td>
<td>$4A7</td>
</tr>
<tr>
<td>$02</td>
<td>$500</td>
<td>$527</td>
</tr>
<tr>
<td>$03</td>
<td>$580</td>
<td>$5A7</td>
</tr>
<tr>
<td>$04</td>
<td>$600</td>
<td>$627</td>
</tr>
<tr>
<td>$05</td>
<td>$680</td>
<td>$6A7</td>
</tr>
<tr>
<td>$06</td>
<td>$700</td>
<td>$727</td>
</tr>
<tr>
<td>$07</td>
<td>$780</td>
<td>$7A7</td>
</tr>
<tr>
<td>$08</td>
<td>$428</td>
<td>$44F</td>
</tr>
<tr>
<td>$09</td>
<td>$4A8</td>
<td>$4CF</td>
</tr>
<tr>
<td>$0A</td>
<td>$528</td>
<td>$54F</td>
</tr>
<tr>
<td>$0B</td>
<td>$5A8</td>
<td>$5CF</td>
</tr>
<tr>
<td>$0C</td>
<td>$628</td>
<td>$64F</td>
</tr>
<tr>
<td>$0D</td>
<td>$6A8</td>
<td>$6CF</td>
</tr>
<tr>
<td>$0E</td>
<td>$728</td>
<td>$74F</td>
</tr>
<tr>
<td>$0F</td>
<td>$7A8</td>
<td>$7CF</td>
</tr>
<tr>
<td>$10</td>
<td>$450</td>
<td>$477</td>
</tr>
<tr>
<td>$11</td>
<td>$4D0</td>
<td>$4F7</td>
</tr>
<tr>
<td>$12</td>
<td>$550</td>
<td>$577</td>
</tr>
<tr>
<td>$13</td>
<td>$5D0</td>
<td>$5F7</td>
</tr>
<tr>
<td>$14</td>
<td>$650</td>
<td>$677</td>
</tr>
<tr>
<td>$15</td>
<td>$6D0</td>
<td>$6F7</td>
</tr>
<tr>
<td>$16</td>
<td>$750</td>
<td>$777</td>
</tr>
<tr>
<td>$17</td>
<td>$7D0</td>
<td>$7F7</td>
</tr>
<tr>
<td>Row Number</td>
<td>Address of Leftmost Column</td>
<td>Address of Rightmost Column</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>$00</td>
<td>$800</td>
<td>$827</td>
</tr>
<tr>
<td>$01</td>
<td>$880</td>
<td>$8A7</td>
</tr>
<tr>
<td>$02</td>
<td>$900</td>
<td>$927</td>
</tr>
<tr>
<td>$03</td>
<td>$980</td>
<td>$9A7</td>
</tr>
<tr>
<td>$04</td>
<td>$A00</td>
<td>$A27</td>
</tr>
<tr>
<td>$05</td>
<td>$A80</td>
<td>$AA7</td>
</tr>
<tr>
<td>$06</td>
<td>$B00</td>
<td>$B27</td>
</tr>
<tr>
<td>$07</td>
<td>$B80</td>
<td>$BA7</td>
</tr>
<tr>
<td>$08</td>
<td>$828</td>
<td>$84F</td>
</tr>
<tr>
<td>$09</td>
<td>$8A8</td>
<td>$8CF</td>
</tr>
<tr>
<td>$0A</td>
<td>$928</td>
<td>$94F</td>
</tr>
<tr>
<td>$0B</td>
<td>$9A8</td>
<td>$9CF</td>
</tr>
<tr>
<td>$0C</td>
<td>$A28</td>
<td>$ACF</td>
</tr>
<tr>
<td>$0D</td>
<td>$AA8</td>
<td>$B4F</td>
</tr>
<tr>
<td>$0E</td>
<td>$B28</td>
<td>$BCF</td>
</tr>
<tr>
<td>$0F</td>
<td>$BA8</td>
<td></td>
</tr>
<tr>
<td>$10</td>
<td>$850</td>
<td>$877</td>
</tr>
<tr>
<td>$11</td>
<td>$8D0</td>
<td>$8F7</td>
</tr>
<tr>
<td>$12</td>
<td>$950</td>
<td>$977</td>
</tr>
<tr>
<td>$13</td>
<td>$9D0</td>
<td>$9F7</td>
</tr>
<tr>
<td>$14</td>
<td>$A50</td>
<td>$A77</td>
</tr>
<tr>
<td>$15</td>
<td>$AD0</td>
<td>$AF7</td>
</tr>
<tr>
<td>$16</td>
<td>$B50</td>
<td>$B77</td>
</tr>
<tr>
<td>$17</td>
<td>$BD0</td>
<td>$BF7</td>
</tr>
</tbody>
</table>

Note that the display addresses do not increase uniformly as we move down, row-by-row, through low-res display page 1 or 2. The addresses increase uniformly from row 0 thru row 7, but from row 7 to row 8 the display addresses do not increase; they decrease! Then they increase uniformly through line $0F (15 decimal), but from line $0F to line $10 (15 to 16 decimal), the display address plummets again. Then from row $10 to row $17 (16 thru 23) the display addresses again increase uniformly.

If you'd like to take a visual tour of the Apple II's low-res display memory, run the BASIC program in listing B3.1. This program will simply poke a blank into each address in low-res display page 1, starting at the lowest address and moving to the highest address. You'll see that the screen does not fill with blanks in a contiguous manner, but follows a pattern of three interleaved parts.
Listing B3.1: APPLE II low-resolution display, memory-mapper program.

100    REM APPLE II LOW-RESOLUTION DISPLAY, MEMORY-MAPPER
105    REM
108    REM BY KEN SKIER
110    REM
120    FIRST = 1024: REM START OF LOW-RESOLUTION PAGE 1.
130    LAST = 2043: REM END OF LOW-RESOLUTION PAGE 1.
140    CHAR = 32: REM CHARACTER TO BE POKEED INTO SCREEN
150    REM WILL BE A WHITE BLANK.
160    REM
170    FOR X = FIRST TO LAST
175    REM FOR EACH ADDRESS IN LOW-RESOLUTION PAGE 1.
180    POKE X, CHAR
185    REM POKE A WHITE BLANK. THEN,
190    GOSUB 1000: REM WAIT A MOMENT...
200    NEXT X: REM BEFORE POKEING NEXT ADDRESS.
210    END
220    REM
230    REM
1000   FOR WAIT = 0 TO 100
1005   REM THIS IS A WAIT SUBROUTINE.
1010   NEXT WAIT: REM IT SLOWS DOWN PROGRAM SO YOU
1020   RETURN: REM CAN FOLLOW THE ACTION.

Must we now write a whole new set of display procedures to accommodate the unusual mapping of the Apple II low-res display pages? We could. But the screen utilities presented in Chapter 5 will work for the Apple II if we think of the Apple low-res screen as three separate screens: the top eight rows are one screen, the middle eight rows are another screen, and the bottom eight rows are a third screen. Each of these “screens” has a set of screen parameters.

The screen utilities in this book will work fine if you limit their scope to a given third of the screen. Use TVTOXY only to set a relative screen position within the third of the screen that you have selected. Use the screen utilities only for the top third of the screen. The middle and bottom thirds of the screen may still be used by the PRINT utilities.

To limit the screen utilities to the top third of low-res display page 1, initialize the screen parameters as follows:

SCREEN .WORD $0400
TVCOLS .BYTE $27
TVROWS .BYTE $07
ROWINC .BYTE $80
If you want to keep text from scrolling into the upper third of the screen, store $08 in address $0022. (In BASIC you may do this with the command POKE 34,8.)

There's one more quirk to the Apple display. If you store an ASCII character in display memory, then you will display a blinking or inverse version of the character. Setting bit 7 in an ASCII character code will cause that character to be displayed in normal mode (a white character on a black background), rather than as a black character on a white background or as a blinking character.

You may experiment with this feature of the Apple II by using the Apple II monitor to store $41 (an ASCII "A") in a location in low-res display page 1. You'll see a blinking "A." Now store $C1 in a location in low-res display page 1. You'll see a normal "A." Why? Because $C1 is $41 with bit 7 set. To understand what's happening here, look at the Apple II's character set given in Table B3.4.

**Table B3.4: The Apple II character set.**

<table>
<thead>
<tr>
<th>RIGHT NYBBLE OF CHARACTER</th>
<th>LEFT NYBBLE OF CHARACTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -A -B -C -D -E -F</td>
<td>0 1 2 3 4 5 6 7 8 9 ; &lt; = &gt; ?</td>
</tr>
<tr>
<td>@ A B C D E F G H I J K L M N O</td>
<td>P Q R S T U V W X Y Z [ \ ]</td>
</tr>
<tr>
<td>1-</td>
<td></td>
</tr>
<tr>
<td># $ % ' ( ) * + , - . /</td>
<td></td>
</tr>
<tr>
<td>2-</td>
<td></td>
</tr>
<tr>
<td>3-</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 ; &lt; = &gt; ?</td>
<td></td>
</tr>
</tbody>
</table>

The Apple II really has only 64 characters in its character set, but it has four ways of displaying each character. Thus, the table shows a set of characters at $00 thru $3F; the same characters, in the same sequence, appear again at $40 thru $7F, at $80 thru $BF, and at $C0 thru $FF. These represent what I call the first, the second, the third, and the fourth quadrants of the character set.
Character codes in this first quadrant ($00 thru $3F) will be displayed in reverse video: as black characters on a white background. Character codes in the second quadrant ($40 thru $7F) will be displayed in a blinking mode. Character codes in the third and fourth quadrants ($80 thru $BF and $C0 thru $FF) will be displayed in normal mode: as white characters on black background.

Before we store any ASCII character in screen memory, we must first call FIXCHR, to convert, if necessary, the ASCII character to the host system's corresponding display code. In the Apple II, FIXCHR is very simple:

\begin{verbatim}
FIXCHR     ORA #$80
RTS
\end{verbatim}

Set bit 7, so character will be displayed in normal mode.

Return appropriate display code to caller.

I/O Vectors

The Apple II has a subroutine in read-only memory to get a character from the keyboard, and another subroutine to print a character on the screen. However, the key-in routine at $FD35 does not return an ASCII code when you press the key for an ASCII character; instead, it returns the appropriate ASCII code with bit 7 set. Similarly, the screen-printing routine at $FBFD will print an ASCII character to the screen, but the character will be in reverse video or blinking. In order to print an ASCII character to the screen, you must first set bit 7 and then call $FBFD. Conversely, to get an ASCII character from the keyboard, you must first call $FD35 and then clear bit 7. Therefore, the following patches are offered:

**Subroutine to Print an ASCII Character to Apple II Screen**

\begin{verbatim}
APLTVT     ORA #$80
           JSR $FBFD
           RTS
\end{verbatim}

Set bit 7 in the ASCII code.

Call the ROM screen printer.

Return to caller, now that ASCII character originally in accumulator has been printed to screen in normal mode.

**Subroutine to Get an ASCII Character from Apple II Keyboard**

\begin{verbatim}
APLKEY     JSR $FD0C
\end{verbatim}

Get ASCII character from keyboard with bit 7 set. (Note: you may call $FD35 instead of calling $FD0C.)
ORA $80

Clear bit 7, leaving the accumulator holding a conventional ASCII code.

RTS

Return to caller, bearing ASCII character code for depressed key.

Apple II System Data Block

The I/O vectors ROMTVT and ROMKEY should be initialized to point to APLTVT and APLKEY, respectively. This has been done in the Apple II system data block. You must enter the Apple II system data block into your system's memory if any of the software in this book is to run on your Apple II. See Appendices C15 and E14.
Appendix B4:
The Atari 800

Screen

The Atari 800 microcomputer has the most flexible — and, perhaps the most confusing — video-display hardware of any system discussed in this book. Unlike the other systems, almost any portion of the Atari computer's memory may be mapped to the screen. Furthermore, there are many different screen-display modes. When the Atari computer is powered-up, the screen is in text mode zero. That's comparable to the Apple II's low-resolution graphics and text display, which is comparable to the only video-display mode available on the Ohio Scientific or PET computers.

The Atari computer makes other screen modes available to the programmer, but the software in this book assumes a low-resolution text display, so you'd better leave your Atari in screen mode zero if you expect to see any of the displays driven by the software in this book. In other words, if you change the screen mode, the Visible Monitor may well become invisible.

I mentioned that the screen buffer may be almost anywhere in memory. If that's true (and it is), how can you determine the HOME address upon which all the displays in this book are based? It's easy. A pointer at $58,$59 (88,89 decimal) points to the lowest address in screen memory: the address we refer to as HOME. Before running any of the software in this book, you must set HOME properly for your system. Simply set HOME equal to the value of that pointer. HI_PAGE, the value of the highest page in screen memory, is equal to (the high byte of HOME) plus three.

Once we've set HOME and HI_PAGE properly, we're home free. The other screen parameters are fixed:

ROWINC .BYTE 40
TVCOLS .BYTE 39
TVROWS .BYTE 23
SPACE .BYTE $20
ARROW .BYTE $7B
Note that the top of screen memory is always at the top of programmable memory, so if you add more programmable memory to your Atari 800, you'll move the screen memory up higher in the address space.

Proper Display of ASCII Characters

Like the PET, and to a lesser extent the APPLE II, the Atari screen requires that we perform a conversion before we can properly display an ASCII character on the screen. To determine the nature of this conversion, let us first look at the ATARI character set in Table B4.1.

Table B4.1: The Atari character set ATASCII.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>space</td>
<td>!</td>
<td>&quot;</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
<td>'</td>
<td>(</td>
<td>)</td>
<td>*</td>
<td>+</td>
<td>,</td>
<td>-</td>
<td>.</td>
<td>/</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>:</td>
<td>;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
</tr>
<tr>
<td>2</td>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>[</td>
<td>\</td>
<td>]</td>
<td>←</td>
</tr>
<tr>
<td>4</td>
<td>special graphics characters</td>
<td>←</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>special graphics characters</td>
<td>←</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td>o</td>
</tr>
<tr>
<td>7</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>←</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A quick examination shows that ASCII characters $20$ thru $5F$ are ATASCII (Atari's character set) characters $00$ thru $3F$. Thus, if an ASCII character is in the range of $20$ thru $5F$, we can convert it to the appropriate ATASCII character simply by subtracting $20$.

Further inspection reveals that ASCII characters $61$ thru $7A$ correspond to ATASCII characters $61$ through $7A$. Thus, if an ASCII character is in the range of $61$ thru $7A$, it needs no conversion to ATASCII; it already is the corresponding ATASCII character.

Finally, if an ASCII character is not in the range $20$ thru $5F$ or $61$ thru $7A$, it's not a printable character and has no agreed-upon graphic representation. For those cases we'll just leave them alone.

Figure B4.1 flow-charts this algorithm.
Figure B4.1: Flowchart of routine to convert an ASCII character for display on Atari screen.

Using the flowchart in figure B4.1 as a guide, we can write source code for FIXCHR, which takes an ASCII character as input and returns an Atari display code so that the character may be properly displayed on the video screen.

**FIXCHR**

```
FIXCHR     AND #$7F
SEC
CMP #$20
```

Clear bit 7 so character is a legitimate ASCII character.
Prepare to compare.
Character less than $20?
BCC BADCHR

CMP #60
BCC SUBS$20
CMP #$7B
BCC EXIT

BADCHR LDA BLANK

EXIT RTS
SUB$20 SBC #$20
RTS

Keyboard Input

If no key has been pressed, then address $02FC (764 decimal) contains $FF. But whenever you depress a key on the Atari keyboard — even if a program is not scanning the keys — an electronic circuit will sense that a key has closed and will store the hardware code for that key in address $02FC. However, the code in $02FC will be a hardware code, not obviously related to ASCII or ATASCII.

Table B4.2: Atari Hardware Key-Codes.

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Key</th>
<th>Hex</th>
<th>Decimal</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00</td>
<td>0</td>
<td>L</td>
<td>$20</td>
<td>32</td>
<td>.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>J</td>
<td>1</td>
<td>33</td>
<td>SPACE</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>;</td>
<td>2</td>
<td>34</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>N</td>
<td>3</td>
<td>35</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>36</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>K</td>
<td>5</td>
<td>37</td>
<td>M</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>+</td>
<td>6</td>
<td>38</td>
<td>/</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>*</td>
<td>7</td>
<td>39</td>
<td>ATARI</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>40</td>
<td>R</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>P</td>
<td>A</td>
<td>42</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>U</td>
<td>B</td>
<td>43</td>
<td>Y</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>RETURN</td>
<td>C</td>
<td>44</td>
<td>TAB</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>I</td>
<td>D</td>
<td>45</td>
<td>T</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>-</td>
<td>E</td>
<td>46</td>
<td>W</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>=</td>
<td>F</td>
<td>47</td>
<td>Q</td>
</tr>
</tbody>
</table>
The Hex and Decimal Columns give the low 6 bits of the hardware key-code stored in address $02FC (764 decimal) when the given keys are pressed. Either SHIFT key sets bit 6. CTRL key sets bit 7.

In order to convert that hardware code to ASCII, we need to understand its nature. The six low-order bits of the hardware key-code uniquely identify the key. (See Table B4.2.) Bits 6 and 7 identify its shift state. Bit 6 is set if the key is typewriter-shifted; bit 7 is set if the key is control-shifted. The key is typewriter-shifted if either SHIFT key is down; the CAPS/LOWR key has no effect on the typewriter-shift state as reflected in the hardware key-code. The keyboard is control-shifted if the CTRL key is down.

If you don’t care about the keyboard’s shift state, but merely want to determine which physical key has been pressed, then you can clear the two high-order bits in the hardware key-code and you’ll be left with a number from 0 to 63 decimal (00 to $3F) uniquely identifying the key most recently depressed. If you care about the keyboard’s typewriter-shift state but are indifferent to its control-shift state, then you can clear bit 7 in the hardware key-code and you’ll be left with a number from 0 to 127 decimal (00 to $7F), which means the keyboard can generate twice as many characters as it has physical keys. To enable control-shifting, simply preserve the hardware key-code, and you double once again the number of characters that the keyboard (and hence the user) may generate.

Since the simple text editor presented in Chapter 11 assigns certain functions to control-shifted keys, and since you never know when you might need some additional character codes from your keyboard, Appendix C16 presents a key-handling subroutine for the Atari. This subroutine is capable of generating different
characters in each of the four different shift-states (unshifted, typewriter-shifted, control-shifted, typewriter- and control-shifted).

It's a simple matter to use the eight-bit hardware keycode as an index into a keyboard definition table. For any given hardware key-code, we may assign any character we like. The keyboard definition table presented in Appendix C16 assigns standard ASCII characters to all letter, number, and punctuation keys, in both the unshifted and typewriter-shifted states. Other keys are assigned values consistent with their expected use by the software in this book (eg: Control-P generates a $10, thus making it a PRINT key in the eyes of the simple text editor). All keys and shift states that have no special meaning to this software have been assigned character codes of zero; feel free to change these character codes to any values you desire.

Assuming that we have in memory a keyboard definition table called ATRKYS, we can get an ASCII character from the Atari keyboard with the following subroutine, ATRKEY:

```
ATRKEY   LDA $02FC   Has a key been depressed?
         CMP #$FF     $FF means no key.
         BEQ ATRKEY   If not, look again. A key has gone down
         TAY          and the accumulator holds its hardware
         LDA ATRKYS,Y key-code.
         RTS          Prepare to use that code as an index.
                  Look up character for that key and shift
                  state.
                  Return with ASCII character
                  corresponding to that key and shift
                  state.
```

Print a Character to the Screen

The Atari 400 and 800 computers each provide a powerful I/O (input/output) routine which allows the programmer to get characters from virtually any source, and to send characters to virtually any device — the screen, the printer, the cassette recorder, and the disk. But, as in the case of Atari's varied screen modes, power breeds complexity. I have found it easier to substitute my own simple routine to print a character on the TV screen, bypassing the Atari I/O routines entirely.

Incidentally, this routine will work with any 6502-based computer that has a low-resolution memory-mapped display. If you need a simple TVT simulator for your home-brew 6502-based system with a video display, TVTSIM might meet your needs. In any event, it prints characters to the screen, and avoids the necessity of plumbing the depths of the many modes and data structures associated with Atari's central I/O routine.
With your system data block initialized as shown in Appendices C16 and E15 (which includes the TVT simulator as the subroutine to print characters to the screen), you are almost ready to run the software in this book on your own system.

Setting the Top Of Memory

Address $2E6 (742 decimal) holds the number of pages of RAM available to the BASIC interpreter. Store a $0D (13 decimal) in that location and BASIC will use memory up to $0DFF, but will not use $0E00 and up.

NOTE: On the Atari, the software in this book uses memory from $0E80 to $1FFF, which is the address space required by the ATARI DOS (Disk Operating System) and the ATARI RS-232 serial interface, so you may not use DOS or RS-232 if you expect to use the software in this book. However, there should be no conflict between software in this book and the cassette-based Atari 800.

Thus, we may set the top of memory with the following BASIC command:

POKE 742,13

When you have used the OBJECT CODE LOADER to READ and POKE object code from all the appropriate E appendices into your Atari computer, run the following BASIC program. It will initialize screen parameters and the top of memory, and then pass control to the Visible Monitor.

```
100 REM Visible Monitor Start-Up Program for the Atari.
110 REM
120 REM First, set the screen parameters.
130 REM
140 REM A pointer at 88,89 points to lowest screen address.
150 LO=PEEK(88): REM Set LO to the low byte of HOME.
160 HI=PEEK(89): REM Set HI to the high byte of HOME.
165 IF HI < 32 THEN PRINT "ON AN 8 K ATARI YOU MAY NOT USE EDITOR OR DISASSEMBLER"
170 POKE 4096,LO: REM Set Low byte of HOME.
180 POKE 4097,HI: REM Set High byte of HOME.
190 POKE 4101,HI+3: REM Set HIPAGE = Highest page in screen memory.
200 REM
210 REM Now set the top of memory available to BASIC.
220 POKE 742,13: Tell BASIC to use only memory up to $0DFF.
230 REM
240 REM Now call the Visible Monitor.
250 X=USR(4615): REM Call the Visible Monitor as a subroutine.
260 END
```
Appendix C1:
Screen Utilities
APPENDIX C1: ASSEMBLER LISTING OF
SCREEN UTILITIES

SEE CHAPTER 5 OF BEYOND GAMES: SYSTEMS
SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

BY KEN SKIER

ZERO PAGE BYTES

TV.PTR=0 THIS POINTER HOLDS THE
ADDRESS OF THE CURRENT
SCREEN LOCATION.

SCREEN PARAMETERS

PARAMS=$1000 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.

TUCOLS=PARAMS+3
TUCOLS IS A BYTE GIVING
NUMBER OF COLUMNS ON SCREEN.
(COUNTING FROM ZERO.)

TUROWS=PARAMS+4
TUROWS 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.

CLEAR SCREEN
1170 ; CLEAR SCREEN, PRESERVING THE ZERO PAGE.
1180 ;
1190 ;
1200 ;
1210 ;
1220 1100 20c411 clr.tv jsr tvpush save zero page bytes that
1230 ; will be changed.
1240 1103 202b11 jsr tvhome set screen location to upper
1250 ; left corner of the screen.
1260 1108 ae03 10 ldx tucols load x,y registers with
1270 1109 ac04 10 ldy turows x,y dimensions of screen.
1280 110c 2031 11 jsr clr.xy clear x columns, y rows
1290 ; from current screen location.
1300 110f 20d3 11 jsr tv.pop restore zero page bytes that
1310 ; were changed.
1320 1112 60 rts return to caller, with zero
1330 ; page preserved.
1340 ;
1350 ;
1360 ;
1370 ;
1380 ;
1390 ;
1400 ;
1410 ;
1420 ;
1430 ;
1440 ;
1450 ;
1460 ;
1470 ;
1480 ;
1490 ;
1500 ;
1510 ;
1520 ;
1530 ;
1540 ;
1550 ;
1560 ;
1570 ;
1580 ;
1590 ;
1600 1113 be2a 11 clr.xy stx cols set the number of columns
1610 ; to be cleared.
1620 1116 90 tya
1630 1117 aa tax now x holds number of rows
1640 ; to be cleared.
1650 ;
1660 1118 ad06 10 clrrow lda blank
1670 ; we'll clear them by
1680 ; writing blanks to the
1690 ; screen.
1700 ;
1710 111b ac2a 11 ldy cols load y with number of
1720 ; columns to be cleared.
1730 111e 9100 clrpos sta (tv.ptr),y clear a position by
1740 ; writing a blank into it.
1750 ;
1760 1120 88 dey adjust index for next
1750 ; POSITION ON THE ROW.
1760 ;
1770 1121 10FB BPL CLRPOS IF NOT DONE WITH ROW,
1780 ; CLEAR NEXT POSITION...
1790 ;
1800 1123 207511 JSR TVDOWN IF DONE WITH ROW, MOVE
1810 ; CURRENT SCREEN LOCATION
1820 ; DOWN BY ONE ROW.
1830 ;
1840 1126 CA DEX DONE LAST ROW YET?
1850 1127 10EF BPL CLRROW IF NOT, CLEAR NEXT ROW...
1860 1129 60 RTS IF SO, RETURN TO CALLER.
1870 ;
1880 112A 00 COLS .BYTE 0 DATA CELL: HOLDS NUMBER OF
1890 ; COLUMNS TO BE CLEARED.
1900 ;
1910 ;
1920 ;
1930 ;
1940 ;
1950 ;
1960 ;
1970 ;
1980 ;
1990 ;
2000 ; ****************************
2010 ;
2020 ; TVHOME
2030 ;
2040 ; ****************************
2050 ;
2060 ;
2070 ;
2080 ;
2090 ;
2100 112B A200 TVHOME LDX $0 SET TV.PTR TO UPPER LEFT
2110 112D A000 LDY $0 CORNER OF SCREEN, BY
2120 ; ZEROING X AND Y AND THEN
2130 112F 18 CLC GOING TO X,Y COORDINATES:
2140 1130 900A BCC TVTOXY
2150 ;
2160 ;
2170 ;
2180 ;
2190 ; ****************************
2200 ; CENTER
2210 ;
2220 ; ****************************
2230 ;
2240 ;
2250 ;
2260 ;
2270 ;
2280 ;
2290 ;
2300 ;
2310 ;
2320 ;

216 BEYOND GAMES
2330 ;
2340 1132 AD0410 CENTER LDA TUROWS
2350 1135 4A LSR A
2360 1136 4B TAY
2370 ;
2380 ;
2390 1137 AD0310 LDA TUCOLS
2400 113A 4A LSR A
2410 113B 4A TAX
2420 ;
2430 ;
2440 ;
2450 ;
2460 ;
2470 ;
2480 ;
2490 ;
2500 ;
2510 ;
2520 ;
2530 ;
2540 ;
2550 ;
2560 ;
2570 ;
2580 ;
2590 ;
2600 ;
2610 ;
2620 ;
2630 ;
2640 ;
2650 ;
2660 ;
2670 ;
2680 ;
2690 ;
2700 ;
2710 113C 3B TVOTOXY SEC
2720 ;
2730 ;
2740 ;
2750 113D EC0310 CPX TUCOLS
2760 1140 5003 BCC X.OK
2770 ;
2780 1142 AE0310 LDX TUCOLS
2790 ;
2800 ;
2810 1145 38 X.OK SEC
2820 1146 CC0410 CPY TUROWS
2830 1149 9003 BCC Y.OK
2840 ;
2850 ;
2860 114B AC0410 LDY TUROWS
2870 ;
2880 ;
2890 ;
2900 114E AD0010 Y.OK LDA HOME
2910 ;

LOAD A WITH TOTAL ROWS.
DIVIDE IT BY TWO.
Y NOW HOLDS THE NUMBER OF
THE SCREEN'S CENTRAL ROW.

LOAD A WITH TOTAL COLUMNS.
DIVIDE IT BY TWO.
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:

*****************************************************************************

TUTOXY
*****************************************************************************

*****************************************************************************

SET CURRENT SCREEN LOCATION
TO COORDINATES GIVEN BY
THE X AND Y REGISTERS.

IS X OUT OF RANGE?
IF NOT, LEAVE IT ALONE.
IF X IS OUT OF RANGE, GIVE
IT ITS HIGHEST LEGAL VALUE.
NOW X IS LEGAL.

IS Y OUT OF RANGE?
IF NOT, LEAVE IT ALONE.
IF Y IS OUT OF RANGE, GIVE
Y ITS HIGHEST LEGAL VALUE.
NOW Y IS LEGAL.

SET TV.PTR = LOWEST SCREEN
2910 1151 8500 STA TV.PTR ADDRESS.
2920 1153 AD0110 LDA HOME+1
2930 1156 8501 STA TV.PTR+1
2940 ;
2950 1158 0B PHP SAVE CALLER'S DECIMAL FLAG.
2960 1159 0B CLD CLEAR DECIMAL FOR BINARY
2970 ; ADDITION.
2980 ;
2990 115A 0A TXA ADD X TO TV.PTR
3000 115B 18 CLC
3010 115C 8500 ADC TV.PTR
3020 115E 9003 BCC COLSET
3030 1160 EE01 INC TV.PTR+1
3040 1162 18 CLC
3050 ;
3060 ;
3070 1163 C000 COLSET CPY $0 ADD Y ROWING TO TV.PTR:
3080 1165 F000 BEQ TV.SET
3090 1167 18 ADDROW CLC
3100 1168 6D0210 ADC ROWING
3110 116B 9002 BCC ++4
3120 116D EE01 INC TV.PTR+1
3130 116F 0B DEY
3140 1170 D0F5 BNE ADDROW
3150 ;
3160 ;
3170 1172 8500 TV.SET STA TV.PTR RESTORE CALLER'S DECIMAL FLAG
3180 1174 28 PLP
3190 1175 60 RTS RETURN TO CALLER
3200 ;
3210 ;
3220 ;
3230 ;
3240 ;
3250 ;
3260 ;
3270 ;
3280 ;
3290 ;
3300 ;**************************************************************************
3310 ; TUDOWN, TVSKIP, and TUPLUS
3320 ;**************************************************************************
3330 ;
3340 ;
3350 ;
3360 ;
3370 ;
3380 ;
3390 ;
3400 1176 AD0210 TUDOWN LDA ROWING MOVE TV.PTR DOWN BY ONE ROW.
3410 1179 18 CLC
3420 117A 9005 BCC TUPLUS
3430 ;
3440 117C 203B11 VUCHAR JSR TV.PUT PUT CHARACTER ON SCREEN
3450 ; AND THEN
3460 ;
3470 117F A901 TVSKIP LDA #1 SKIP ONE SCREEN LOCATION
3480 ; BY INCREMENTING TV.PTR

218 BEYOND GAMES
3490  ;
3500  ;
3510 1181 08  TUPLUS PHP  TUPLUS ADDS ACCUMULATOR
3520 1182 08  CLD  TO TV.PTR, KEEPING TV.PTR
3530 1183 18  CLC  WITHIN SCREEN MEMORY.
3540 1194 6500  ADC TV.PTR
3550 1195 9002  BCC *+4
3560 1196 6501  INC TV.PTR+1
3570 1197 8500  STA TV.PTR
3580 1198 38  SEC  IS CURRENT SCREEN LOCATION
3590 1199 90510  LDA HIPAGE  OUTSIDE OF SCREEN MEMORY?
3600 119A C501  CMP TV.PTR+1
3610 119B 6005  BCS TV.OK
3620  ;
3630 119C AD010  LDA HOME+1  IF SO, WRAP AROUND FROM
3640 119D 6501  STA TV.PTR+1  BOTTOM TO TOP OF SCREEN.
3650  ;
3660 119E 29  TV.OK  PLP  RESTORE ORIGINAL DECIMAL
3670 119F 60  RTS  FLAG AND RETURN TO CALLER.
3680  ;
3690  ;
3700  ;
3710  ;
3720  ;
3730  ;
3740  ;
3750  ;
3760  ;
3770  ;
3780  ;
3790  ;
3800  ;
3810  ;
3820  ;
3830  ;
3840  ;
3850  ;
3860  ;
3870  ;
3880  ;
3890 119B 201110  TV.PUT JSR FIXCHR  CONVERT ASCII CHARACTER
3900  ;  TO YOUR SYSTEM'S DISPLAY
3910  ;  CODE.
3920  ;
3930 119C A000  LDY $0  PUT CHARACTER AT CURRENT
3940 119D 9100  STA (TV.PTR),Y SCREEN LOCATION.
3950 119E 60  RTS  THEN RETURN.
3960  ;
3970  ;
3980  ;
3990  ;
4000  ;
4010  ;
4020  ;
4030  ;
4040  ;
4050  ;
4060  ;

219
DISPLAY A BYTE IN HEX FORMAT

**------------------------------------------------------------------------**

4070
4080
4090
4100  ;
4110
4120
4130
4140
4150 11A3 48  ;
4160 11A4 4A  ;
4170 11A5 4A  ;
4180 11A6 4A  ;
4190 11A7 4A  ;
4200
4210 11AB 20B611 ;
4220
4230
4240 11AB 207C11 ;
4250
4260
4270
4280 11AE 68
4290 11AF 20B611 ;
4300
4310
4320 11B2 207C11 ;
4330
4340
4350
4360
4370
4380 11B5 60
4390
4400
4410
4420
4430
4440
4450
4460
4470
4480
4490
4500
4510
4520
4530
4540
4550
4560
4570
4580
4590 11B5 09
4600 11B7 D8
4610 11BB 200F
4620 11BA C00A
4630
4640 11BC 3002

**------------------------------------------------------------------------**

.ASCII PHP

**------------------------------------------------------------------------**

.HEX-TO-ASCII

4590 11B5 09
4600 11B7 D8
4610 11BB 200F
4620 11BA C00A
4630
4640 11BC 3002

.this routine returns ASCII
for 4 LSB in accumulator.
clear high 4 bits in A.
is accumulator greater
than 9?

220 BEYOND GAMES
IF $00, IT MUST BE A-F.
ADD $36 HEX TO CONVERT IT.
TO CORRESPONDING ASCII CHAR.
IF A IS 0-9, ADD $30 HEX
TO CONVERT IT TO
CORRESPONDING ASCII CHAR.

PLP
RESTORE ORIGINAL DECIMAL
FLAG, AND
RETURN TO CALLER.

***********

TVPUSH

***********

SAVE CURRENT SCREEN LOCATION
ON STACK, FOR CALLER.

TVPUSH PLA
PULL RETURN ADDRESS FROM
STACK AND SAVE IT IN X AND
Y REGISTERS.

LDA TV.PTR+1
GET TV.PTR AND
PHA

LDA TV.PTR
PUSH IT ONTO THE STACK.
PHA

TYA
PLACE RETURN ADDRESS
PHA

TXA
BACK ON STACK.
PHA

RTS
THEN RETURN TO CALLER.
CALLER WILL FIND TV.PTR ON STACK, LOW BYTE ON TOP.

TV.POP

RESTORE SCREEN LOCATION PREVIOUSLY SAVED ON STACK.

FULL RETURN ADDRESS FROM STACK, SAVING IT IN X...

...AND IN Y

RESTORE...

...TV.PTR FROM STACK.

PLACE RETURN ADDRESS BACK...

...ON STACK.

RETURN TO CALLER.
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 BEYOND GAMES: SYSTEMS
SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

BY KEN SKIER

; ************************************************************************
; EQUATES
; ************************************************************************

300 0000=   TV.PTR = 0
300
300 0002=   GETPTR = 2
310
320
330
340
350
360
370

400 0000=   PARAMS = $1000 ADDRESS OF SYSTEM DATA
410   BLOCK.
420
430 1000=   ARROW = PARAMS+7
440   THIS DATA BYTE HOLDS YOUR
450   SYSTEM’S CHARACTER CODE
460   FOR AN UP-ARROW.
470 1000=   ROMKEY = PARAMS+8
480   ROMKEY IS A POINTER TO
490   YOUR SYSTEM’S SUBROUTINE
500   TO GET AN ASCII CHARACTER
510   FROM THE KEYBOARD.
520 1000=   SPACE = $20
530
540
550
560
570
580 0020=   
590
590 ; RUBOUT = $7F
600 007F= ; CR = $0D
610 ; ASCII FOR CARRIAGE RETURN.
620 000D= ;
630 ;
640 ;
650 ;
660 ;
670 ;
680 ;
690 ;
700 ;
710 ;
720 ;
730 ;
740 ;
750 ;
760 ;
770 ;
780 ;
790 ;
800 ;
810 ;
820 ;
830 1100= TVSUBS = $1100
840 1100= CLR.TV = TVSUBS
850 1113= CLR.XY = TVSUBS+$13
860 112B= TUHOME = TVSUBS+$2B
870 113C= TUTOXY = TVSUBS+$3C
880 1176= TUDOWN = TVSUBS+$76
890 117C= VUCHAR = TVSUBS+$7C
900 117F= TUSKIP = TVSUBS+$7F
910 1181= TVPLUS = TVSUBS+$81
920 11A3= VUBYTE = TVSUBS+$A3
930 11B6= ASCII = TVSUBS+$B6
940 11C4= TVPUSH = TVSUBS+$C4
950 11D3= TV.POP = TVSUBS+$D3
960 ;
970 ;
980 ;
990 1200= * = $1200
1000 ;
1010 ;
1020 ;
1030 12E3= UPDATE = *+$E3
1040 ;
1050 ;
1060 ;
1070 ;
1080 ;
1090 ;
1100 ;
1110 ;
1120 ;
1130 ;
1140 ;
1150 ;
1160 ;
; USER-MODIFIABLE DATA

1270 1200 00 FIELD .BYTE 0  NUMBER OF CURRENT FIELD.
1280 ; (MUST BE 0-6.)
1290 ;
1300 1281 00 REG.A .BYTE 0  IMAGE OF ACCUMULATOR.
1310 ;
1320 1282 00 REG.X .BYTE 0  IMAGE OF X-REGISTER.
1330 ;
1340 1283 00 REG.Y .BYTE 0  IMAGE OF Y-REGISTER.
1350 ;
1360 1284 00 REG.P .BYTE 0  IMAGE OF PROCESSOR STATUS
1370 ;
1380 ;
1390 1281-  REGS = REG.A
1400 ;
1410 1285 0000 SELECT .WORD 0  POINTER TO CURRENTLY-
1420 ; SELECTED ADDRESS.
1430 ;
1440 ;
1450 ;
1460 ;
1470 ;
1480 ;
1490 ;
1500 ;
1510 ;
1520 ;
1530 ;
1540 ;
1550 ;
1560 ;
1570 ;
1580 ;
1590 ;
1600 1207 03 VISMON PHP  SAVE CALLER'S STATUS FLAGS.
1610 1206 03 CLD  CLEAR DEcimal MODE, SINCE
1620 ; ARITHMETIC OPERATIONS IN THIS
1630 ; BOOK ARE ALWAYS BINARY.
1640 ;
1650 1209 201212 JSR DISPLAY  PUT MONITOR DISPLAY ON
1660 ; SCREEN.
1670 ;
1680 120C 20E312 JSR UPDATE  GET USER REQUEST AND
1690 ; HANDLE IT.
1700 120F 18 CLC
1710 1210 80F6 BCC VISMON+1 LOOP BACK TO DISPLAY...
1750 ;
1760 ;
1770 ;
1780 ;
1790 ;
1800 ;
1810 ;
1820 ;
1830 ;
1840 ;
1850 ;
1860 ;
1870 ;
1880 ;
1890 ;
1900 ;
1910 1212 20C411 DISPLAY JSR TVPUSH SAVE ZERO PAGE BYTES THAT
1920 ; WILL BE MODIFIED.
1930 ;
1940 1215 202512 JSR CLRMON CLEAR A PORTION OF SCREEN.
1950 1218 203412 JSR LINE.1 DISPLAY LABEL LINE.
1960 121B 2065C12 JSR LINE.2 DISPLAY DATA LINE.
1970 121E 20AF12 JSR LINE.3 DISPLAY ARROW LINE.
1980 ;
1990 1221 20D311 JSR TV.Pop RESTORE ZERO PAGE BYTES
2000 ; THAT WERE SAVED ABOVE.
2010 ;
2020 1224 60 RTS RETURN TO CALLER.
2030 ;
2040 ;
2050 ;
2060 ;
2070 ;
2080 ;
2090 ;
2100 ;
2110 ;
2120 ;
2130 ; CLEAR PORTION OF SCREEN
2140 ;
2150 ;
2160 ;
2170 ;
2180 ;
2190 ;
2200 ;
2210 ;
2220 ;
2230 1225 A202 CLRMON LDX $2 SET TV.PTR TO COLUMN 2,
2240 1227 A002 LDY $2 ROW 2.
2250 1229 203C11 JSR TVTOXY
2260 ;
2270 122C A219 LDX $25 LOAD X WITH NUMBER OF
2280 ; COLUMNS (25) TO BE CLEARED.
2290 ;
2300 122E A003 LDY $3 LOAD Y WITH NUMBER OF
2310 ; ROWS (3) TO BE CLEARED.
2320 ;
JSR CLR.XY  CLEAR X COLUMNS, Y ROWS.
RTS  RETURN TO CALLER.

**********************************************************************
DISPLAY LABEL LINE
**********************************************************************

X-COORDINATE OF LABEL "A".
Y-COORDINATE OF LABEL "A".
SET TV.PTR TO POINT TO SCREEN LOCATION OF LABEL "A"

PUT LABELS ON SCREEN.
INITIALIZE LABEL COLUMN COUNTER.

GET A CHARACTER AND PUT IT ON THE SCREEN.
PREPARE FOR NEXT CHARACTER.
DONE LAST CHARACTER?

IF NOT, DO NEXT CHARACTER.
RETURN TO CALLER.
DATA CELL: HOLDS COLUMN OF CHARACTER TO BE COPIED.

LABELS .BYTE 'A X Y P'
2820: 125C A20Z LINE.Z LDX #2 LOAD X WITH STARTING
2830: 010 ; COLUMN OF DATA LINE.
2840: 020 ;
2850: 030 125E A003 LDY #3 LOAD Y WITH ROW NUMBER
2860: 040 ; OF DATA LINE.
2870: 050 ;
2880: 060 1260 203C11 JSR TVTOXY SET TV.PTR TO POINT TO
2890: 070 ; THE START OF THE DATA LINE.
2900: 080 ;
2910: 090 12S3 AD0512 LDA SELET+1 DISPLAY HIGH BYTE OF
2920: 0A0 1266 20A311 JSR VUBYTE CURRENTLY-SELECTED ADDRESS.
2930: 0B0 1269 AD0512 LDA SELECT DISPLAY LOW BYTE OF
2940: 0C0 126C 20A311 JSR VUBYTE CURRENTLY-SELECTED ADDRESS.
2950: 0D0 ;
2960: 0E0 1265 207F11 JSR TVSKIP SKIP ONE SPACE AFTER
2970: 0F0 ; ADDRESS FIELD.
3000: 100 ;
3010: 110 1272 209412 JSR GET.SL GET CURRENTLY-SELECTED
3020: 120 ; BYTE.
3030: 130 ;
3040: 140 1275 40 PHA SAVE IT.
3050: 150 ;
3060: 160 1276 20A311 JSR VUBYTE DISPLAY IT, IN HEX FORMAT,
3070: 170 ; IN FIELD 1.
3080: 180 ;
3090: 190 1278 207F11 JSR TVSKIP SKIP ONE SPACE AFTER FIELD
3100: 1A0 ; 1.
3110: 1B0 ;
3120: 1C0 127C 60 PLA RESTORE CURRENTLY-SELECTED
3130: 1D0 ; BYTE TO ACCUMULATOR.
3140: 1E0 ;
3150: 1F0 127D 207C11 JSR VUCHAR DISPLAY IT IN CHARACTER
3160: 200 ; FORMAT, IN FIELD 2.
3170: 210 ;
3180: 220 1280 207F11 JSR TVSKIP SKIP ONE SPACE AFTER FIELD 2.
3190: 230 ;
3200: 240 6502 REGISTER
3210: 250 ; IMAGES IN FIELDS 3-6:

230 BEYOND GAMES
3400 1283 A200    LDX $0
3410 ;
3420 1285 BD0112 VUREGS LDA REGS,X
3430 1288 20A311 JSR VUBYTE
3440 1288 207F11 JSR TVSKIP
3450 ;
3460 ;
3470 128E E8 INX
3480 128F E004 CPX $4
3490 1291 D0F2 BNE VUREGS
3500 ;
3510 1293 60 RTS
3520 ;
3530 ;
3540 ;
3550 ;
3560 ;
3570 ;
3580 ;
3590 ;
3600 ;
3610 ;
3620 ;
3630 ;
3640 ;
3650 ;
3660 ;
3670 ;
3680 ;
3690 ;
3700 ;
3710 ;
3720 ;
3730 ;
3740 ;
3750 1294 A502 GET.SL LDA GETPTR
3760 1295 48 PHA
3770 1297 A603 LDX GETPTR+1 (PRESERVING THE ZERO PAGE).
3780 ;
3790 1299 A0512 LDA SELECT
3800 129C 8502 STA GETPTR
3810 129E A0512 LDA SELECT+1
3820 12A1 8503 STA GETPTR+1
3830 ;
3840 12A3 A000 LDY $0
3850 12A5 B122 LDA (GETPTR),Y
3860 12A7 A8 TAY
3870 12A8 60 PLA
3880 12A9 8502 STA GETPTR
3890 12AB 8603 STX GETPTR+1
3900 12AD 98 TYA
3910 12AE 60 RTS
3920 ;
3930 ;
3940 ;
3950 ;
3960 ;
3970 ;
START WITH ACCUMULATOR IMAGE.
LOOK UP THE REGISTER IMAGE.
DISPLAY IT IN HEX FORMAT.
SKIP ONE SPACE AFTER HEX FIELD.
GET READY FOR NEXT REGISTER...
DONE FOUR REGISTERS YET?
IF NOT, DO NEXT ONE...
IF ALL REGISTERS DISPLAYED, RETURN.

GET SELECTED BYTE

GET BYTE POINTED TO BY THE SELECT POINTER
(PRESERVING THE ZERO PAGE).

RETURN TO CALLER.
3980
3990
4000
4010
4020
4030
4040
4050
4060
4070
4080
4090
4100
4110

; **************************************************
; DISPLAY ARROW LINE
; **************************************************
4120 12AF A202 LINE.3 LDX #2 LOAD X WITH STARTING COLUMN.
4130 12B1 A004 LDY #4 LOAD Y WITH ROW NUMBER.
4140 12B3 203C11 JSR TVTOXY SET TV.PTR TO BEGINNING
; OF ARROW LINE.
4150
4160
4170 12B5 AC0012 LDY FIELD LOOK UP CURRENT FIELD.
4180 12B9 38 SEC
4190 12BA C007 CPY #7
4200 12BC 5005 BCC FLD.OK
4210 12BE A000 LDY #0
4220 12C0 B00012 STY FIELD
4230 12C3 B9CD12 FLD.OK LDA FIELDS,Y LOOK UP COLUMN NUMBER FOR CURRENT FIELD.
4240
4250
4260 12C6 A0 TAY USE THAT COLUMN NUMBER AS AN INDEX INTO THE ROW.
4270
4280
4290 12C7 AD0710 LDA ARROW PLACE AN UP-ARROW IN
4300 12CA 9100 STA (TV.PTR),Y COLUMN OF THE ARROW LINE.
4310 12CC 60 RTS RETURN TO CALLER.
4320
4330
4340 12CD 03 FIELDS .BYTE 3,6,8 THIS DATA AREA SHOWS WHICH
4340 12CE 06
4350 12CF 08
4350 12D0 0B .BYTE $0B,$0E COLUMN SHOULD GET AN UP-
4350 12D1 0E
4360 12D2 11 .BYTE $11,$14 ARROW TO INDICATE ANY ONE
4360 12D3 14 OF FIELDS 0-6. CHANGING
4370
4380
4390
4400
4410
4420
4430

232 BEYOND GAMES
Appendix C3:

Visible Monitor (Update Subroutine)
APPENDIX C3: ASSEMBLER LISTING OF
THEVISIBLE MONITOR

UPDATE SUBROUTINE

SEE CHAPTER 6 OF BEYOND GAMES: SYSTEMS
SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

BY KEN SKIER

EQUATES

PARAMS = $1000 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.
580 ;
590 ;
600 0020= ; SPACE = $20
610 ;
620 007F= ; RUBOUT = $7F
630 ;
640 000D= ; CR = $0D ASCII FOR CARRIAGE RETURN.
650 ;
660 ;
670 ;
680 ;
690 ;
700 ;
710 ;
720 ;
730 ;
740 ;
750 ;
760 ;
770 ;
780 ;
790 ;
800 ;
810 ;
820 ;
830 ;
840 ;
850 ;
860 ;
870 1100= ; TVSUBS = $1100
880 1100= ; CLR.TV = TVSUBS CLR.TV CLEARS THE SCREEN.
890 ;
8A0 ;
8B0 ;
8C0 ;
8D0 ;
8E0 1200= ; VMSUBS = $1200 STARTING PAGE OF VISIBLE
8F0 ; MONITOR CODE.
900 ;
910 ;
920 ;
930 ;
940 1294= ; GET.SL = VMSUBS+$54
950 ; GET.SL GETS THE CURRENTLY-SELECTED BYTE.
960 ;
970 ;
980 ;
990 ;
9A0 ;
9B0 ;
9C0 ;
9D0 ;
9E0 ;
9F0 ;
A00 ;
A10 ;
A20 ;
A30 ;
A40 ;
A50 ;
A60 ;
A70 ;
A80 ;
A90 ;
AA0 ;
AB0 ;
AC0 ;
AD0 ;
AE0 ;
AF0 ;
B00 ;
B10 ;
B20 ;
B30 ;
B40 ;
B50 ;
B60 ;
B70 ;
B80 ;
B90 ;
BA0 ;
BB0 ;
BC0 ;
BD0 ;
BE0 ;
BF0 ;
C00 ;
C10 ;
C20 ;
C30 ;
C40 ;
C50 ;
C60 ;
C70 ;
C80 ;
C90 ;
CA0 ;
CB0 ;
CC0 ;
CD0 ;
CE0 ;
CF0 ;
D00 ;
D10 ;
D20 ;
D30 ;
D40 ;
D50 ;
D60 ;
D70 ;
D80 ;
D90 ;
DA0 ;
DB0 ;
DC0 ;
DD0 ;
DE0 ;
DF0 ;
E00 ;
E10 ;
E20 ;
E30 ;
E40 ;
E50 ;
E60 ;
E70 ;
E80 ;
E90 ;
EA0 ;
EB0 ;
EC0 ;
ED0 ;
EE0 ;
EF0 ;
F00 ;
F10 ;
F20 ;
F30 ;
F40 ;
F50 ;
F60 ;
F70 ;
F80 ;
F90 ;
FA0 ;
FB0 ;
FC0 ;
FD0 ;
FE0 ;
FF0 ;
236 BEYOND GAMES
1160  
1170  
1180  
1190  
1200 1200  * = VMSUBS
1210  
1220  
1230  
1240  
1250 1200 00  FIELD .BYTE 0  NUMBER OF CURRENT FIELD.
1260  
1270  
1280 1201 00  REG.A .BYTE 0  IMAGE OF ACCUMULATOR.
1290  
1300 1202 00  REG.X .BYTE 0  IMAGE OF X-REGISTER.
1310  
1320 1203 00  REG.Y .BYTE 0  IMAGE OF Y-REGISTER.
1330  
1340 1204 00  REG.P .BYTE 0  IMAGE OF PROCESSOR STATUS
1350  
1360  
1370 1201=  REGS = REG.A
1380  
1390 1205 0000  SELECT .WORD 0  POINTER TO CURRENTLY-
1400  
1410  
1420  
1430  
1440  
1450  
1460  
1470  
1480  
1490  
1500  
1510  
1520  
1530  
1540  
1550  
1560  
1570  
1580 12E0  * = VMSUBS+$E0
1590  
1600  
1610 12E0 6C0810 GETKEY JMP (ROMKEY)  JSR GETKEY CALLS YOUR
1620  
1630  
1640  
1650  
1660  
1670  
1680  
1690  
1700  
1710  
1720  
1730  
237
MONITOR-UPDATE

GET A CHARACTER FROM THE KEYBOARD.
IS IT THE 'Y' KEY?
IF NOT, PERFORM NEXT TEST.

IF SO, SELECT NEXT FIELD.
IF ARROW WAS UNDER RIGHT-MOST FIELD, PLACE IT UNDER LEFT-MOST FIELD.
THEN RETURN TO CALLER.

IS IT THE '<' KEY?
IF NOT, PERFORM NEXT TEST.

IF SO, SELECT PREVIOUS FIELD: THE FIELD TO THE LEFT OF THE CURRENT FIELD.
THEN RETURN

IS IT THE SPACE BAR?
IF NOT, PERFORM NEXT TEST.

IF SO, STEP FORWARD THROUGH MEMORY BY INCREMENTING THE POINTER THAT SELECTS THE ADDRESS TO BE DISPLAYED.
THEN RETURN TO CALLER.

IS IT THE CARRIAGE RETURN?
IF NOT, PERFORM NEXT TEST.

IF SO, STEP BACKWARD THROUGH MEMORY BY DECREMENTING THE POINTER THAT SELECTS THE ADDRESS TO BE DISPLAYED.
THEN RETURN.

IS ARROW UNDER CHARACTER FIELD (FIELD 2)?
BNE IF.GO IF NOT, PERFORM NEXT TEST.

IF SO, STORE THE

LDA TU.PTR CHARACTER IN THE CURRENTLY-

SELECTED ADDRESS. (PRESEVING THE ZERO PAGE.)

LDA SELECT

STA TU.PTR

LDA SELECT+1

STA TU.PTR+1

TYA

LDY #0

STA (TU.PTR),Y

STA TU.PTR+1

PLA

STA TU.PTR

THEN RETURN.

RTS

IS IT 'G' FOR GO?

BNE IF.HEX IF NOT, PERFORM NEXT TEST.

LDY REG.Y IF SO, LOAD REGISTERS

LDB REG.X FROM REGISTER IMAGES...

LDA REG.P

PHA

LDA REG.A

PLP

JSR CALLIT AND CALL SELECTED ADDRESS.

THEN WHEN THE SUBROUTINE RETURNS.

STA REG.A SAVE REGISTER VALUES IN

LDA REG.P REGISTER IMAGES.

STA REG.P THEN RETURN TO CALLER.

RTS

JSR CALLIT CALLS THE

CURRENTLY-SELECTED ADDRESS,

INDIRECTLY.

JSR BINARY IS IT ASCII CHAR FOR 0-9 OR

A-F? IF SO, CONVERT TO BINARY.

IF KEYBOARD CHAR WAS N

0-9 OR A-F, PERFORM NEXT

TEST.

PULL KEYBOARD CHARACTER

FROM STACK, WHILE SAVING

BINARY EQUIVALENT IN A AND Y.

IS ARROW UNDER ADDRESS

FIELD (FIELD 0)?

EMI IF.CLR
2500 :  
2510 137D A203 ADRFLD LDX $3 
2520 137F 18 ADLOOP CLC 
2530 1380 00512 ASL SELECT 
2540 1383 2E012 ROL SELECT+1 
2550 1386 CA DEX 
2560 1387 10F6 EPL ADLOOP 
2570 1389 90 TYA 
2580 138A 00512 ORA SELECT 
2590 138B 2D0512 STA SELECT 
25A0 1380 60 RTS 
25B0 ;  
25C0 13E1 E081 NOTADR CPX #1 
25D0 13E3 D018 BNE REGFLD 
25E0 ;  
25F0 13F5 2SOF ROL.SL AND #$0F 
2600 13F7 48 PHA 
2610 13F9 2DS412 JSR GET.SL 
2620 13FA 00 A SL A 
2630 13FC 0A A SL A 
2640 13FD 0A A SL A 
2650 13FF 23F0 AND #$F0 
2660 13H1 6DAC13 STA TEMP 
2670 13H4 68 PLA 
2680 13H5 0DAC13 ORA TEMP 
2690 13H8 282D13 JSR PUT.SL 
26A0 13HA 60 RTS 
26B0 ;  
26C0 13AC 00 TEMP .BYTE 0 
26D0 ;  
26E0 ;  
26F0 13AD CA REGFLD DEX 
2700 13AE CA DEX 
2710 13AF CA DEX 
2720 13B0 A603 LDY #3 
2730 ;  
2740 13E2 18 RGLOOP CLC 
2750 13E3 1E0112 ASL REGS,X 
2760 13E5 88 DEY 
2770 13E7 10F9 BPL RGLOOP 
2780 13EB 1D0112 ORA REGS,X 
2790 13EC 9D0112 STA REGS,X 
27A0 13E8 60 RTS 
27B0 ;  
27C0 ;  
27D0 13C0 68 IF.CLR PLA 
27E0 13C1 C37F CMP #RUBOUT 
27F0 3410 ;  
2800 3420 ;  
2810 3430 ;  
2820 3448 ;  
2830 ;  
2840 3460 15C3 D004 BNE NOTCLR 
2850 ;  
2860 240 BEYOND GAMES 

SINCE ARROW IS UNDER ADDRESS 
FIELD, ROLL HEX DIGIT INTO 
ADDRESS FIELD BY ROLLING IT 
IT INTO THE POINTER THAT 
SELECTS THE DISPLAYED 
ADDRESS. 

THEN RETURN. 

IS ARROW UNDER FIELD 1? 
IF NOT, IT MUST BE UNDER 
A REGISTER FIELD. 

ROLL 4 LSB IN A INTO 
CURRENTLY-SELECTED BYTE. 
GET THE CURRENTLY-SELECTED 
BYTE AND SHIFT LEFT 4 TIMES... 

PUT IT IN CURRENTLY-SELECTED 
ADDRESS AND RETURN. 

THE ARROW MUST BE UNDER A 
REGISTER IMAGE: FIELD 3. 
4, 5, OR 6. 

ROLL HEX DIGIT INTO 
APPROPRIATE REGISTER IMAGE. 

RESTORE KEYBOARD CHARACTER. 
IS IT RUBOUT? (IF YOUR 
SYSTEM DOESN'T HAVE A 
RUBOUT KEY, SUBSTITUTE THE 
CODE FOR THE KEY YOU'LL USE 
TO CLEAR THE SCREEN.) 

IF IT ISN'T THE 'CLEAR 
SCREEN' KEY, PERFORM NEXT 

240 BEYOND GAMES
3480          ;
3480          ;
3500  13CE 200011     JSR CLR.TV   IF IT IS, THEN CLEAR THE
3510  13CB 60     RTS       SCREEN AND RETURN.
3520          ;
3530          ;
3540  13C9 C551   NOTCLR CMP #0'Q  IS IT 'Q' FOR QUIT?
3550  13CB D004   BNE OTHER   IF NOT, PERFORM NEXT TEST.
3560          ;
3570          ;
3580          ;
3590          ;
3600          ;
3610  13CD 68     PLA       POP UPDATE'S RETURN ADDRESS.
3620  13CE 68     PLA   
3630          ;
3640  13CF 28     PLP   RESTORE INITIAL 6502 FLAGS.
3650          ;
3660          ;
3670  13DB 60     RTS   VISION'S RETURN ADDRESS IS
3680          ;   NOW ON THE STACK.
3690          ;
3700          ;
3710          ;
3720          ;
3730          ;
3740  13D1 201010  OTHER JSR DUMMY   REPLACE THIS CALL TO
3750          ;   DUMMY WITH A CALL TO ANY
3760          ;   SUBROUTINE THAT EXTENDS
3770          ;   FUNCTIONALITY OF THE
3780          ;   VISIBLE MONITOR.
3790  13D4 60     RTS   THEN RETURN.
3800          ;
3810          ;
3820          ;
3830          ;
3840          ;
3850          ;
3860          ;
3870          ;
3880          ;
3890          ;
3900          ;
3910          ;
3920          ;
3930          ;
3940          ;
3950          ;
3960          ;
3970          ;
3980          ;
3990          ;
4000          ;
4010          ;
4020          ;
4030          ;
4040          ;
4050          ;

*************** ASCII TO BINARY ***************

*************** ASCII TO BINARY ***************

4060          ;

IF ACCUMULATOR HOLDS ASCII
0-9 OR A-F, THIS ROUTINE
RETURNS BINARY EQUIVALENT--
OTHERWISE, IT RETURNS $FF.
4060 13D5 3B   BINARY SEC
4070 13D6 E330  SBC #$30
4080 13D8 900F  BCC BAD
4090 13DA C90A  CMP #$0A
4100 13DB 900E  BCC GOOD
4110 13DE E907  SBC #$7
4120 13E0 C910  CMP #$10
4130 13E2 B005  BCS BAD
4140 13E4 3B    SEC
4150 13E5 C90A  CMP #$0A
4160 13E7 B003  BCS GOOD
4170 13E9 A9FF  BAD LDA #$FF
4180 13EB 68    RTS
4190    ;
4200 13EC A200  GOOD LDX 40
4210 13EE 68    RTS
Appendix C4:
Print Utilities
APPENDIX C4: ASSEMBLER LISTING OF PRINT UTILITIES

SEE CHAPTER 7 OF BEYOND GAMES: SYSTEMS SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

CONSTANTS

CR = $0D  CARRIAGE RETURN.

ETX = $FF  THIS CHARACTER MUST TERMINATE ANY MESSAGE STRING.

LF = $0A  LINE FEED.

OFF = 0

ON = $FF

EXTERNAL ADDRESSES
530
560
610
620
630
640
650
660
670
680
690
700
710
720
730
740
750
760
770
780
790
800
810
820
830
840
850
860
870
880
890
900
910
920
930
940
950
960
970
980
990
1000
1010
1020
1030
1040
1050
1060
1070
1080
1090
1100
1110
1120
1130
1140
1150
1160

PARAMS = $1000 ADDRESS OF SYSTEM DATA BLOCK.

ROMRT = PARAMS+$0C
POINT TO ROM ROUTINE THAT
SENDS CHAR TO SERIAL OUTPUT.

ROMTUT = PARAMS+$0A
POINT TO ROM ROUTINE THAT
PRINTS A CHAR TO THE SCREEN.

USROUT = PARAMS+$0E
POINT TO USER-WRITTEN
CHARACTER OUTPUT ROUTINE.

TVSUBS = $1100
ASCII = TVSUBS+$06

UMPAGE = $1200 VISIBLE MONITOR STARTING
PAGE

SELECT = UMPAGE+5
GET.SL = UMPAGE+$84
INC.SL = UMPAGE+$10D

***********************************************************************

VARIABLES

***********************************************************************
1170 ;
1180 1400  ;  \* = 1400
1190  ;
1200 1400 00  PRINTR  .BYTE  OFF  // PRINTER OUTPUT FLAG.
1210  ;
1220 1401  FF  TUT  .BYTE  ON  // TUT OUTPUT FLAG.
1230  ;
1240  ;
1250 1402  00  USER  .BYTE  OFF  // OUTPUT FLAG FOR USER-PROVIDED OUTPUT SUBROUTINE.
1260  ;
1270  ;
1280 1403  00  CHAR  .BYTE  0  // CHARACTER MOST RECENTLY PRINTED BY PR.CHR.
1290  ;
1300  ;  CHAR=00 MEANS PR.CHR HAS NEVER PRINTED A CHARACTER.
1310  ;
1320  ;
1330  ;
1340 1404  00  REPEAT  .BYTE  0  // THIS BYTE IS USED AS A COUNTER BY SPACES, CHARS, AND CR.LFS.
1350  ;
1360  ;
1370  ;
1380  ;
1390 1405  00  TENP.X  .BYTE  0  // DATA CELL: USED BY PR.MSG.
1400  ;
1410  ;
1420 1406  0000  RETURN  .WORD  0  // THIS POINTER IS USED BY PUSHSL AND POP.SL.
1430  ;
1440  ;
1450  ;
1460  ;
1470  ;
1480  ;
1490  ;
1500  ;
1510  ;  ******************************************************
1520  ;
1530  ;
1540  ;  ******************************************************
1550  ;
1560  ;
1570  ;
1580  ;
1590  ;
1600  ;
1610  ;
1620  ;
1630 1408  ASFF  TUT.ON  LDA  #ON  // SELECT SCREEN FOR OUTPUT
1640 140A  8D0114  STA  TUT  // BY SETTING ITS DEVICE FLAG.
1650 140D  60  RTS
1660  ;
1670  ;
1680  ;
1690  ;
1700  ;
1710  ;
1720 140E  AS00  TUT.OFF  LDA  #OFF  // DE-SELECT SCREEN FOR
1730 1410  8D0114  STA  TUT  // OUTPUT BY CLEARING ITS
1740 1413  60  RTS  // DEVICE FLAG.

247
1750 ;
1760 ;
1770 ;
1780 ;
1790 ;
1800 1414 ASFF PR.ON LDA #ON SELECT PRINTER FOR OUTPUT
1810 1416 8D0014 STA PRINTR BY SETTING ITS DEVICE FLAG.
1820 1419 60 RTS
1830 ;
1840 ;
1850 ;
1860 ;
1870 ;
1880 141A A900 PR.OFF LDA #OFF DE-SELECT PRINTER FOR OUTPUT
1890 141C 8D0014 STA PRINTR BY CLEARING ITS DEVICE FLAG.
1900 141F 60 RTS
1910 ;
1920 ;
1930 ;
1940 ;
1950 ;
1960 1420 ASFF USR.ON LDA #ON SELECT USER-WRITTEN
1970 1422 8D0214 STA USER SUBROUTINE BY SETTING
1980 1425 60 RTS USER'S DEVICE FLAG.
1990 ;
2000 ;
2010 ;
2020 ;
2030 ;
2040 1426 A900 USR.OFF LDA #OFF DE-SELECT USER-WRITTEN
2050 1428 8D0214 STA USER OUTPUT SUBROUTINE BY
2060 142B 60 RTS CLEARING ITS DEVICE FLAG.
2070 ;
2080 ;
2090 ;
2100 ;
2110 ;
2120 142C 200814 ALL.ON JSR TUT.ON SELECT ALL OUTPUT DEVICES
2130 142F 201414 JSR PR.ON BY SELECTING EACH OUTPUT
2140 1432 202014 JSR USR.ON DEVICE INDIVIDUALLY.
2150 1435 60 RTS
2160 ;
2170 ;
2180 ;
2190 ;
2200 ;
2210 1435 205E14 ALLOFF JSR TUTOFF DE-SELECT ALL OUTPUT DEVICES
2220 1439 201A14 JSR PR.OFF BY DE-SELECTING EACH ONE
2230 143C 202B14 JSR USR.OFF INDIVIDUALLY.
2240 143F 60 RTS
2250 ;
2260 ;
2270 ;
2280 ;
2290 ;
2300 ;
2310 ;
2320 ;
A GENERAL CHARACTER PRINT ROUTINE

PRINT CHARACTER IN ACCUMULATOR
ON ALL CURRENTLY-SELECTED OUTPUT DEVICES.

2500 1440 CS00 FR.CHX CMP #0 TEST CHARACTER.
2520 1442 F024 BEQ EXIT IF IT'S A NULL, RETURN
2530 ; WITHOUT PRINTING IT.
2540 1444 BD0314 STA CHAR SAVE CHARACTER.
2550 ;
2560 1447 AD0114 LDA TVT IS SCREEN SELECTED?
2570 144A F006 BEQ IF.FR IF NOT, TEST NEXT DEVICE.
2580 ;
2590 144C AD0314 LDA CHAR IF SO, SEND CHARACTER
2600 144F 206914 JSR SEND.1 INDIRECTLY TO SYSTEM'S
2610 ; TVT OUTPUT ROUTINE.
2620 ;
2630 ;
2640 1452 AD0014 IF.FR LDA PRINTR IS PRINTER SELECTED?
2650 1455 F006 BEQ IF.USR IF NOT, TEST NEXT DEVICE.
2660 ;
2670 1457 AD0314 LDA CHAR IF SO, SEND CHARACTER
2680 145A 206C14 JSR SEND.2 INDIRECTLY TO SYSTEM'S
2690 ; PRINTER DRIVER.
2700 ;
2710 ;
2720 145D AD0214 IF.USR LDA USER IS USER-WRITTEN OUTPUT
2730 ; SUBROUTINE SELECTED?
2740 1460 F006 BEQ EXIT IF NOT, RETURN.
2750 ;
2760 1462 AD0314 LDA CHAR IF SO, SEND CHARACTER
2770 1465 206F14 JSR SEND.3 INDIRECTLY TO USER-WRITTEN
2780 ; SUBROUTINE.
2790 ;
2800 1468 60 EXIT RTS RETURN TO CALLER.
2810 ;
2820 ;
2830 ;
2840 ;
2850 ;
2860 ;
2870 ;
2880 1469 6C0A10 SEND.1 JMP (RONTVT)
2890 ;
2900 146C 6C0C10 SEND.2 JMP (ROMPRT)
2910  ;
2920 146F 6C0E10  SEND.3 JMP (USROUT)
2930  ;
2940  ;
2950  ;
2960  ;
2970  ;
2980  ;
2990  ;
3000  ;
3010  ;
3020  ;
3030  ;
3040  ;
3050  ;
3060  ;
3070  ;
3080  ;
3090  ;
3100  ;
3110  ;
3120  ;
3130 1472 A90D  CR.LF  LDA #$CR  SEND A CARRIAGE RETURN
3140 1474 204014  JSR PR.CHAR
3150 1477 A90A  LDA #$LF  AND A LINE-FEED TO ALL
3160 1479 204014  JSR PR.CHAR  CURRENTLY-SELECTED DEVICES.
3170 147C 60  RTS  THEN RETURN.
3180  ;
3190  ;
3200  ;
3210  ;
3220  ;
3230  ;
3240  ;
3250  ;
3260  ;
3270 147D A920  SPACE  LDA #$20  LOAD ACCUMULATOR WITH AN
3280 147F 204014  JSR PR.CHAR  ASCII SPACE AND PRINT IT.
3290 1482 60  RTS  THEN RETURN.
3300  ;
3310  ;
3320  ;
3330  ;
3340  ;
3350  ;
3360  ;
3370  ;
3380  ;
3390  ;
3400  ;
3410  ;
3420  ;
3430  ;
3440  ;
3450  ;
3460  ;
3470  ;
3480  ;

250 BEYOND GAMES
3490 ;
3500 ;
3510 ; PR.BYT OUTPUTS THE ACCUMULATOR, IN HEX,
3520 ; TO ALL CURRENTLY-SELECTED DEVICES.
3530 ;
3540 ;
3550 ;
3560 14B3 48 PR.BYT PHA ; SAVE BYTE.
3570 14B4 4A LSR A ; DETERMINE ASCII FOR 4 MSB...
3580 14B5 4A LSR A
3590 14B6 4A LSR A
3600 14B7 4A LSR A
3610 1488 20B611 JSR ASCII ; ...IN THE BYTE.
3620 148B 204014 JSR PR.CHAR ; PRINT THAT ASCII CHAR TO CURRENT DEVICE(S).
3630 ;
3640 14B1 60 PLA ; DETERMINE ASCII FOR 4 LSB
3650 14BF 20B611 JSR ASCII ; IN THE ORIGINAL BYTE.
3660 1492 204014 JSR PR.CHAR ; PRINT THAT CHARACTER.
3670 1495 60 RTS ; RETURN TO CALLER.
3680 ;
3690 ;
3700 ;
3710 ;
3720 ;
3730 ;
3740 ;
3750 ;*****************************************************************************
3760 ; REPETITIVE CHARACTER OUTPUT
3770 ;
3780 ;*****************************************************************************
3790 ; PRINT X SPACES:
3800 ;
3810 ;
3820 ;
3830 ;
3840 ;
3850 ;
3860 14B6 A920 SPACES LDA #$20 ; LOAD A WITH ASCII SPACE.
3870 ;
3880 ;
3890 ;
3900 ;
3910 ;
3920 ;
3930 ;
3940 ;
3950 ;
3960 ;
3970 149B BE0414 CHARS STX REPEAT ; PRINT CHAR IN A X TIMES.
3980 14B9 48 RPLOOP PHA ; SAVE CHAR TO BE REPEATED.
3990 149C AE0414 LDX REPEAT ; REPEAT COUNTER TIMED OUT?
4000 149F F00A BEQ RPTEND ; IF SO, EXIT. IF NOT,
4010 14A1 CE0414 DEC REPEAT ; DECREMENT REPEAT COUNTER.
4020 14A4 204014 JSR PR.CHAR ; PRINT CHARACTER.
4030 ;
4040 14A7 68 PLA ; RESTORE CHARACTER TO A.
4050 14A6 18 CLC ; LOOP BACK TO PRINT IT
4060 14A9 90F0 BCC RPLOOP ; AGAIN IF NECESSARY.
4070 ;
4080 14AB 68 RPTEND PLA CLEAN UP STACK AND
4090 14AC 60 RTS RETURN TO CALLER.
4100 ;
4110 ;
4120 ;
4130 ; PRINT X NEWLINES
4140 ;
4150 ;
4160 14AB 8E0414 CR.LFS STX REPEAT INITIALIZE REPEAT COUNTER.
4170 14B0 AE0414 CRLOOP LDX REPEAT EXIT IF REPEAT COUNTER
4180 14B3 F009 BEQ END.CR HAS TIMED OUT.
4190 14B5 CE0414 DEC REPEAT DECREMENT REPEAT COUNTER.
4200 14B8 207214 JSR CR.LF PRINT A CARRIAGE RETURN
4210 ; AND A LINE FEED.
4220 14BB 18 CLC LOOP BACK TO SEE IF DONE
4230 14BC 90F2 BCC CRLOOP YET.
4240 ;
4250 14BE 60 END.CR RTS RETURN TO CALLER.
4260 ;
4270 ;
4280 ;
4290 ;
4300 ;
4310 ;
4320 ;
4330 ;
4340 ;
4350 ; ;***************************************************************************
4360 ;
4370 ; PRINT A MESSAGE
4380 ;
4390 ; ;***************************************************************************
4400 ;
4410 ;
4420 ;
4430 ;
4440 ; Xth POINTER IN ZERO PAGE
4450 ; POINTS TO THE MESSAGE.
4460 ;
4470 ;
4480 14BF 8E0514 PR.MSG STX TEMP.X SAVE X REGISTER, WHICH
4490 ; SPECIFIES MESSAGE POINTER.
4500 ;
4510 14C2 B501 LDA 1,X SAVE MESSAGE POINTER.
4520 14C4 48 PHA
4530 14C5 B500 LDA 0,X
4540 14C7 48 PHA
4550 ;
4560 14CA AE0514 LOOP LDXX TEMP.X RESTORE ORIGINAL X, SO IT
4570 ; SPECIFIES MESSAGE POINTER.
4580 14CB A100 LDA (0,X) GET NEXT CHARACTER FROM
4590 14CD C9FF CMP #ETX MESSAGE. IS MESSAGE OVER?
4600 14CF F00C BEQ MSGEND IF SO, HANDLE END OF MESSAGE.
4610 ;
4620 14D1 F600 INC 0,X IF NOT, INCREMENT POINTER.
4630 14D3 D002 BNE NEXT SO IT POINTS TO NEXT
4640 14D5 F601 INC 1,X CHARACTER IN MESSAGE.
PRINT THE CHARACTER.
LOOP BACK FOR NEXT CHARACTER...
RESTORE ORIGINAL MESSAGE POINTER.
RETURN TO CALLER, WITH MESSAGE POINTER PRESERVED.

PRINT THE FOLLOWING TEXT

PULL RETURN ADDRESS FROM STACK AND SAVE IT IN X AND Y REGISTERS.
SAVE THE SELECT POINTER.
SET SELECT=RETURN ADDRESS.

ADVANCE SELECT TO STX.
SELECT NEXT CHARACTER.
GET IT.
IS IT END OF MESSAGE?
IF SO, RETURN.
IF NOT, PRINT CHARACTER.
LOOP BACK FOR NEXT CHARACTER...

RESTORE SELECT POINTER.
PUSH ADDRESS OF ETX ONTO THE STACK.
5230 1511 60       RTS       RETURN (TO BYTE IMMEDIATELY
5240               FOLLOWING THE ETX.)
5250               ;
5260               ;
5270               ;
5280               ;
5290               ;
5300               ;
5310               ;
5320               ;
5330               ;
5340               ;
5350               ;
5360               ;
5370               ;
5380               ;
5390               ;
5400               ;
5410               ;
5420               ;
5430               ;
5440               ;
5450 1512 68       PUSHSL PLA      PULL RETURN ADDRESS FROM
5460 1513 800614    STA RETURN     STACK AND SAVE IT IN RETURN.
5470 1516 68        PLA          PULL SELECT+1
5480 1517 800714    STA RETURN+1   ON THE STACK.
5490               ;
5500               ;
5510 151A AD0612    LDA SELECT+1  PUSH RETURN ADDRESS BACK
5520 151D 48        PHA          ON THE STACK.
5530 151E AD0512    LDA SELECT    RETURN TO CALLER. CALLER
5540 1521 48        PHA          WILL FIND SELECT ON STACK.
5550               ;
5560               ;
5570 1522 AD0714    LDA RETURN+1  SAVE RETURN ADDRESS.
5580 1525 48        PHA          PLA
5590 1526 AD0614    LDA RETURN    STA RETURN
5600 1529 48        PHA          PLA
5610               ;
5620               ;
5630 152A 60        RTS          STA RETURN+1
5640               ;
5650               ;
5660               ;
5670               ;
5680               ;
5690               ;
5700               ;
5710               ;
5720               ;
5730 152B 69        POP.SL PLA     LOAD SELECT FROM STACK
5740 152C 800614    STA RETURN    STA SELECT
5750 152F 68        PLA          PLA
5760 1530 800714    STA RETURN+1
5770               ;
5780               ;
5790 1533 68        PLA          STA SELECT
5800 1534 800512    STA SELECT
PLA
STA SELECT+1

LDA RETURN+1
PHA
PLACE RETURN ADDRESS BACK ON STACK.

LDA RETURN
PHA

RTS
RETURN TO CALLER.
Appendix C5:
Two Hexdump Tools
APPENDIX C5: ASSEMBLER LISTING OF
TWO HEXDUMP TOOLS

SEE CHAPTER B OF BEYOND GAMES: SYSTEMS
SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

BY KEN SKIER

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

TUVSUBS=$1100  STARTING PAGE OF DISPLAY CODE.
CLR.TV=TUVSUBS
ASCII =TUVSUBS+$B6

VMPAGE=$1200  STARTING PAGE OF VISIBLE MONITOR CODE.
SELECT=VMPAGE+5
VISMON=VMPAGE+7
GET.SL=VMPAGE+$94
INC.SL=VMPAGE+$10D

PFRPAGE=$1400  STARTING PAGE OF PRINT UTILITIES.
TUT.ON=PFRPAGE+8
TUTOFF=PFRPAGE+$0E
PR.ON =PFRPAGE+$14
PR.OFF=PFRPAGE+$1A
PR.CHAR=PFRPAGE+$40
CR.LF =PFRPAGE+$72
SPACE =PFRPAGE+$7D
SPACES=PFRPAGE+$96
PR.BYT=PFRPAGE+$B3
PRINT:=PFRPAGE+$E4
PUSHSL=PFRPAGE+$112
POP.SL=PFRPAGE+$12B

VARIABLES
1160 ;
1170 ;
1180 ;
1190 ;
1200 ;
1210 1550 =$1550
1220 ;
1230 ;
1240 ;
1250 ;
1260 ;
1270 1550 00 COUNTR .BYTE 0 THIS BYTE COUNTS THE LINES
1280 ;
1290 ;
1300 1551 04 NUMLNS .BYTE 4 DUMPED BY TVDUMP.
1310 ;
1320 ;
1330 ;
1340 1552 0000 SA .WORD 0 NUMBER OF LINES TO BE
1350 ;
1360 1554 FFFF EA .WORD $FFFF DUMPED BY TVDUMP.
1370 ;
1380 ;
1390 ;
1400 1556 00 COLUMN .BYTE 0 DATA CELL: USED BY PRLINE
1410 ;
1420 ;
1430 ;
1440 ;
1450 ;
1460 ;
1470 ;
1480 ;
1490 ;
1500 ;
1510 ;
1520 ;
1530 ;
1540 ;
1550 ;
1560 ;
1570 ;
1580 ;
1590 1557 200814 TVDUMP JSR TVT.ON SELECT TUT AS OUTPUT DEVICE.
1600 155A A5115 LDA NUMLNS SET COUNTR TO NUMBER OF
1610 155D 0D5015 STA COUNTR LINES TO BE Dumped.
1620 ;
1630 1550 AD0512 LDA SELECT SET SELECT TO BEGINNING OF
1640 1563 29FB AND #$FB A SCREEN LINE, BY ZEROING
1650 1565 800512 STA SELECT 3 LSB IN SELECT.
1660 ;
1670 1558 207214 JSR CR.LF SKIP TWO LINES ON THE
1680 1558 207214 JSR CR.LF SCREEN.
1690 ;
1700 155E 20A115 DUMPLN JSR FR.ADR PRINT THE SELECTED ADDRESS.
1710 ;
1720 1571 207214 JSR CR.LF ADVANCE TO A NEW LINE ON
1730 ;
SCREEN. (NOT NEEDED ON
1740 1574 207D14 DMPBYT JSR SPACE
1790  ; PRINT A SPACE TO THE SCREEN.
1800 1577 209A15 JSR DUMP5L
1810  ; DUMP SELECTED BYTE.
1820 157A 200013 JSR INC.SL
1830  ; SELECT NEXT BYTE.
1840 157D AD0512 LDA SELECT
1850 1560 2907 AND #07
1860 1582 D0F0 BNE DMPBYT
1870  ; IS IT THE BEGINNING OF A NEW SCREEN LINE (3 LSB=0?)
1880  ; IF NOT, DUMP NEXT BYTE...
1890 1584 207214 JSR CR.LF
1900  ; IF SO, ADVANCE TO A NEW LINE ON THE SCREEN.
1910  ; DOES THIS ADDRESS MARK THE BEGINNING OF A NEW HEX LINE?
1920 1587 AD0512 LDA SELECT
1930 158A 290F AND #0F
1940  ; (4 LSB = 0?)
1950  ; IF SO, ADVANCE TO A NEW LINE ON SCREEN.
1960 159C D003 BNE IFDONE
1970 159E 207214 JSR CR.LF
1980  ; IF NOT, DUMP NEXT LINE.
1990  ; DUMPED LAST LINE YET?
2000 1591 CE5015 IFDONE DEC COUNTR
2010 1594 D008 BNE DUMPLN
2020  ; DE-SELECT T宇 AS OUTPUT DEVICE.
2030  ; RETURN TO CALLER.
2040 1596 200E14 JSR TUTOFF
2050  ;
2060  ;
2070 1559 60 RTS
2080  ;
2090  ;
2100  ;
2110  ;
2120  ;
2130  ;
2140  ;
2150  ;
2160  ;
2170  ;
2180  ;
2190  ;
2200  ;
2210  ;
2220  ;
2230  ;
2240  ;
2250  ;
2260  ;
2270 159A 209412 DUMP5L JSR GET.SL
2280 159D 208314 JSR PR.BYT
2290 15A0 60 RTS
2300  ; GET CURRENTLY-SELECTED BYTE
2310  ; AND PRINT IT IN HEX FORMAT.
2320  ; RETURN TO CALLER.
2320    ;
2330    ;
2340    ;
2350    ;
2360    ;
2370    ;
2380    ;
2390    ;
2400    ;
2410    ;
2420    ;
2430    ; PRINT SELECTED ADDRESS
2440    ;
2450    ;
2460    ;
2470    ;
2480    ;
2490    ;
2500    ;
2510    ;
2520 15A1 AD0612 PR.ADDR LDA SELECT+1 FIRST PRINT THE HIGH BYTE...
2530 15A4 200314 JSR PR.BYT
2540 15A7 AD0512 LDA SELECT ...THEN PRINT THE LOW BYTE.
2550 15AA 200314 JSR PR.BYT
2560 15AD 60 RTS
2570    ;
2580    ;
2590    ;
2600    ;
2610    ;
2620    ;
2630    ;
2640    ;
2650    ;
2660    ;
2670    ;
2680    ;
2690    ;
2700    ;
2710    ;
2720    ;
2730    ;
2740    ;
2750    ;
2760 15AE 20C915 PRDUMP JSR TITLE DISPLAY THE TITLE
2770 15B1 20E915 JSR SETADS LET USER SET START ADDRESS
2780    ; AND END ADDRESS OF MEMORY TO BE DUMPED.
2790    ; (SETADS RETURNS W/SELECT=EA.)
2800    ;
2810 15B4 20A017 JSR GOTOSA SET SELECT=SA.
2820 15B7 201414 JSR PR.OH SELECT PRINTER FOR OUTPUT.
2830    ;
2840 15BA 20EB16 JSR HEADER OUTPUT HEXDUMP HEADER.
2850    ;
2860    ;
2870 15BD 204217 HXLOOP JSR PRLINE DUMP ONE LINE.
2880 15C0 10FB BPL HXLOOP DUMP LAST LINE? IF NOT,
2890    ; DUMP NEXT LINE.

263
2800  ;
2810  15c2 207214  ; JSR CR.LF IF SO, GO TO A NEW LINE.
2820  ;
2830  15c5 201a14  ; JSR PR.OFF DE-SELECT PRINTER FOR OUTPUT.
2840  ;
2850  15c8 60  ; RTS RETURN TO CALLER.
2860  ;
2870  ;
2880  ;
2890  ;
2900  ;
2910  ;
2920  ;
2930  ;
2940  ;
2950  ;
2960  ;
2970  ;
2980  ;
2990  ;
3000  ;
3010  ;
3020  ;
3030  ;
3040  ;
3050  ;
3060  ;
3070  ;
3080  ;
3090  ;
3100  ;
3110  ;
3120  ;
3130  ;
3140  ;
3150  15c9 200011 TITLE JSR CLR.TV CLEAR THE SCREEN.
3160  15cc 200814 JSR TUT.ON SELECT SCREEN FOR OUTPUT.
3170  15cf 20e414 JSR PRINT: OUTPUT THE FOLLOWING TEXT:
3180  15d2 7f .BYTE TEX TEXT STRING MUST START
3190  ; WITH A START OF TEXT CHAR.
3200  15d3 0d .BYTE CR,'PRINTING HEXDUMP',CR,LF,LF
3200  15d4 50
3200  15d5 52
3200  15d6 49
3200  15d7 4e
3200  15d8 54
3200  15d9 49
3200  15da 4e
3200  15db 47
3200  15dc 20
3200  15dd 48
3200  15de 45
3200  15df 50
3200  15e0 44
3200  15e1 55
3200  15e2 4d
3200  15e3 50
3200  15e4 0d
3200  15e5 0a
3200  15e6 0a
3210  15e7 ff .BYTE ETX TEXT STRING MUST END WITH
3220  ; AN END OF TEXT CHARACTER.
3230  15e8 60  ; RTS RETURN TO CALLER.
3240  ;
3250  ;
3260  ;
3270  ;
3280  ;

264 BEYOND GAMES
3450 15E9 200814 SETADS JSR TUT.ON SELECT SCREEN FOR OUTPUT
3470 15EC 20E414 JSR PRINT: PUT PROMPT ON SCREEN:
3480 15EF 7F .BYTE TEX
3490 15F0 0D .BYTE CR,LF,'SET STARTING ADDRESS'
3490 15F1 6A
3490 15F2 53
3490 15F3 45
3490 15F4 54
3490 15F5 20
3490 15F6 53
3490 15F7 54
3490 15F8 41
3490 15F9 52
3490 15FA 54
3490 15FB 43
3490 15FC 4E
3490 15FD 47
3490 15FE 20
3490 15FF 41
3490 1600 44
3490 1601 44
3490 1602 52
3490 1603 45
3490 1604 53
3490 1605 53
3490 1606 20
3500 1607 41
3500 1608 4C
3500 1609 44
3500 160A 20
3500 160B 59
3500 160C 52
3500 160D 45
3500 160E 53
3500 160F 53
3500 1610 20
3500 1611 22
3500 1612 51
3500 1613 22
3500 1614 2E
3510 1615 FF .BYTE ETX

3290
3300
3310
3320
3330
3340
3350
3360
3370
3380
3390
3400
3410
3420
3430
3440
3450

LET USER SET STARTING ADDRESS AND END ADDRESS OF A BLOCK OF MEMORY:

**********
3520 1616 200712  JSR VISMON CALL VISIBLE MONITOR. SO
3530  ; USER CAN SELECT START ADDRESS
3540  ; OF THE BLOCK.
3550  JSR SHERE SET START ADDRESS (SA)=SELECT
3560  ; HAVING SET THE START ADDRESS,
3570  ; SA, LET'S SET THE END ADDRESS,
3580  ; EA.
3590  ;
3600  ;
3610  ;
3620  ;
3630  ;
3640  ;
3650  ;
3660  ;
3670  ;
3680  ;
3690  ;
3700  JSR TUT.ON SELECT SCREEN FOR OUTPUT.
3710 161C 200814 SET.EA  JSR PRINT: PUT PROMPT ON SCREEN:
3720 161F 28E414 .BYTE TEx  .BYTE CR,LF,'SET END ADDRESS'
3730 1622 7F
3740 1623 0D
3750 1624 0A
3760 1625 53
3770 1626 45
3780 1627 54
3790 1628 20
3800 1629 45
3810 162A 4E
3820 162B 44
3830 162C 20
3840 162D 41
3850 162E 44
3860 162F 44
3870 1630 52
3880 1631 45
3890 1632 53
3900 1633 53
3910 1634 20
3920 1635 41
3930 .BYTE 'AND PRESS "Q".',ETX
3940 1636 4E
3950 1637 44
3960 1638 20
3970 1639 50
3980 163A 52
3990 163B 45
3990 163C 53
3990 163D 53
3990 163E 20
3990 163F 22
3990 1640 51
3990 1641 22
3990 1642 2E
3990 1643 FF
3990
3990 JSR VISMON LET USER SELECT END ADDRESS.
3990  ;
3990

286 BEYOND GAMES
3790 1647 30  SEC IF USER TRIED TO SET AN
3800 1648 ADD012  LDA SELECT+1 ADDRESS LESS THAN THE
3810 164B CDS315  CMP SA+1 STARTING ADDRESS,
3820 164E 9024  BCC TOOLW MAKE USER DO IT OVER.
3830 1650 D009  BNE EAHERE IF SELECT>SA, SET EA=SELECT.
3840  ; THAT WILL MAKE EA>SA,
3850  9;
3860  9;
3870  9;
3880 1652 ADD012 LDA SELECT
3890 1655 CDS215  CMP SA
3900 1656 901A  BCC TOOLW
3910  9;
3920  9;
3930  9;
3940  9;
3950 1658 ADD012 EAHERE LDA SELECT+1 SET EA=SELECT.
3960 165D 80515  STA EA+1
3970 1660 ADD012 LDA SELECT
3980 1663 80515  STA EA
3990 1666 60  RTS RETURN WITH EA SET BY CALLER
4000  9; (JSR EAHERE); EA SET BY USER
4010  9; (JSR SET.EA); OR SA AND EA
4020  9; SET BY USER (JSR SETADS).
4030  9;
4040 1667 ADD012, EAHERE LDA SELECT+1 SET SA=SELECT.
4050 166A 80515  STA SA+1
4060 166D ADD012 LDA SELECT
4070 1670 80515  STA SA
4080 1673 60  RTS RETURN WITH SA=SELECT.
4090  9;
4100 1674 20E414 TOOLW JSR PRINT: SINCE USER SET ENDING
4110  9; ADDRESS TOO LOW, PUT A
4120  9; PROMPT ON THE SCREEN:
4130 1677 7F  .BYTE TEX
4140 1678 80  .BYTE CR,LF,LF,LF,' ERROR!!!'
4140 1679 8A  
4140 167A 8A  
4140 167B 8A  
4140 167C 20  
4140 167D 45  
4140 167E 52  
4140 167F 52  
4140 1680 4F  
4140 1681 52  
4140 1682 21  
4140 1683 21  
4140 1684 21  
4140 1685 20  
4150 1686 45  
4150 1687 4E  
4150 1688 44  
4150 1689 20  
4150 168A 41  
4150 168B 44  
4150 168C 44  
4150 168D 52  
4150 168E 45  

.BYTE 'END ADDRESS LESS THAN START ADDRESS,'
4150 168F 53
4150 1692 53
4150 1694 20
4150 1692 4C
4150 1693 4F
4150 1694 53
4150 1695 53
4150 1696 20
4150 1697 54
4150 1698 48
4150 1699 41
4150 169A 4E
4150 169B 20
4150 169C 53
4150 169D 54
4150 169E 41
4150 169F 52
4150 16A0 54
4150 16A1 20
4150 16A2 41
4150 16A3 44
4150 16A4 44
4150 16A5 52
4150 16A6 45
4150 16A7 53
4150 16A8 53
4150 16A9 2C
4160 16AA 20
4160 16AB 57
4160 16AC 48
4160 16AD 43
4160 16AE 43
4160 16AF 48
4160 16B0 20
4160 16B1 49
4160 16B2 53
4160 16B3 20
4160 16B4 FF
4170 16B5 205B16
4180
4190 1680 4C1C16
4200
4210
4220
4230
4240
4250
4260
4270
4280
4290
4300
4310
4320
4330
4340
4350
4360

.JSR PR.SA
PRINT START ADDRESS.

; JMP SET.CA
AND LET THE USER SET A
NEW END ADDRESS.

; ****************************************

; PRINT START ADDRESS

; ****************************************

268 BEYOND GAMES
4370 ;
4380 ;
4390 16BD A924 PR.SA LDA $8$ PRINT A DOLLAR SIGN, TO
4400 16DD 204014 JSR PR.CH R INDICATE HEXADECIMAL.
4410 ;
4420 16C0 AD5315 LDA SA+1 PRINT HIGH BYTE OF START
4430 16C3 208314 JSR PR.BY T ADDRESS.
4440 ;
4450 16C6 AD5215 LDA SA PRINT LOW BYTE OF START
4460 16C9 208314 JSR PR.BY T ADDRESS.
4470 16CC 60 RTS RETURN TO CALLER.
4480 ;
4490 ;
4500 ;
4510 ;
4520 ;
4530 ;
4540 ;
4550 ; ;*****************************************************************
4560 ;
4570 ; PRINT END ADDRESS
4580 ;
4590 ; ;*****************************************************************
4600 ;
4610 ;
4620 ;
4630 ;
4640 ;
4650 16CD A924 PR.EA LDA $8$ PRINT A DOLLAR SIGN, TO
4660 16CF 204014 JSR PR.CH R INDICATE HEXADECIMAL.
4670 16D2 AD5515 LDA EA+1 PRINT HIGH BYTE OF END
4680 16D5 208314 JSR PR.BY T ADDRESS.
4690 16D8 AD5415 LDA EA PRINT LOW BYTE OF END
4700 16DB 208314 JSR PR.BY T ADDRESS.
4710 16DE 60 RTS RETURN TO CALLER.
4720 ;
4730 ;
4740 ;
4750 ;
4760 ;
4770 ;
4780 ;
4790 ;
4800 ;
4810 ;
4820 ; ;*********************************************************************************
4830 ;
4840 ; PRINT RANGE OF ADDRESSES
4850 ; ;*********************************************************************************
4860 ;
4870 ;
4880 ;
4890 ;
4900 ;
4910 ;
4920 16DF 20BB16 RANGE JSR PR.SA PRINT STARTING ADDRESS.
4930 16E2 A92D LDA $8$ PRINT A HYPHEN.
4940 16E4 204014 JSR PR.CH R

269
43E0 16E7 20CD16 JSR PR. EA PRINT END ADDRESS.
43E0 16EA 60 RTS RETURN TO CALLER.

4970 ;
4980 ;
4990 ;
5000 ;
5010 ;
5020 ;
5030 ;
5040 ;
5050 ;
5060 ;
5070 ;
5080 ;
5090 ;
5100 ;
5110 ;
5120 ;
5130 ;
5140 ;
5150 ;
5160 ;
5170 16EE 23E414 HEADER JSR PRINT:
5180 16EE 7F .BYTE TEX
5190 16EF 00 .BYTE CR,LF,LF,'DUMPING '
51A0 16F0 0A
51B0 16F1 0A
51C0 16F2 44
51D0 16F3 55
51E0 16F4 4B
51F0 16F5 50
5200 16F6 49
5210 16F7 4E
5220 16F8 47
5230 16F9 20
5240 16FA FF .BYTE ETX
5250 16FB 20DF16 JSR RANGE
5260 16FE 207214 JSR CR,LF
5270 1701 20E414 JSR PRINT:
5280 1704 7F .BYTE TEX,LF,LF
5290 1705 0A .BYTE ' 0 1 2 3 4 5 6 7 '
52A0 1706 0A
52B0 1707 20
52C0 1708 20
52D0 1709 20
52E0 170A 20
52F0 170B 20
5300 170C 20
5310 170D 20
5320 170E 20
5330 170F 30
5340 1710 20
5350 1711 20
5360 1712 31
5370 1713 20
5380 1714 20
5390 1715 32
53A0 1716 20

270 BEYOND GAMES
5250 1717 20
5250 1718 33
5250 1719 20
5250 171A 20
5250 171B 34
5250 171C 20
5250 171D 20
5250 171E 35
5250 171F 20
5250 1720 20
5250 1721 36
5250 1722 20
5250 1723 20
5250 1724 37
5250 1725 20
5250 1726 20
5250 1727 38
5250 1728 20
5250 1729 20
5250 172A 39
5250 172B 20
5250 172C 20
5250 172D 41
5250 172E 20
5250 172F 20
5250 1730 42
5250 1731 20
5250 1732 20
5250 1733 43
5250 1734 20
5250 1735 20
5250 1736 44
5250 1737 20
5250 1738 20
5250 1739 45
5250 173A 20
5250 173B 20
5250 173C 46
5270 173D 00
5270 173E 0A
5270 173F 0A
5270 1740 FF
5280 1741 60

; BYTE 'S S A B C D E F'

; BYTE CR,LF,LF,ETX

; RTS

; ******************************************

; DUMP ONE LINE TO PRINTER

; ******************************************
5440 ;
5450 ;
5460 ;
5470 ;
5480 ;
5490 1742 207214 PRLINE JSR CR.LF
5500 1745 AD0512 LDA SELECT DETERMINE STARTING COLUMN.
5510 1749 48 PHA FOR THIS DUMP.
5520 1749 290F AND #0F
5530 174A 8D5615 STA COLUMN NOW COLUMN HOLDS NUMBER OF
5540 ; HEX COLUMN IN WHICH WE DUMP
5550 ; THE FIRST BYTE.
5560 174C 68 PLA SET SELECT=BEGINNING OF A
5570 174F 29F0 AND #$F0 HEX LINE.
5580 1751 8D0512 STA SELECT PRINT LINE'S START ADDRESS.
5590 1754 20A115 JSR PR.ADR SPACE 3 TIMES--TO THE
5600 1757 A203 LDX #3 FIRST HEX COLUMN.
5610 1759 209614 JSR SPACES
5620 ;
5630 ;
5640 175C AD5615 LDA COLUMN DO WE DUMP FROM THE FIRST
5650 ; HEX COLUMN?
5660 175F F00D BEQ COL.OK IF SO, WERE AT THE CORRECT
5670 ; COLUMN NOW.
5680 ;
5690 1761 A203 LOOP LDIX #3 IF NOT, SPACE 3 TIMES FOR
5700 1763 209614 JSR SPACES EACH BYTE NOT DUMPED.
5710 1766 200D13 JSR INC.SL
5720 1769 CE5615 DEC COLUMN
5730 176C D0F3 BNE LOOP
5740 ;
5750 176E 209A15 COL.OK JSR DUMPSL DUMP SELECTED BYTE.
5760 1771 207D14 JSR SPACE SPACE ONCE.
5770 1774 208317 JSR NEXTSL SELECT NEXT BYTE
5780 ;
5790 1777 3009 BMI EXIT MINUS MEANS WE'VE DUMPED
5800 ; THROUGH TO THE END ADDRESS.
5810 ;
5820 ;
5830 1779 AD0512 NOT.EA LDA SELECT DUMPED ENTIRE LINE?
5840 177C 290F AND #$F0 (4LSB OF SELECT=0?)
5850 177E C300 CMP #0 IF SO, WE'VE DUMPED THE
5860 ; ENTIRE LINE. IF NOT,
5870 1780 D0EC BNE COL.OK SELECT NEXT BYTE AND DUMP IT.
5880 1782 60 EXIT RTS RETURN MINUS IF EA DUMPED;
5890 ; RETURN PLUS IF EA NOT DUMPED.
5900 ;
5910 ;
5920 ;
5930 ;
5940 ;
5950 ;
5960 ;
5970 ;
5980 ;
5990 ;
6000 ;
6010 ;

272 BEYOND GAMES
SELECT NEXT BYTE (IF < END ADDRESS)

***********

6100 1783 38   NEXTSL SEC
6110 1784 ADD612  LDA SELECT+i HIGH BYTE OF SELECT LESS
6120 1787 CD5515  CMP EA+i THAN HIGH BYTE OF EA?
6130 178A 900B   BCC SL.OK  IF SO, SELECT<END ADDRESS.
6140 178C D00F   BNE NO.INC IF SELECT>EA, DON'T
6150 ;                INCREMENT SELECT.
6160 ;
6170 178E 38   SEC SELECT IS IN SAME PAGE AS EA.
6180 178F ADD512  LDA SELECT
6190 1792 CD5415  CMP EA
6200 1795 B006   BCS NO.INC
6210 ;
6220 1797 200113 SL.OK JSR INC.SL SINCE SELECT <= EA, WE MAY
6230 ;                INCREMENT SELECT.
6240 ;
6250 179A 8000   LDA $0 SET "INCREMENTED" RETURN
6260 179C 60    RTS CODE AND RETURN.
6270 ;
6280 179D A0FF   NO.INC LDA $IFF SET "NO INCREMENT" RETURN
6290 179F 60    RTS CODE AND RETURN.
6300 ;
6310 ;
6320 ;
6330 ;
6340 ;
6350 ;
6360 ;***********
6370 ;
6380 ; SELECT START ADDRESS
6390 ;***********
6400 ;
6410 ;
6420 ;
6430 ;
6440 ;
6450 ;
6460 17A0 AD5215 GOTOSA LDA SA SET SELECT=SA.
6470 17A3 BD6512 STS SELECT
6480 17A4 AD5315 LDA SA+i
6490 17A5 BD6512 STS SELECT+i
6500 17AC 60    RTS RETURN w?SELECT=SA.
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 BEYOND GAMES: SYSTEM
SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

BY KEN SKIER

; **********************************************

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

; **********************************************
590 ;
600 ;
610 ;
620 ;
630 ;
640 1200= ;
650 ; VMPAGE=$1200  STARTING PAGE OF VISIBLE
650 1205= ; MONITOR CODE.
660 1207= ; SELECT=VMPAGE+5
670 1254= ; VISMON=VMPAGE+7
680 1300= ; GET.SL=VMPAGE+$94
690 130D= ; INC.SL=VMPAGE+$10D
700 131A= ; DEC.SL=VMPAGE+$11A
710 ;
720 ;
730 1400= ; FRPAGE=$1400  STARTING PAGE OF PRINT
740 ; UTILITIES.
750 1408= ; TUT.ON=FRPAGE+8
760 140E= ; TUTOFF=FRPAGE+$0E
770 1414= ; PR.ON =FRPAGE+$14
780 141A= ; PR.OFF=FRPAGE+$1A
790 1440= ; PR.CH=FRPAGE+$40
800 1472= ; CR.LF =FRPAGE+$72
810 147D= ; SPACE =FRPAGE+$7D
820 1496= ; SPACES=FRPAGE+$96
830 1483= ; FR.BYT=FRPAGE+$B3
840 14E4= ; PRINT=FRPAGE+$E4
850 1512= ; PUSHSL=FRPAGE+$112
860 152B= ; POP.SL=FRPAGE+$12B
870 ;
880 ;
890 1500= ; HEX.PG=$1500  ADDRESS OF PAGE IN WHICH
890 ; HEXDUMP CODE STARTS.
900 ;
910 ;
920 1552= ; SA=HEX.PG+$52
930 1554= ; EA=SA+2
940 159A= ; DUMP=HEX.PG+$9A
950 15A1= ; PR.ADR=HEX.PG+$A1
960 16DF= ; RANGE=HEX.PG+$DF
970 15E9= ; SETADS=HEX.PG+$E9
980 17B3= ; NEXTSL=HEX.PG+$2B3
990 17A0= ; GOTOSA=HEX.PG+$2A0
1000 ;
1010 ;
1020 ;
1030 ;
1040 ;
1050 ;
1060 ;
1070 ;
1080 ;
1090 1900= ; DSPAGE=$1900  STARTING PAGE OF DISASSEMBLER
1100 ;
1110 1B18= ; SUBS =DSPAGE+$21B
1120 1B58= ; MNAMES=DSPAGE+$258
1130 1C09= ; MCODES=DSPAGE+$308
1140 1D00= ; MCODES=DSPAGE+$408
1150 ;
1160 ;

278 BEYOND GAMES
1170
1150
1130
1200
1210
1220
1230
1240
1250
1260
1270 1300
1280
1290
1300
1310
1320
1330 1900 05
1340
1350
1360 1901 00
1370
1380 1902 00
1390
1400
1410 1903 00
1420
1430 1904 0000
1440
1450
1460 1905 00
1470
1480 1906 00
1490
1500 1907 00
1510
1520
1530
1540
1550
1560
1570
1580
1590
1600
1610
1620
1630
1640
1650
1660
1670
1680
1690
1700
1710
1720
1730
1740 1909 200814

;******************************************************************************

VARIABLES

;******************************************************************************

*DSPAGE

;******************************************************************************

DISLNS .BYTE 5

NUMBER OF LINES TO BE

DISASSEMBLED BY TV.DIS.

;******************************************************************************

LINUM .BYTE 0

DATA CELL: USED BY TV.DIS.

;******************************************************************************

LETTER .BYTE 0

COUNTS LETTERS PRINTED IN

A MNEMONIC. USED BY MNEMON.

;******************************************************************************

TEMP.X .BYTE 0

DATA CELL USED BY MNEMON.

;******************************************************************************

SUBPTR .WORD 0

POINTER TO A SUBROUTINE.

SET, USED BY MODE.X

;******************************************************************************

OPBYTES .BYTE 0

DATA CELL: USED BY FINISH.

;******************************************************************************

OPCHRS .BYTE 0

DATA CELL: USED BY FINISH.

;******************************************************************************

ADRCOL .BYTE 16

STARTING COLUMN FOR ADDRESS

FIELD. OSI C-IP OWNERS:

FOR NARROW FORMAT, SET

ADRCOL=80B. SEE NOTES

IN LISTING FOR ADDRESS MODE

SUBROUTINES.)

;******************************************************************************

;******************************************************************************

TV-DISASSEMBLER

;******************************************************************************

;******************************************************************************

SELECT SCREEN FOR OUTPUT.
1750 190C AD0019  LDA DISLNS  INITIALIZE LINE COUNTER WITH
1760 190F 8D0119  STA LINUM  # OF LINES TO DISASSEMBLE.
1770  ;
1780 1912 A9FF  LDA #$FF  SET END ADDRESS TO $FFFF,
1790 1914 8D5415  STA EA  SO NEXTSL WILL ALWAYS
1800 1917 8D5515  STA EA+1  INCREMENT SELECT POINTER.
1810 191A 207214  JSR CR.LF  ADVANCE TO A NEW LINE.
1820  ;
1830 191D 207D19 TVLOOP JSR D SLINE  DISASSEMBLE ONE LINE.
1840 1920 CE0119  DEC LINUM  DONE LAST LINE YET?
1850 1923 D0F8  ENP TVLOOP  IF NOT, DO NEXT ONE.
1860 1925 60  RTS  IF SO, RETURN.
1870  ;
1880  ;
1890  ;
1900  ;
1910  ;
1920  ;
1930  ;
1940  ;
1950  ;
1960  ;
1970  ;***********************************************************
1980  ;
1990  ; PRINTING DISASSEMBLER
2000  ;***********************************************************
2010  ;***********************************************************
2020  ;
2030  ;
2040  ;
2050  ;
2060  ;
2070 1926 201A14 PR.DIS,JSR P R.OFF  DE-SELECT PRINTER
2080 1929 200B14 JSR TUT.ON  SELECT SCREEN FOR OUTPUT.
2090 192C 20E414 JSR PRINT:  DISPLAY TITLE.
2100 192F 7F  .BYTE TEX,CR,LF
2100 1930 0D
2100 1931 0A
2110 1932 20  .BYTE 'PRINTING DISASSEMBLER.'
2110 1933 20
2110 1934 20
2110 1935 20
2110 1936 20
2110 1937 50
2110 1938 52
2110 1939 49
2110 193A 4E
2110 193B 54
2110 193C 49
2110 193D 4E
2110 193E 47
2110 193F 20
2110 1940 44
2110 1941 49
2110 1942 53
2110 1943 41
2110 1944 53
2110 1945 53

280 BEYOND GAMES
2110 1946 45
2110 1947 4D
2110 1948 42
2110 1949 4C
2110 194A 45
2110 194B 52
2110 194C 2E
2120 ;
2130 194D 0D .BYTE CR,LF,ETX
2130 194E 0A
2130 194F FF
2140 ;
2150 1950 20E915 JSR SETADS LET USER SET START, END
2160 ; ADDRESSES OF MEMORY TO BE
2170 ; DISASSEMBLED.
2180 1953 201414 JSR PR.ON SELECT PRINTER FOR OUTPUT.
2190 1956 20E414 JSR PRINT:
2200 1959 7F .BYTE T艾,CR,LF
2200 195A 0D
2200 195B 0A
2210 195C 44 .BYTE 'DISASSEMBLING'
2220 195D 49
2230 195E 53
2240 195F 41
2250 1960 53
2260 1961 53
2270 1962 45
2280 1963 4D
2290 1964 42
22A0 1965 4C
22B0 1966 49
22C0 1967 4E
22D0 1968 47
22E0 1969 20
22F0 196A FF .BYTE ETX
2300 196B 200F16 JSR RANGE PRINT RANGE OF MEMORY TO
2310 ; BE DISASSEMBLED.
2320 196C 20A017 JSR GOTOASA SET SELECT=START OF BLOCK.
2330 ;
2340 1971 207214 JSR CR.LF ADVANCE TO A NEW LINE.
2350 1974 207D19 PRL0OP JSR DLINES DISASSEMBLE ONE LINE.
2360 1977 10FB BPL PRL0OP IF IT WASN'T THE LAST LINE.
2370 ; DISASSEMBLE THE NEXT ONE.
2380 ;
2390 1979 201A14 JSR PR.OFF DE-SELECT PRINTER FOR OUTPUT.
23A0 ;
23B0 197C 5B RTS RETURN TO CALLER.
23C0 ;
23D0 ;
23E0 ;
23F0 ;
2400 ;
2410 ;
2420 ;
2430 ;
2440 ;
2450 ;
; DISASSEMBLE ONE LINE.

2560 197D 209412 DSIINE JSR GET.SL GET CURRENTLY-SELECTED BYTE.
2570 1980 48 PHA SAVE IT ON STACK.
2580 1981 209219 JSR MNEMON PRINT MNEMONIC REPRESENTED
2590 ; BY THAT OPCODE.
2600 1984 207D14 JSR SPACE SPACE ONCE.
2610 1987 60 PLA RESTORE OPCODE.
2620 1988 20AF19 'JSR OPERND PRINT OPERAND REQUIRED BY
2630 ; THAT OPCODE.
2640 1988 20011A JSR FINISH FINISH THE LINE BY PRINTING
2650 ; FIELDS 3-6. FINISH LEAVES
2660 ; SELECT POINTING TO LAST
2670 ; BYTE OF INSTRUCTION.
2680 ;
2690 198E 209317 JSR NEXTSL SELECT NEXT BYTE, IF
2700 ; SELECT=EA.
2710 1991 60 RTS RETURN W/RETURNCODE FROM
2720 ; NEXTSL. SELECT POINTS TO
2730 ; NEXT OPCODE, OR SELECT=EA.
2740 ;
2750 ;
2760 ;
2770 ;
2780 ;
2790 ;
2800 ;
2810 ;
2820 ;
2830 ;
2840 ;
2850 ;
2860 ;
2870 ;
2880 ;
2890 ;
2900 ;
2910 ;
2920 1992 A203 MNEMON LDX #3 WE'LL PRINT THREE LETTERS.
2930 1994 8E0219 STX LETTER PREPARE TO USE OPCODE AS AN
2940 1997 AA TAX INDEX.
2950 ; LOOK UP MNEMONIC CODE FOR
2960 ; THAT OPCODE. MCODES IS
2970 ; TABLE OF MNEMONIC CODES.
2980 ;
2990 ;
3000 ; PREPARE TO USE THAT MNEMONIC
3010 193B AA TAX CODE AS AN INDEX.
3020 ; GET A MNEMONIC CHARACTER.
3030 193C BD501B MNLOOP LDA MNAMES,X

282 BEYOND GAMES
3040 ; (MNAME IS A LIST OF
3050 ; MNEMONIC NAMES.)
3060 ;
3070 199F 8E0319 STX TEMP.X SAVE X-REGISTER, SINCE
3080 ; PRINTING MAY CHANGE X.
3090 19A2 2B4014 JSR PR.CHR PRINT THE MNEMONIC CHARACTER.
3100 19A5 AE0319 LDX TEMP.X RESTORE X,
3110 19A8 E8 INX ADJUST INDEX FOR NEXT LETTER.
3120 19A9 CE0219 DEC LETTER PRINTED 3 LETTERS YET?
3130 19AC DBEE BNE MNLOOP IF NOT, PRINT NEXT ONE.
3140 19AE 60 RTS IF SO, RETURN TO CALLER.
3150 ;
3160 ;
3170 ;
3180 ;
3190 ;
3200 ;
3210 ;
3220 ;
3230 ;
3240 ;
3250 ;*************************************************************
3250 ; PRINT OPERAND
3260 ;
3270 ;*************************************************************
3280 ;
3290 ;
3300 ;
3310 ;
3320 ;
3330 ;
3340 ;
3350 19AF AA OPERND TAX LOOK UP ADDRESSING MODE
3360 19B0 BD001D LDA MODES,X CODE FOR THIS OPCODE.
3370 ;
3380 19B3 AA TAX X NOW INDICATES ADDRESSING
3390 ; MODE.
3400 ;
3410 19B4 2B8819 JSR MODE.X HANDLE THAT ADDRESSING MODE.
3420 19B7 60 RTS RETURN TO CALLER.
3430 ;
3440 ;
3450 ;
3460 ;
3470 ;
3480 ;
3490 ;
3500 ;
3510 ;
3520 ;
3530 ;*************************************************************
3540 ; HANDLE ADDRESSING MODE "X"
3550 ;*************************************************************
3560 ;
3570 ;
3580 ;
3590 ;
3600 ;
3610 ;

283
3620 ; 
3630 ;
3640 19B8 BD1B1B MODE.X LDA SUBS.X STA SUBPTR
3650 19B8 8D0419 GET LOW BYTE OF Xth POINTER
3660 ; IN TABLE OF SUBROUTINE
3670 19E8 80 INX
3680 19BF BD1B1B ADJUST INDEX FOR NEXT BYTE.
3690 19C2 8D0519 GET HIGH BYTE OF POINTER.
3700 19C5 6C0419 JUMP TO SUBROUTINE SPECIFIED
3710 ; BY SUBROUTINE POINTER.
3720 ; THAT SUBROUTINE WILL RETURN
3730 ; TO THE CALLER OF MODE.X,
3740 ; NOT TO MODE.X ITSELF.
3750 ;
3760 ;
3770 ;
3780 ;
3790 ;
3800 ;
3810 ;
3820 ;
3830 ;
3840 ;
3850 ;
3860 ;
3870 ;
3880 ;
3890 ;
3900 ;
3910 ;
3920 ;
3930 ;
3940 ;
3950 ;
3960 ;
3970 ;
3980 ;
3990 19C9 200D13 ONEBYT JSR INC.SL ADVANCE TO BYTE FOLLOWING
4000 ; OPCODE.
4010 19CB 209A15 JSR DUMPSL DUMP THAT BYTE.
4020 19CE 68 RTS RETURN TO CALLER.
4030 ;
4040 ;
4050 ;
4060 ;
4070 ;
4080 ;
4090 ;
4100 ;
4110 ;
4120 19CF 200D13 TWOBYT JSR INC.SL ADVANCE TO FIRST BYTE OF
4130 ; OPERAND.
4140 19D2 209412 JSR GET.SL LOAD THAT BYTE INTO ACC.
4150 19D5 48 PHA SAVE IT.
4160 19D5 200D13 JSR INC.SL ADVANCE TO 2ND BYTE OF
4170 ; OPERAND.
4180 19D9 209A15 JSR DUMPSL DUMP IT.
4190 19DC 68 PLA RESTORE FIRST BYTE TO ACC.

284 BEYOND GAMES
4200 1900 208314          JSR PR.BYT           DUMP IT.
4210 19E0 60              RTS               RETURN TO CALLER.
4220
4230
4240
4250
4260
4270  ; PRINT LEFT, RIGHT PARENTHESES
4280
4290
4300
4310 19E1 A920          LPAREN LDA #'(  
4320 19E3 D002          BNE SENDIT
4330  
4340
4350 19E5 A929          RPAREN LDA #' )
4360
4370 19E7 204014 SENDIT JSR PR.CHR
4380 18EA 60              RTS
4390
4400
4410
4420
4430
4440  ; PRINT A COMMA AND AN "X"
4450
4460
4470
4480 19EB A92C        XINDEX LDA #' ,
4490 19ED 204014          JSR PR.CHR           PRINT A COMMA.
4500 19F0 A958          LDA #'X
4510 19F2 204014          JSR PR.CHR           PRINT AN "X".
4520 19F5 60              RTS
4530
4540
4550
4560
4570
4580  ; PRINT A COMMA AND A "Y"
4590
4600
4610
4620 19F6 A92C        YINDEX LDA #' ,
4630 19F8 204014          JSR PR.CHR           PRINT COMMA.
4640 19FB A959          LDA #'Y
4650 19FD 204014          JSR PR.CHR           PRINT A "Y".
4660 1A00 60              RTS
4670
4680
4690
4700
4710
4720
4730
4740
4750
4760
4770  ; **************************************************
4720 1A01 8D0719 FINISH STA OPCHRS
4730 1A04 6E0619 STX OPBYTES
4740 ;
4750 ; NOTE: EVERY ADDRESSING MODE
4760 ; SUBROUTINE MUST END BY
4770 ; SETTING X=# OF BYTES IN
4780 ; OPERAND, AND ACC=# OF
4790 ; CHARACTERS IN OPERAND.
4800 ;
4810 ;
4820 1A05 3006 SAVE THE LENGTH OF THE
4830 1A0A 201A13 OPERAND, IN CHARACTERS AND
4840 1A0D CA AND IN BYTES. 0 MEANS NO
4850 1A0E 10FA OPERAND.
4860 ;
4870 1A07 CA IF NECESSARY, DECREMENT THE
4880 ; SELECT POINTER SO IT POINTS
4890 ; TO THE OPCODE.
4900 ;
4910 ;
4920 1A08 3005 NOW SELECT POINTS TO OPCODE.
4930 1A0A 201A13 JSR DEC.SL
4940 1A0D CA DEX
4950 1A0E 10FA
4960 ;
4970 1A07 CA
4980 ;
4990 ;
5000 1A10 88 SEL.OK PHP
5010 1A11 88 CLD
5020 1A12 30 SEC
5030 1A13 AD0819 LDA ADDR+1
5040 1A16 E304 SBC #4
5050 ;
5060 1A19 ED0719 SBC OPCHRS
5070 1A10 28 PLP
5080 1A1A 28 TAX
5090 1A1B 209614 JSR SPACES
5100 ;
5110 1A1C 20A115 JSR PR.ADR
5120 ;
5130 1A1E 207D14 LOOP.Z JSR SPACE
5140 1A21 206A15 JSR DUMPSL
5150 1A22 200D13 JSR INC.SL
5160 1A23 CE0619 DEC OPBYTES
5170 1A24 10F2 BPL LOOP.Z
5180 1A25 201A13 JSR DEC.SL
5190 ;
5200 1A26 209915 JSR SPACE
5210 1A27 200813 JSR DUMPSL
5220 1A28 CE0619 DEC OPBYTES
5230 1A2F 10F2 BPL LOOP.Z
5240 1A31 201A13 JSR DEC.SL
5250 ;
5260 1A32 207D14 FINISH JSR CR.LF
5270 ;
5280 1A33 60 RTS
5290 ;
5300 ;
5310 ;
5320 1A34 207214 HAVING DISASSEMBLED ONE LINE.
5330 ;
5340 1A37 60 GO TO A NEW LINE.
5350 ;
5360 1A38 60 RETURN TO CALLER.
Appendix C7:

Table-Driven Disassembler
(Addressing Mode Subroutines)
APPENDIX C7: ASSEMBLER LISTING OF
TABLE-DRIVEN DISASSEMBLER:

ADDRESSING MODE SUBROUTINES

SEE CHAPTER 9 OF BEYOND GAMES: SYSTEM
SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

BY KEN SKIER

*******************************************************************

CONSTANTS

*******************************************************************

CR = $0D  CARRIAGE RETURN.
LF = $0A  LINE FEED.
TEXT = $7F  THIS CHARACTER MUST START ANY MESSAGE.
ETX = $FF  THIS CHARACTER MUST END ANY MESSAGE.
580 ;
590 ;
600 ;
610 ;
620 ;
630 ;
640 ;
650 ;
660 ;
670 ;
680 ;
690 ;
700 ;
710 ;
720 ;
730 ;
740 ;
750 ;
760 ;
770 ;
780 ;
790 ;
800 ;
820 ;
830 1203= ;
840 ;
850 1205= ;
860 1207= ;
870 1234= ;
880 1301= ;
890 131A= ;
8A0 ;
8B0 ;
8C0 ;
8D0 ;
8E0 ;
8F0 ;
900 1400= ;
910 ;
920 1440= ;
930 1472= ;
940 147D= ;
950 1496= ;
960 14B3= ;
970 14E4= ;
980 1512= ;
990 152B= ;
1000 ;
1010 ;
1020 1500= ;
1030 ;
1040 ;
1050 15A1= ;
1060 17B3= ;
1070 ;
1080 ;
1090 1900= ;
1100 ;
1110 19C8= ;
1120 19CF= ;
1130 19E1= ;
1140 19E5= ;
1150 19EB= ;

;*****************************************************************************

EXTERNAL ADDRESSES

;*****************************************************************************

UMPAGE=$1200  STARTING PAGE OF VISIBLE
      MONITOR CODE.

SELECT=UMPAGE+5
VISMON=UMPAGE+7
GET.SL=UMPAGE+$94
INC.SL=UMPAGE+$100
DEC.SL=UMPAGE+$11A

PRPAGE=$1400  STARTING PAGE OF PRINT
      UTILITIES.

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=$1500  ADDRESS OF PAGE IN WHICH
      HEXDUMP CODE STARTS.

PR.ADR=HEX.PG+$A1
NEXTSL=HEX.PG+$2B3

D$PAGE=$1900  START OF DISASSEMBLER CODE.

ONEBYT=D$PAGE+$C8
TWOBYT=D$PAGE+$CF
LPAREN=D$PAGE+$E1
RPAREN=D$PAGE+$E5
XINDEX=D$PAGE+$EB

280 BEYOND GAMES
1160 19F6= YINDEX=D$PAGE+$F6
1170 ;
1180 ;
1190 ;
1200 ;
1210 ;
1220 ;
1230 ;
1240 1A40 ==D$PAGE+$140
1250 ;
1260 ;
1270 ;
1280 ;
1290 ;
1300 ;
1310 ;
1320 ;
1330 ;
1340 ;
1350 ;
1360 ;******************************
1370 ;
1380 ;
1390 ;
1400 ;******************************
1410 ;
1420 ;
1430 ;
1440 ;
1450 ;
1460 ;
1470 ;
1480 ;
1490 ;
1500 ;
1510 1A40 20CF19 ABSLUT JSR TWOBYT PRINT A TWO-BYTE OPERAND.
1520 1A43 A282 LDX #2 OPERAND HAS TWO BYTES...
1530 1A45 A904 LDA #4 ...AND FOUR CHARACTERS.
1540 1A47 E0 RTS RETURN TO CALLER.
1550 ;
1560 ;
1570 ;
1580 ;
1590 ;
1600 ;
1610 ;
1620 ;
1630 ;
1640 1A48 20481A ABS.X JSR ABSLUT
1650 1A48 20EB19 JSR XINDEX PRINT A COMMA AND AH "X".
1660 1A4E A282 LDX #2 OPERAND HAS 2 BYTES...
1670 1A50 A906 LDA #6 ...AND SIX CHARACTERS.
1680 1A52 E0 RTS RETURN TO CALLER.
1690 ;
1700 ;
1710 ;
1720 ;
1730 ;
1740 ; ABSOLUTE MODE
1750 ;
1760 ;
1770 ;
1780 1A53 20401A ABS.Y JSR ABSLUT
1790 1A55 20FB19 JSR YINDEX
1800 1A53 A202 LDX #$2
1810 1A55 A905 LDA #$6
1820 1A50 60 RTS
1830 ;
1840 ;
1850 ;
1860 ;
1870 ;
1880 ;
1890 ;
1900 ;
1910 1A5E A841 ACC LDA #$A PRINT THE LETTER "A"
1920 1A60 204014 JSR PR.CHAR
1930 1A63 A200 LDX #$0 OPERAND HAS NO BYTES...
1940 1A55 A901 LDA #$1 ...AND ONE CHARACTER.
1950 1A57 60 RTS RETURN TO CALLER.
1960 ;
1970 ;
1980 ;
1990 ;
2000 ;
2010 ;
2020 ;
2030 ;
2040 ;
2050 1A68 A200 IMPLID LDX #$0 OPERAND HAS NO BYTES...
2060 1A6A A900 LDA #$0 ...AND NO CHARACTERS.
2070 1A6C 60 RTS
2080 ;
2090 ;
2100 ;
2110 ;
2120 ;
2130 ;
2140 ;
2150 ;
2160 ;
2170 1A6D A823 IMMEDT LDA #$ PRINT A "#" CHARACTER.
2180 1A6F 204014 JSR PR.CHAR
2190 ;
2200 1A72 A924 LDA #$" PRINT A DOLLAR SIGN TO
2210 1A74 204014 JSR PR.CHAR INDICATE HEXADECIMAL.
2220 1A77 20C319 JSR ONEBYTE PRINT ONE-BYTE OPERAND IN
2230 ; HEXADECIMAL FORMAT.
2240 1A7A A201 LDX #$1 OPERAND HAS ONE BYTE...
2250 1A7C A904 LDA #$4 ...AND FOUR CHARACTERS.
2260 1A7E 60 RTS RETURN TO CALLER.
2270 ;
2280 ;
2290 ;
2300 ;
2310 ;
INDIRECT MODE

2320 ;
2330 ;
2340 ;
2350 ;
2360 1A7F 20E119 INDIRECT JSR LPAREN PRINT LEFT PARENTHESIS.
2370 1A82 20481A JSR ABSLUT PRINT TWO-BYTE OPERAND.
2380 1A85 20E519 JSR RPAREN PRINT RIGHT PARENTHESIS.
2390 1A88 A906 LDA $6 A HOLDS NUMBER OF CHARACTERS
2400 ; IN OPERAND.
2410 1A8A A202 LDX #2 X HOLDS NUMBER OF BYTES IN
2420 ; OPERAND.
2430 1A8C 60 RTS RETURN TO CALLER.
2440 ;
2450 ;
2460 ;
2470 ;
2480 ;
2490 ;
2500 ;
2510 ;
2520 ;
2530 1A8D 20E119 IND.X JSR LPAREN PRINT A ZERO PAGE ADDRESS.
2540 1A90 20E81A JSR ZERO.X A COMMA, AND THE LETTER "X".
2550 ;
2560 1A93 20E519 JSR RPAREN ONE BYTE IN OPERAND.
2570 1A95 A201 LDX #1 8 CHARACTERS IN OPERAND.
2580 1A98 A908 LDA $8 (C-IP OWNERS: A8 06, NOT
2590 ; A9 06, FOR NARROW FORMAT.)
2600 ;
2610 1A9A 60 RTS
2620 ;
2630 ;
2640 ;
2650 ;
2660 ;
2670 ;
2680 ;
2690 ;
2700 ;
2710 1A9B 20E119 IND.Y JSR LPAREN PRINT A ZERO PAGE ADDRESS.
2720 1A9E 20B81A JSR ZERO.FS PRINT A COMMA AND A "Y".
2730 1A9F 20E519 JSR RPAREN OPERAND HAS 1 BYTE...
2740 1AA4 20F619 JSR YINDEX ...AND 8 CHARACTERS.
2750 1AA7 A201 LDX #1 (C-IP OWNERS: A8 06, NOT
2760 1A99 A908 LDA $8 A9 06, FOR NARROW FORMAT.)
2770 ;
2780 ;
2790 1A9B 60 RTS
2800 ;
2810 ;
2820 ;
2830 ;
2840 ;
2850 ;
2860 ;
2870 ;
2880 ;
2890 1AAC 200D13 RELATV JSR INC.SL SELECT NEXT BYTE.

RELATIVE MODE
SAVE SELECT POINTER ON STACK.
GET OPERAND BYTE.
SAVE IT ON STACK.
INCREMENT SELECT POINTER
SO IT POINTS TO NEXT OPCODE.
(RELATIVE BRANCHES ARE
RELATIVE TO NEXT OPCODE.)
RESTORE OPERAND BYTE TO ACC.
IS IT PLUS OR MINUS?
IF PLUS, IT MEANS A FORWARD
BRANCH.

OPERAND IS MINUS, SO WE'LL
BRANCH BACKWARD.
BRANCHING BACKWARD IS LIKE
BRANCHING FORWARD FROM ONE
PAGE LOWER IN MEMORY.

SAVE CALLER'S DECIMAL FLAG.
CLEAR DECIMAL MODE, FOR
BINARY ADDITION.
PREPARE TO ADD.
ADD OPERAND BYTE TO SELECT.

NOW SELECT POINTS TO ADDRESS
SPECIFIED BY RELATIVE
BRANCH INSTRUCTION.
RESTORE CALLER'S DECIMAL
FLAG.
PRINT ADDRESS SPECIFIED
BY INSTRUCTION.
RESTORE SELECT=ADDRESS OF
OPERAND.
OPERAND HAD ONE BYTE...
AND FOUR CHARACTERS.
RETURN TO CALLER.

ZERO PAGE MODE

PRINT TWO ASCII ZERO'S TO
ALL SELECTED BYTES.
(C-IP OWNERS: SUBSTITUTE NOPs
--EA EA EA--FOR JSR PR.BYT,
TO GET NARROW FORMAT.)
PRINT ONE-BYTE OPERAND.
OPERAND HAS ONE BYTE...
...AND FOUR CHARACTERS.
(C-IP OWNERS: A9 02,
NOT A9 04, FOR NARROW FORMAT.)
ZERO PAGE, X MODE

3570 1AE8 20DB1A ZERO,X JSR ZEROPG
3580 1AE8 20EB19 JSR XINDEX
3590 1AEE A201 LDX #1
3600 1AF0 A906 LDA #6
3610 ;
3620 ;
3630 1AF2 60 RTS
3640 ;
3650 ;
3660 ;
3670 ;
3680 ;
3690 ;
3700 ;
3710 ;
3720 ;
3730 1AF3 20DB1A ZERO,Y JSR ZEROPG
3740 1AF6 20F619 JSR YINDEX
3750 1AF9 A201 LDX #1
3760 1AFB A906 LDA #6
3770 ;
3780 1AFD 60 RTS
3790 ;
3800 ;
3810 ;
3820 ;
3830 ;
3840 ;
3850 ;
3860 ;
3870 ;
3880 ;
3890 ;
3900 ;
3910 ;
3920 ;
3930 ;
3940 ;
3950 ;
3960 ;
3970 ;
3980 ;
3990 ;
4000 ;
4010 ;
4020 ;
4030 ;
4040 ;
4050 ;
0

*******

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.

4050 1AFE 68 TXMODE PLA POP RETURN ADDRESS TO
4056 1AFF 68 PLA OPERND.
4060 ;
4062 1B00 68 PLA POP RETURN ADDRESS TO
4064 1B01 68 PLA DLINE.
4068 ;
4070 1B02 208317 JSR NEXTSL NOW DLINE'S CALLER IS ON
4076 1B05 3000 BMI TXEXIT THE STACK.
4078 1B07 209412 JSR GET.SL
4080 1B0A 97FF CMP *ETX
4082 1B0C F005 BEQ TXEXIT IS IT END OF TEXT?
4084 1B0E 204014 JSR PR.CHR IF SO, STRING ENDED.
4086 1B11 18 CLC IF NOT, PRINT CHARACTER.
4088 1B12 90EE BCC TXMODE+4 BRANCH BACK TO GET NEXT
408A ; CHARACTER.

4090 ;
4092 1B14 207214 TXEXIT JSR CR.LF ADVANCE TO A NEW LINE.
4098 1B17 208317 JSR NEXTSL ADVANCE TO NEXT OPCODE.
409A 1B1A 60 RTS RETURN TO CALLER OF DLINE.

;********************************************************************
; TABLE OF ADDRESSING MODE SUBROUTINES
;********************************************************************
ADDRESSING MODE 0 IS INVALID, HENCE IMPLIED.
Appendix C8:

Table-Driven Disassembler (Tables)
APPENDIX C8: ASSEMBLER LISTING OF
TABLE-DRIVEN DISASSEMBLER

TABLES

SEE CHAPTER 9 OF BEYOND GAMES: SYSTEM
SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

BY KEN SKIER

; *****************************************************************
;
; CONSTANTS
;
; *****************************************************************
;
370 007F=   TEX = $7F  THIS CHARACTER MUST START
380         ANY MESSAGE.
390 00FF=   ETX = $FF  THIS CHARACTER MUST END
400         ANY MESSAGE.
410 420 430 440 450 460 470 480 490 500 510 520 530 540 550 560 570 ;
PAGE 1300 STARTING PAGE OF DISASSEMBLER

LIST OF MNEMONICS

Since this table is a string of characters, start it with the TEXT pseudo-op.

MACHINES .BYTE TEX

840 1B51 42
840 1B52 41
840 1B53 44
850 1B54 41
850 1B55 44
850 1B56 43
860 1B57 41
860 1B58 4E
860 1B59 44
870 1B5A 41
870 1B5B 53
870 1B5C 4C
880 1B5D 42
880 1B5E 43
880 1B5F 43
890 1B60 42
890 1B61 43
890 1B62 53
900 1B63 42
900 1B64 45
900 1B65 51
910 1B66 42
910 1B67 49
910 1B68 54
920 1B69 42
920 1B6A 4D
920 1B6B 49
930 1B6C 42
930 1B6D 4E
930 1B6E 45
940 1B6F 42
940 1B70 50

302 BEYOND GAMES
| 1140 | 1BAB 4C | .BYTE 'LDA' |
| 1148 | 1BAC 44 | | 
| 1148 | 1BAD 41 | | 
| 1150 | 1BAE 4C | .BYTE 'LDX' |
| 1150 | 1BAF 44 | | 
| 1150 | 1BB0 58 | .BYTE 'LDY' |
| 1160 | 1BB1 4C | | 
| 1160 | 1BB2 44 | | 
| 1160 | 1BB3 59 | | 
| 1170 | 1BB4 4C | .BYTE 'LSR' |
| 1170 | 1BB5 53 | | 
| 1170 | 1BB6 52 | | 
| 1180 | 1BB7 4E | .BYTE 'NOP' |
| 1180 | 1BB8 4F | | 
| 1180 | 1BB9 50 | | 
| 1190 | 1BBA 4F | .BYTE 'ORA' |
| 1190 | 1BBB 52 | | 
| 1190 | 1BBC 41 | | 
| 1200 | 1BBD 50 | .BYTE 'PHA' |
| 1200 | 1BEE 48 | | 
| 1200 | 1BFF 41 | | 
| 1210 | 1B08 50 | .BYTE 'PHP' |
| 1210 | 1B09 4F | | 
| 1220 | 1B0A 50 | | 
| 1220 | 1B0B 4F | .BYTE 'PLP' |
| 1220 | 1B0C 4C | | 
| 1230 | 1B0D 41 | | 
| 1230 | 1B0E 50 | | 
| 1230 | 1B0F 4C | | 
| 1240 | 1B10 50 | .BYTE 'ROL' |
| 1240 | 1B11 4F | | 
| 1240 | 1B12 4C | | 
| 1250 | 1B13 52 | .BYTE 'ROR' |
| 1250 | 1B14 4F | | 
| 1250 | 1B15 4C | | 
| 1260 | 1B16 52 | .BYTE 'RTI' |
| 1260 | 1B17 52 | | 
| 1250 | 1B18 54 | .BYTE 'RTS' |
| 1260 | 1B19 49 | | 
| 1270 | 1B1A 52 | | 
| 1270 | 1B1B 54 | .BYTE 'SEC' |
| 1270 | 1B1C 53 | | 
| 1280 | 1B1D 53 | | 
| 1260 | 1B1E 42 | | 
| 1280 | 1B1F 43 | | 
| 1250 | 1B20 53 | .BYTE 'SBC' |
| 1250 | 1B21 45 | | 
| 1250 | 1B22 43 | | 
| 1250 | 1B23 53 | | 
| 1250 | 1B24 43 | | 
| 1300 | 1B25 53 | .BYTE 'SED' |
| 1300 | 1B26 45 | | 
| 1300 | 1B27 44 | | 
| 1310 | 1B28 53 | .BYTE 'SED' |
| 1310 | 1B29 45 | | 
| 1310 | 1B2A 49 | | 
| 1320 | 1B2B 53 | .BYTE 'SEI' |
| 1320 | 1B2C 54 | | 
| 1320 | 1B2D 41 | | 
| 1330 | 1B2E 53 | .BYTE 'STA' |
| 1330 | 1B2F 53 | | 
| 1330 | 1B30 53 | .BYTE 'STX' |
1330  IBES 54
1330  IBES 58
1340  IBF7 53
1340  IBF8 54
1340  IBF8 59
1350  IBF9 54
1360  IBF9 59
1370  IBF0 54
1370  IBF1 53
1370  IBF2 59
1380  IBF3 54
1380  IBF4 59
1390  IBF5 41
1390  IBF5 54
1390  IBF7 59
1390  IBF8 53
1400  IBF9 54
1400  IBFA 59
1410  IBFA 41
1410  IBFC 54
1410  IBFD 45
1410  IBFE 58
1420
1430  IBFF FF
1440
1440  .BYTE ETX
1450  ; SINCE THIS IS THE END OF A
1460  ; STRING OF CHARACTERS, USE
1470  ; ETX TO INDICATE END OF TEXT.
1480
1490
1500
1510
1520
1530
1540
1550
1560
1570
1580
1590
1600
1610
1620
1630
1640
1650
1660
1670
1680
1690
1700
1710
1720  1C00 22
1720  1C01 6A

MCD05 .BYTE $22,$6A,1,1,1,$6A,$0A,1,$70
1790 1C3C 01
1790 1C3D 07
1790 1C3E 79
1790 1C3F 01
16D8 1C40 7F
1800 1C41 49
1800 1C42 01
1800 1C43 01
1800 1C44 01
1800 1C45 49
1800 1C46 64
1800 1C47 01
1810 1C48 6D
1810 1C49 49
1810 1C4A 01
1810 1C4B 01
1810 1C4C 55
1810 1C4D 49
1810 1C4E 64
1810 1C4F 01
1820 1C50 25
1820 1C51 49
1820 1C52 01
1820 1C53 01
1820 1C54 01
1820 1C55 49
1820 1C56 64
1820 1C57 01
1830 1C58 31
1830 1C59 49
1830 1C5A 01
1830 1C5B 01
1830 1C5C 01
1830 1C5D 49
1830 1C5E 64
1830 1C5F 01
1840 1C60 02
1840 1C61 04
1840 1C62 01
1840 1C63 01
1840 1C64 01
1840 1C65 04
1840 1C66 7C
1840 1C67 01
1850 1C68 73
1850 1C69 04
1850 1C6A 7C
1850 1C6B 01
1850 1C6C 55
1850 1C6D 04
1850 1C6E 7C
1850 1C6F 01
1860 1C70 28
1860 1C71 04
1860 1C72 01
1860 1C73 01
1860 1C74 01
1860 1C75 04
1940 1CB0 10
1940 1CB1 5B
1940 1CB2 01
1940 1CB3 01
1940 1CB4 61
1940 1CB5 5B
1940 1CB6 5E
1940 1CB7 01
1950 1CBB 34
1950 1CBB 5B
1950 1CBA 9E
1950 1CBA 01
1950 1CBB 61
1950 1CBB 5B
1950 1CBB 5E
1950 1CBF 01
1560 1CC0 3D
1560 1CC1 37
1560 1CC2 01
1560 1CC3 01
1560 1CC4 3D
1560 1CC5 37
1560 1CC6 40
1560 1CC7 01
1570 1CC8 52
1570 1CC9 37
1570 1CCA 43
1570 1CBB 01
1570 1CBB 3D
1570 1CBB 37
1570 1CBB 40
1570 1CCF 01
1580 1CD0 1C
1580 1CD1 37
1580 1CD2 01
1580 1CD3 01
1580 1CD4 01
1580 1CD5 37
1580 1CD6 40
1580 1CDE 01
1590 1CDE 2E
1590 1CDB 37
1590 1CD8 01
1590 1CD9 01
1590 1CDB 01
1590 1CDC 01
1590 1CDD 37
1590 1CDE 40
1590 1CDF 01
2000 1CE0 3A
2000 1CE1 85
2000 1CE2 01
2000 1CE3 01
2000 1CE4 3A
2000 1CE5 85
2000 1CE6 4C
2000 1CE7 01
2010 1CE8 4F
2010 1CE9 85

.BYTE $10,$5B,1,1,$61,$5B,$5E,1

.BYTE $34,$5B,$3E,1,$61,$5B,$5E,1

.BYTE $3D,$37,1,1,$3D,$37,$40,1

.BYTE $52,$37,$43,1,$3D,$37,$40,1

.BYTE $1C,$37,1,1,$37,$40,1

.BYTE $2E,$37,1,1,$37,$40,1

.BYTE $3A,$85,1,1,$3A,$85,$4C,1

.BYTE $4F,$85,$87,1,$3A,$85,$4C,1
Table of Addressing Mode Codes

An addressing mode's code is its offset into SUBS, the table of addressing mode subroutines.

MODES \( \text{BYTE } 18, 22, 0, 0, 6, 6, 0 \)
2330 1007 00  .BYTE 18,4,2,0,0,12,12,0
2340 1008 12  .BYTE 20,24,0,0,0,14,14,0
2350 1009 04  .BYTE 18,16,0,0,0,22,22,0
2360 1009 10  .BYTE 12,22,0,0,6,6,6,0
2370 1009 20  .BYTE 18,4,2,0,12,12,12,0
2380 1009 30  .BYTE 20,24,0,0,0,8,8,0
2390 1009 40  .BYTE 18,16,0,0,0,14,14,0
2390 1009 50  .BYTE 18,22,0,0,6,6,6,0
2410 1D41 16
2410 1D42 00
2410 1D43 00
2410 1D44 00
2410 1D45 06
2410 1D46 06
2410 1D47 00
2420 1D48 12
2420 1D49 0C
2420 1D4A 02
2420 1D4B 00
2420 1D4C 0C
2420 1D4D 0C
2420 1D4E 0C
2420 1D4F 00
2430 1D50 14
2430 1D51 19
2430 1D52 00
2430 1D53 00
2430 1D54 00
2430 1D55 08
2430 1D56 08
2430 1D57 00
2440 1D58 12
2440 1D59 10
2440 1D5A 00
2440 1D5B 00
2440 1D5C 00
2440 1D5D 0C
2440 1D5E 0C
2440 1D5F 00
2450 1D60 12
2450 1D61 16
2450 1D62 00
2450 1D63 00
2450 1D64 00
2450 1D65 06
2450 1D66 06
2450 1D67 00
2460 1D68 12
2460 1D69 04
2460 1D6A 02
2460 1D6B 00
2460 1D6C 1A
2460 1D6D 0C
2460 1D6E 0C
2460 1D6F 00
2470 1D70 14
2470 1D71 18
2470 1D72 00
2470 1D73 00
2470 1D74 00
2470 1D75 00
2470 1D76 00
2470 1D77 00
2480 1D78 12
2480 1D79 10
2480 1D7A 00

.BYTE 16, 12, 2, 0, 12, 12, 12, 0

.BYTE 20, 24, 0, 0, 0, 8, 8, 0

.BYTE 16, 16, 0, 0, 0, 14, 14, 0

.BYTE 18, 16, 0, 0, 0, 14, 14, 0

.BYTE 18, 22, 0, 0, 0, 6, 6, 0

.BYTE 18, 22, 0, 0, 0, 6, 6, 0

.BYTE 18, 4, 2, 0, 26, 12, 12, 0

.BYTE 20, 24, 0, 0, 0, 8, 8, 0

.BYTE 18, 16, 0, 0, 0, 14, 14, 28

312 BEYOND GAMES
2480 1D78 00
2480 1D7C 00
2480 1D7D 0C
2480 1D7E 0C
2480 1D7F 1C
2490
2500 1D80 00 .BYTE 0,22,0,0,6,6,6,0
2500 1D81 16
2500 1D82 00
2500 1D83 00
2500 1D84 06
2500 1D85 06
2500 1D86 06
2500 1D87 00
2510 1D88 12 .BYTE 18,0,18,0,12,12,12,0
2510 1D89 00
2510 1D8A 12
2510 1D8B 00
2510 1D8C 0C
2510 1D8D 0C
2510 1D8E 0C
2510 1D8F 00
2520 1D90 14 .BYTE 20,24,0,0,8,8,10,0
2520 1D91 18
2520 1D92 00
2520 1D93 00
2520 1D94 08
2520 1D95 08
2520 1D96 0A
2520 1D97 00
2530 1D98 12 .BYTE 18,16,18,0,9,14,0,0
2530 1D99 10
2530 1D9A 12
2530 1D9B 00
2530 1D9C 00
2530 1D9D 0E
2530 1D9E 00
2530 1D9F 00
2540 1DA0 04 .BYTE 4,22,4,0,6,6,6,0
2540 1DA1 16
2540 1DA2 04
2540 1DA3 00
2540 1DA4 06
2540 1DA5 06
2540 1DA6 06
2540 1DA7 00
2550 1DA8 12 .BYTE 18,4,18,0,12,12,12,0
2550 1DA9 04
2550 1DAA 12
2550 1DAB 00
2550 1DAC 0C
2550 1DAD 0C
2550 1DAE 0C
2550 1DAF 00
2560 1DB0 14 .BYTE 20,24,0,0,8,8,10,0
2560 1DB1 18
2560 1DB2 00
2560 1DB3 00
2630 1DEF 0C
2630 1DEF 00
2640 1DF0 14
2640 1DF1 18
2640 1DF2 00
2640 1DF3 00
2640 1DF4 00
2640 1DF5 00
2640 1DF6 00
2640 1DF7 00
2650 1DF8 12
2650 1DF9 10
2650 1DFA 00
2650 1DFB 00
2650 1DFC 00
2650 1DFD 0E
2650 1DFE 0E
2650 1dff 00

 BYTE 20,24,0,0,0,0,0,0

 BYTE 18,16,0,0,0,14,14,0
Appendix C9:

Move Utilities
APPENDIX C9: ASSEMBLER LISTING OF
MOVE UTILITIES

SEE CHAPTER 10 OF BEYOND GAMES: SYSTEMS
SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER.

BY KEN SKIER

10 ;
20 ;
30 ;
40 ;
50 ;
60 ;
70 ;
80 ;
90 ;
100 ;
110 ;
120 ;
130 ;
140 ;
150 ;
160 ;
170 ;
180 ;
190 ;
200 ;
210 ;
220 ;
230 ;
240 ;
250 ;

;**********************************************************************

160 ;
170 ;
180 ;
190 ;
200 ;
210 ;
220 ;
230 ;
240 ;
250 ;

260 6000= CR=$0D CARRIAGE RETURN.
270 008A= LF=$0A LINE FEED.
280 007F= T=$7F START OF TEXT CHARACTER.
290 00FF= ETX=$FF END OF TEXT CHARACTER.

;**********************************************************************

300 ;
310 ;
320 ;
330 ;
340 ;
350 ;
360 ;
370 ;
380 ;
390 ;
400 ;
410 ;
420 ;
430 ;
440 ;
450 ;
460 ;
470 ;
480 ;

490 1200= VMPAGE=$1200 STARTING PAGE OF VISIBLE MONITOR CODE.
500 ;
510 ;
520 1205= SELECT=VMPAGE+5
530 1207= USIM=VMPAGE+7
540 ;
550 ;
560 ;
570 1400= PRPAG=$1400 STARTING PAGE OF PRINT CODE.
580 ;
590 1499=  TUT.0N=PRPAGE+8
600 14E4=  PRINT:=PRPAGE+$E4
610 1512=  PUSHSL=PRPAGE+$112
620 152D=  POP. SL=PRPAGE+$12D
630  
640  
650 1500=  HEX.PG=$1500  ADDRESS OF PAGE IN WHICH
660  
670  
680  
690  
700  
710 15E9=  SETADS=HEX.PG+$E9
720  
730  
740  
750  
760  
770  
780  
790  
800  
810  
820  
830  
840  
850  
860  
870  
880  
890  
900 1780  *=+$1780
910  
920  
930 1552=  SA=HEX.PG+$52  POINTER TO START ADDRESS
940  
950  
960 1554=  EA=SA+2  POINTER TO END OF BLOCK TO
970  
1000  
1010 17B0 0000  NUM .WORD 0  NUMBER OF BYTES IN BLOCK
1020  
1030  
1040  
1050  
1060 17B2 0000  DEST .WORD 0  POINTER TO BLOCK’S
1070  
1080  
1090  
1100  
1110  
1120  
1130  
1140  
1150 0000=  GETPTR=0  THESE TWO "PAGE POINTERS"
1160 0002=  PUTPTR=GETPTR+2 GET AND PUT BYTES.
1170  
1180  

320 BEYOND GAMES
1130  ;
1200  ;
1210  ;
1220  ;
1230  ;
1240  ;
1250  ;
1260  ;
1270  ;
1280  ;
1290  ;
1300  ;
1310  ;
1320  ;
1330  ;
1340  ;
1350  ;
1360  ;
1370  17B4 200814 MOVER  JSR TUT.ON  SELECT SCREEN FOR OUTPUT.
1380  17B7 20E414 JSR PRINT:  DISPLAY A TITLE.
1390  17BA 7F  .BYTE TEX,CR,LF
1390  17BE 0D
1390  17BC 0A
1400  17BD 20
1400  17BE 20
1400  17DF 20
1400  17C0 20
1400  17C1 20
1400  17C2 4D
1400  17C3 4F
1400  17C4 56
1400  17C5 45
1400  17C6 20
1400  17C7 54
1400  17C8 4F
1400  17C9 4F
1400  17CA 4C
1400  17CB 2E
1410  17CC 0D  .BYTE CR,LF,LF,ETX
1410  17CD 0A
1410  17CE 0A
1410  17CF FF
1420  ;
1430  17D0 20E915 JSR SETADS GET START ADDRESS, END
1440  ; ADDRESS FROM USER.
1450  ;
1460  17D3 20B918 JSR SET.DA GET DESTINATION ADDRESS
1470  ; FROM USER.
1480  ; WITH THOSE POINTERS SET,
1490  ; WE'RE READY TO EXECUTE MOV.EA:
1560 ; MOV.EA: MOVE BLOCK SPECIFIED BY SA, EA, DEST
1565 ;
1570 ; RETURNS CODES:
1575 ;
1580 ;
1585 ;
1590 ;
1600 ;
1605 ;
1610 ;
1615 ;
1620 ;
1625 ;
1630 ;
1635 ;
1640 ;
1645 ;
1650 ;
1655 ;
1660 ;
1665 ;
1670 ;
1675 ;
1680 ;
1685 ;
1690 ;
1695 ;
1700 0000= ERROR=0 THIS RETURN CODE MEANS
1705 ; SA < EA, SO MOVE ABORTED.
1710 ;
1715 00FF= OKAY=$FF THIS RETURN CODE MEANS
1720 ; MOVE ACCOMPLISHED.
1725 ;
1730 ;
1735 ;
1740 ;
1745 ;
1750 ;
1755 ;
1760 ;
1765 ;
1770 17D6 AE515 MOV.EA LDX EA+1 SET NUM = EA - SA:
1775 1796 6F 3B SEC
1781 179C 0515 LDA EA
1787 17DD ED5215 SBC SA
1793 17E0 8D8017 STA NUM
1799 17E3 B002 BCS MOVE.1
1805 17E5 CA DEX
1811 17E6 39 SEC
1817 17E7 6A MOVE.1 TXA
1823 17E8 ED315 SBC SA+1
1829 17EA 8D8117 STA NUM+1
1835 17EE B003 BCS MOVNUM
1841 1800 17F0 9900 ER.RTN LDA #ERROR IF EA < SA,
1847 1810 17F2 60 RTS RETURN WITH ERROR CODE.
1852 1820 ;
1857 1830 ;
1862 1840 ;
1867 1850 ;
1868 ;
1869 ;
1870 ;
1871 ;
1872 ;
1873 ;
1874 ;
1875 ;
1876 ;
1877 ;
1878 ;
1879 ;
1880 ;
1881 ;
1882 ;
1883 ;
1884 ;
1885 ;
1886 ;
1887 ;
1888 ;
1889 ;
1890 ;
1891 ;
1892 ;
1893 ;
1894 ;
1895 ;
1896 ;
1897 ;
1898 ;
1899 ;
1900 ;
1901 ;
1902 ;
1903 ;
1904 ;
1905 ;
1906 ;
1907 ;
1908 ;
1909 ;
1910 ;
1911 ;
1912 ;
1913 ;
1914 ;
1915 ;
1916 ;
1917 ;
1918 ;
1919 ;
1920 ;
1921 ;
1922 ;
1923 ;
1924 ;
1925 ;
1926 ;
1927 ;
1928 ;
1929 ;
1930 ;
1931 ;
1932 ;
1933 ;
1934 ;
1935 ;
1936 ;
1937 ;
1938 ;
1939 ;
1940 ;
1941 ;
1942 ;
1943 ;
1944 ;
1945 ;
1946 ;
1947 ;
1948 ;
1949 ;
1950 ;
1951 ;
1952 ;
1953 ;
1954 ;
1955 ;
1956 ;
1957 ;
1958 ;
1959 ;
1960 ;
1961 ;
1962 ;
1963 ;
1964 ;
1965 ;
1966 ;
1967 ;
1968 ;
1969 ;
1970 ;
1971 ;
1972 ;
1973 ;
1974 ;
1975 ;
1976 ;
1977 ;
1978 ;
1979 ;
1980 ;
1981 ;
1982 ;
1983 ;
1984 ;
1985 ;
1986 ;
1987 ;
1988 ;
1989 ;
1990 ;
1991 ;
1992 ;
1993 ;
1994 ;
1995 ;
1996 ;
1997 ;
1998 ;
1999 ;
2000 ;
2001 ;
2002 ;
2003 ;
2004 17F3 A003 MOVNUM LDY #3 SAVE ZERO PAGE BYTES THAT
2009 17F5 B90000 LOOP.1 LDA GETPTR,Y WILL BE CHANGED.
2014 17F8 48 PHA
2019 17F9 00 DEX
2024 17FA 10F9 BPL LOOP.1
2029 ;
2034 ;
2039 ;
2044 17FC 3B SEC IF DEST>SA, BRANCH TO MOVE-UP
2049 17FD AD515 LDA SA+1
2054 1800 CDB317 CMP DEST+1
2059 1805 9040 BCC MOVEUP
2064 1805 D018 BNE MOVEDN
2069 ;
2074 ;
2079 ;
2084 ;
2089 ;
2094 ;
2099 ;
2104 ;
2109 ;
2114 ;
2119 ;
2124 ;
2129 ;
2134 ;
2139 ;
2144 ;
2149 ;
2154 ;
2159 ;
2164 ;
2169 ;
2174 ;
2179 ;
2184 ;
2189 ;
2194 ;
2199 ;
21999 BEYOND GAMES
LDA SA
CMP DEST
BCC MOVEUP
BNE MOVEIN

RETURN BEARING "OKAY" CODE.

RESTORE ZERO PAGE BYTES THAT WERE CHANGED.

RETURN W/ "OKAY" CODE.

SET PAGE POINTERS TO LOWEST PAGES IN ORIGIN, DESTINATION BLOCKS.

INITIALIZE PAGE INDEX TO BOTTOM OF PAGE.

USE X TO COUNT THE NUMBER OF PAGES TO MOVE. MORE THAN ONE PAGE TO MOVE?

IF NOT, MOVE LESS THAN A PAGE.

IF SO,

MOVE A PAGE DOWN,

STARTING AT THE BOTTOM.

INCREMENT PAGE INDEX.

IF PAGE NOT MOVED, MOVE NEXT BYTE...

INCREMENT PAGE POINTERS.

DECREMENT PAGE COUNT.

IF A PAGE LEFT TO MOVE,

MOVE IT AS A PAGE.

MOVE LESS THAN A PAGE

DOWN, STARTING AT THE BOTTOM.

MOVED LAST BYTE?

IF NOT, MOVE NEXT BYTE...

"OKAY" CODE.

MORE THAN A PAGE TO MOVE?

IF NOT, MOVE LESS THAN A PAGE.
TO MOVE MORE THAN A PAGE, SET PAGE POINTERS TO HIGHEST PAGES IN ORIGIN, DESTINATION BLOCKS.

TO DO THIS, FIRST SET (X,Y) = NUM - $FF, (RELATIVE ADDRESS OF HIGHEST PAGE IN A BLOCK.)

NOW (X,Y) = NUM - $FF. X IS LOW BYTE, Y IS HIGH BYTE

(LAST PAGE IN SOURCE BLOCK.)

NOW PUTPTR=DEST+NUM-$FF. (LAST PAGE IN DEST BLOCK.)
LDX NUM+1    LOAD X WITH NUMBER OF
PAGES TO MOVE.

FAGEUP LDY #$FF    SET PAGE INDEX TO TOP OF
PAGE.

LOOP.3 LDA (GETPTR),Y MOVE A PAGE UP, STARTING
STA (PUTPTR),Y AT THE TOP OF THE BLOCK.
DEY    DECREMENT PAGE INDEX.
ABOUT TO MOVE LAST BYTE
IN PAGE?
BNE LOOP.3 IF NOT, HANDLE NEXT BYTE.
AS BEFORE.

LDA (GETPTR),Y IF SO, MOVE THIS BYTE FROM
STA (PUTPTR),Y SOURCE TO DESTINATION.
DEC GETPTR+1    DECREMENT PAGE POINTERS.
DEC PUTPTR+1    DECREMENT PAGE COUNTER.
BNE PAGEUP IF A PAGE LEFT TO MOVE,
MOVE IT AS A PAGE....

LESSUP JSR LOPAGE MOVE LESS THAN A PAGE UP,
LDY NUM STARTING AT THE TOP.

MOVE.6 LDA (GETPTR),Y COPY A BYTE FROM ORIGIN
STA (PUTPTR),Y TO DESTINATION.
DEY    DECREMENT PAGE INDEX.
CPY #$FF    COPIED THE LAST BYTE?
BNE MOVE.6 IF NOT, HANDLE AS BEFORE...
JMP OK.RTN IF SO, RETURN BEARING
"OKAY" CODE.

RESET POINTERS TO BOTTOM OF
ORIGIN, DESTINATION BLOCKS.
328 BEYOND GAMES
4320 18D0 4E
4320 18D1 20
4320 18D2 41
4320 18D3 4E
4320 18D4 44
4320 18D5 20
4320 18D6 50
4320 18D7 52
4320 18D8 45
4320 18D9 53
4320 18DA 53
4320 18DB 20
4320 18DC 51
4320 18DD 2E
4330 18DE FF .BYTE ETX
4340 18DF 200712 JSR VISMON LET USER SET AN ADDRESS.
4350 18E2 000512 DAHERE LDA SELECT SET DEST=SELECT.
4360 18E5 00B217 STA DEST
4370 18E8 000512 LDA SELECT+1
4380 18E8 00B317 STA DEST+1
4390 ;
4400 18EE 60 RTS RETURN WITH DEST=SELECT.
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 BEYOND GAMES: SYSTEMS
SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

BY KEN SKIER

; ************************************************************************
; 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=0**  
**PARAMS=$1000**  
**TVCOLS=PARAMS+3**  
**TVROWS=PARAMS+4**  
**ARR=PARAMS+7**  
**TVSUBS=$1100**  
**CLR.XY=TVSUBS+$13**  
**TVHOME=TVSUBS+$2B**  
**TUOXY=TVSUBS+$3C**  
**TUDOWN=TVSUBS+$76**  
**TVSKIP=TVSUBS+$7F**  
**TVPLUS=TVSUBS+$81**  
**TV.PUT=TVSUBS+$9B**  
**VBYTE=TVSUBS+$A3**  
**TVUSH=TURBS+$C4**  
**TV.POP=TVSUBS+$D3**  
**UMPAGE=$1200**  
**SELECT=UMPAGE+5**  
**GET.SL=UMPAGE+$94**  
**INC.SL=UMPAGE+$10D**  
**DEC.SL=UMPAGE+$11A**  
**PRPAGE=$1400**  
**TUT.ON=PRPAGE+$8**  
**TUT.OFF=PRPAGE+$CE**  
**PR.ON=PRPAGE+$14**  
**PR.OFF=PRPAGE+$1A**  
**PR.CH=true=PRPAGE+$40**  
**PRINT=true=PRPAGE+$E4**  
**PUSHSL=PRPAGE+$112**  
**POP.SL=PRPAGE+$12B**  
**HEX.PG=$1500**  
**SA=HEX.PG+$52**  
**EA=SA+2**  
**SETADS=HEX.PG+$E9**  
**NEXTSL=HEX.PG+$293**  
**GOTOSL=HEX.PG+$2A0**
EDPAGE=$1E00  STARTING PAGE OF EDITOR.
EDIT=EDPAGE+$C8

; *****************************************************************************

; VARIABLES

; *****************************************************************************

; **EDPAGE

; COUNTR .BYTE 0  COUNTER USED BY LINE.2.
; EIMODE .BYTE 0  FLAG: 0=OVERSTRIKE, 1=INSERT.

; *****************************************************************************

; TEXT EDITOR: TOP LEVEL

; *****************************************************************************

; EDITOR JSR SETBUF INITIALIZE BUFFER POINTERS.
; JSR SHOWIT SHOW USER A PORTION OF
; JSR EDITIT LET THE USER EDIT THE BUFFER
; OR MOVE ABOUT WITHIN IT.
; CLC LOOP BACK TO SHOW THE
; BCC EDLOOP CURRENT TEXT.

; *****************************************************************************

; INITIALIZE BUFFER POINTERS
1750 ;
1760 ; ****************************************************
1770 ;
1780 ;
1790 ;
1800 ;
1810 1E0F 200814 SETBUF JSR TUT.ON SELECT SCREEN.
1820 1E12 20E414 JSR PRINT: DISPLAY "SET UP EDIT BUFFER."
1830 1E15 7F .BYTE TEX,CR,LF,LF
1830 1E16 0D
1830 1E17 0A
1830 1E18 0A
1840 1E19 53 .BYTE 'SET UP EDIT BUFFER.
1840 1E1A 45
1840 1E1B 54
1840 1E1C 20
1840 1E1D 55
1840 1E1E 50
1840 1E1F 20
1840 1E20 45
1840 1E21 44
1840 1E22 49
1840 1E23 54
1840 1E24 20
1840 1E25 42
1840 1E26 55
1840 1E27 46
1840 1E28 46
1840 1E29 45
1840 1E2A 52
1840 1E2B 2E
1850 1E2C 0D .BYTE CR,LF,LF,ETX
1850 1E2D 0A
1850 1E2E 0A
1850 1E2F FF
1860 1E30 20E915 ; JSR SETADS LET USER SET LOCATION AND
1870 ; SIZE OF EDIT BUFFER.
1880 1E33 20A817 ; JSR GOTOSA SET SELECT-START OF BUFFER.
1890 1E36 60 ; RTS RETURN TO CALLER.
1900 ;
1910 ;
1920 ;
1930 ;
1940 ;
1950 ;
1960 ;
1970 ; ****************************************************
1980 ;
1990 ; DISPLAY A PORTION OF EDIT BUFFER
2000 ; ****************************************************
2020 ;
2030 ;
2040 ;
2050 ;
2060 ;
2070 ;
2080 1E37 20C411 SHOWIT JSR TUPUSH SAVE THE ZERO PAGE BYTES
2090 ; WE'LL USE.
2100 1E3A 202B11 JSR TVHOME SET HOME POSITION OF EDIT DISPLAY.
2110 ;
2120 ;
2130 ;
2140 1E3D AE0310 LDX TVCOLS CLEAR THREE ROWS FOR THE EDIT DISPLAY.
2150 1E40 A003 LDY #3
2160 1E42 201311 JSR CLR.XY
2170 ;
2180 ;
2190 1E45 202B11 JSR TVHOME RESTORE TV.PTR TO HOME POSITION OF EDIT DISPLAY.
2200 ;
2210 1E48 207611 JSR TVDOWN SET TV.PTR TO BEGINNING OF LINE TWO AND SAVE IT.
2220 1E48 20C411 JSR TVPUSH DISPLAY TEXT IN LINE TWO.
2230 1E4E 205E1E JSR LINE.2
2240 ;
2250 2260 1E51 20D311 JSR TV.POP SET TV.PTR TO BEGINNING OF OF THIRD LINE OF EDIT DISPLAY.
2270 1E54 207611 JSR TVDOWN
2280 ;
2290 1E57 20891E JSR LINE.3 DISPLAY THIRD LINE OF EDIT DISPLAY.
2300 ;
2310 ;
2320 1E5A 20D311 JSR TV.POP RESTORE ZERO PAGE BYTES USED.
2330 1E5D 60 RTS RETURN TO CALLER, WITH EDIT DISPLAY ON SCREEN, REST OF SCREEN UNCHANGED, AND ZERO PAGE PRESERVED.
2340 ;
2350 2440 ;
2360 ;
2370 ;
2380 ;
2390 ;
2400 ;
2410 ;
2420 ;
2430 ;
2440 ;
2450 ;
2460 ;
2470 ;
2480 ;
2490 ;
2500 ;
2510 ;
2520 ;
2530 1E5E 201215 LINE.2 JSR PUSHL SAVE SELECT POINTER.
2540 1E61 AD0310 LDA TVCOLS SET X EQUAL TO HALF THE WIDTH
2550 1E64 4A LSR A OF THE SCREEN.
2560 1E65 AA TAX
2570 1E66 CA DEX
2580 1E67 CA DEX
2590 ;
2600 1E68 201A13 LOOP.1 JSR DEC.SL DECREMENT SELECT...
2610 1E6B CA DEX ...
2620 1E6C 10FA BPL LOOP.1 X TIMES.
2630 ;
2640 1E6E AD0310 LDA TVCOLS INITIALIZE COUNTR.
2650 1E71 8D081E STA COUNTR (WE'LL DISPLAY TVCOLS
2660 ; CHARACTER.)
2670 1E74 209412 LOOP.2 JSR GET.SL GET A CHARACTER FROM BUFFER.
2680 1E77 209B11  JSR TV.PUT  PUT IT ON SCREEN.
2690 1E7A 207F11  JSR TUSKIP  GO TO NEXT SCREEN POSITION.
2700 1E7D 200D13  JSR INC.SL  ADVANCE TO NEXT BYTE IN
2710 ;  BUFFER.
2720 1E80 CE011E  DEC COUNTR  DONE LAST CHARACTER IN ROW?
2730 1E83 10EF  BPL LOOP.2  IF NOT, DO NEXT CHARACTER.
2740 ;
2750 ;
2760 1E85 202B15  JSR POP.SL  RESTORE SELECT FROM STACK.
2770 1E88 60  RTS  RETURN TO CALLER.
2780 ;
2790 ;
2800 ;
2810 ;
2820 ;
2830 ;
2840 ;
2850 ;
2860 ;
2870 ;
2880 ;
2890 ;
2900 ;
2910 ;
2920 ;
2930 ;
2940 ;
2950 ;
2960 ;
2970 ;
2980 ;
2990 ;
3000 1E92 AD0310  LINE.3 LDA TUCOLS  SELECT CENTER POSITION...
3040 1E8C 4A  LSR A  A=TUCOLS/2
3050 1E8D E902  SBC $2  A=(TUCOLS/2)-2
3060 1E8F 200111  JSR TVPLUS  NOW TV.PTR IS POINTING TWO
3070 ;  CHARACTERS TO THE LEFT OF
3080 ;  CENTER OF LINE 3 OF THE
3090 ;  EDIT DISPLAY.
3100 1E92 AD011E  LDA EDMODE  WHAT IS CURRENT MODE?
3110 1E95 C901  CMP $1  IS IT INSERT MODE?
3120 1E97 D005  BNE OVMODE  IF NOT, IT MUST BE OVERSTRIKE
3130 ;  MODE.
3140 1E99 A949  LDA $INSCHR  IF SO, GET INSERT GRAPHIC.
3150 1E9B 18  CLC
3160 1E9C S002  BCC TUMODE
3170 1E9E A94F  OVMODE LDA $OURCHR  LOAD A W/OVERSTRIKE CHARACTER.
3180 3EA0 209B11  TVMODE JSR TV.PUT  PUT MODE GRAPHIC ON SCREEN.
3190 3EA3 A002  LDA $2  MOVE TWO POSITIONS TO THE
31A0 3EAE 208111  JSR TVPLUS  RIGHT, SO TV.PTR POINTS TO
31B0 ;  CENTER OF LINE 3 OF EDIT
31C0 ;  DISPLAY.
31D0 3E98 AD0710  LDA ARROW  DISPLAY AN UP-ARROW HERE.
31E0 3EAB 205B11  JSR TV.PUT  RETURN TO CALLER.
31F0 ;
3200 1EB3 AD0612  LDA SELECT+1
3210 1EB6 20A111  JSR VUSBYTE
3220 1EB9 AD0512  LDA SELECT
3230 1EBC 208311  JSR VUSBYTE
3240 ;
3250 1EBF 60  RTS  RETURN TO CALLER.
Appendix C11:

Simple Text Editor (EDITIT Subroutine)
APPENDIX C11: ASSEMBLER LISTING OF
A SIMPLE TEXT EDITOR
EDITIT SUBROUTINE

SEE CHAPTER 11 OF BEYOND GAMES: SYSTEMS
SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

BY KEN SKIER

***************
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
***************
500 ; ****************************************
550 ;
600 ;
610 ;
620 ;
630 ;
640 1200=
650 ;
660 1205=
670 1207=
680 1234=
690 12E0=
700 130D=
710 131A=
720 132D=
730 ;
740 ;
750 1400=
760 ;
770 1414=
780 141A=
790 1440=
7A0 14E4=
7B0 1512=
7C0 152B=
7D0 ;
7E0 ;
7F0 ;
800 1552=
810 ;
820 ;
830 1500=
840 ;
850 ;
860 1554=
870 1657=
880 1763=
890 17A0=
8A0 ;
8B0 ;
8C0 1780=
8D0 1792=
8E0 17D6=
8F0 18E2=
900 ;
910 ;
920 ;
930 17B0=
940 17B2=
950 17B6=
960 18E2=
970 ;
980 1E00=
990 1E08=
9A0 ;
9B0 ;
9C0 ;
9D0 ;
9E0 ;
9F0 ;
1000 ;
1010 ;
1020 ;
1030 ;
1040 ;
1050 ;
1060 ;
1070 ;
1080 ;
1090 ;
10A0 ;
10B0 ;
10C0 ;
10D0 ;
10E0 ;
10F0 ;
1100 ;
1110 ;
1120 ;
1130 ;

VMPAGE=$1200 STARTING PAGE OF VISIBLE MONITOR CODE.
SELECT=VMPAGE+5
VISMON=VMPAGE+7
GET.SL=VMPAGE+$94
GETKEY=VMPAGE+$E0
INC.SL=VMPAGE+$10D
DEC.SL=VMPAGE+$11A
PUT.SL=VMPAGE+$12D

PRPAGE=$1400 STARTING PAGE OF PRINT UTILITIES.
PR.ON =PRPAGE+$14
PR.OFF=PRPAGE+$1A
PR.CHR=PRPAGE+$40
PRINT:=PRPAGE+$E4
PUSHSL=PRPAGE+$112
POPSL=PRPAGE+$12B

HEX.PG=$1500 ADDRESS OF PAGE IN WHICH HEXDUMP CODE STARTS.
SFi=HEX.PG+$52
EA=SA+2
SAHERE=HEX.PG+$167
NEXTSL=HEX.PG+$293
GotoSa=HEX.PG+$2A0

MOVERS=$17D0 START OF MOVE OBJECT CODE.
DEST=MOVERS+2
MOV.EA=MOVERS+$26
DAHERE=MOVERS+$132

EDPAGE=$1E00 STARTING PAGE OF EDITOR.
EDKEYS=EDPAGE+$C0

340 BEYOND GAMES
1140 1E00 *=EDPAGE
1150 ;
1160 ;
1170 ;
1180 1E01= EDMODE=EDPAGE+1 0=OVERSTRIKE MODE.
1190 ; 1=INSERT.
1200 ;
1210 1EC0 *=EDKEYS
1220 ;
1230 ; EDIT FUNCTION KEYS
1240 ;
1250 ;
1260 ;
1270 ;
1280 ;
1290 ;
1300 ;
1310 ;
1320 ;
1330 ;
1340 1EC0 05 FLUSHK .BYTE $05
1350 ; THIS KEY FLUSHES THE
1360 ; BUFFER OF ANY TEXT. $05 IS
1370 ; CONTROL-F. THUS, CONTROL-F
1380 ; TO FLUSH THE BUFFER.
1390 ;
1400 1EC1 03 MODEKY .BYTE $03
1410 ; THIS KEY CAUSES THE EDIT
1420 ; TO CHANGE MODES, FROM INSERT
1430 ; TO OVERSTRIKE, AND VICE VERSA.
1440 ; $03 IS CONTROL-C. THUS,
1450 ; CONTROL-C TO CHANGE MODES.
1460 1EC2 3E NEXTKY .BYTE 'Y'
1470 ; THIS KEY SELECTS THE NEXT
1480 ; CHARACTER IN THE BUFFER.
1490 ;
1500 ;
1510 ;
1520 1EC3 3C PREVKY .BYTE '<'
1530 ; SELECT PREVIOUS CHARACTER
1540 ; IN THE BUFFER. SUBSTITUTE
1550 ; LEFT-ARROW IF YOUR KEYBOARD
1560 ; HAS IT.
1570 1EC4 10 PRTKEY .BYTE $10
1580 ; THIS KEY PRINTS THE BUFFER.
1590 ; CONTROL-P
1600 ; TO PRINT THE BUFFER.
1610 1EC5 7F RUBKEY .BYTE $7F
1620 ; THIS KEY RUBS OUT THE
1630 ; CURRENT CHARACTER. IF YOU
1640 ; HAVE DELETE KEY BUT NOT RUBOUT,
1650 ; USE YOUR SYSTEM'S CODE FOR
1660 ; THE DELETE KEY.
1670 ;
1680 1EC6 51 QUITKY .BYTE 'Q'
1690 ; TWO QUIT KEYS IN A ROW
1700 ; CAUSE THE EDITOR TO RETURN
1710 ; TO ITS CALLER.
1720 ;
1730 ;
1740 ;
1750 ;
1760 ;
1770 ;
1780 1EC7 00  TEMPCH .BYTE 0 THIS BYTE USED BY EDITIT.
1790 ;
1800 ;
1810 ;
1820 ;
1830 ;
1840 ;
1850 ;
1860 ;
1870 ;
1880 ;
1890 ;
1900 ;
1910 ;
1920 ;
1930 ;
1940 ;
1950 ;
1960 ;
1970 ;
1980 ;
1990 ;
2000 ;
2010 1EC8 20E012 EDITIT JSR GETKEY GET A KEYSTROKE FROM USER.
2020 ;
2030 1ECB CDC61E CMP QUITKY IS IT THE "QUIT" KEY?
2040 1ECE D017 BNE DO.KEY IF NOT, DO WHAT THE KEY
2050 ;
2060 ;
2070 1ED0 48 PHA IF IT IS THE "QUIT" KEY, SAVE
2080 1ED1 20E012 JSR GETKEY IT AND GET A NEW KEY FROM
2090 ;
2100 1ED4 CDC61E CMP QUITKY IS THIS A "QUIT" KEY, TOO?
2110 1ED7 D004 BNE NOTEND IF NOT, THEN THIS IS NOT THE
2120 ;
2130 ;
2140 ;
2150 1ED9 6B ENDED? PLA END THE EDIT SESSION?
2160 ;
2170 1EDA 6B PLA POP FIRST "QUIT" KEY FROM
2180 1EDB 68 PLA POP RETURN ADDRESS TO
2190 1EDC 60 RTS EDITOR'S TOP LEVEL.
2200 ;
2210 1EED 8DC71E NOTEND STA TEMPCH RETURN TO EDITOR'S CALLER.
2220 ;
2230 1EE0 6B PLA SAVE TH KEY THAT FOLLOWED
2240 1EE1 20E71E JSR DO.KEY THE "QUIT" KEY.
2250 1EE4 ADC71E LDA TEMPCH POP FIRST "QUIT" KEY FROM STACK.
2260 ;
2270 ;
2280 ;
2290 ;

342 BEYOND GAMES
; IS IT THE "CHANGE MODE" KEY?
2320 1EEA D08B  BNE IFNEXT    IF NOT, PERFORM NEXT TEST.
2330 1EEC CE011E  DEC EMDMDE    IF SO, CHANGE THE EDITOR'S
2340 1EEF 1005  BPL DO.END    MODE.
2350 1EF1 A9C1  LDA #1
2360 1EF3 ED011E  STA EMDMDE    RETURN TO CALLER.
2370 1EF6 60  DO.END  RTS
2380  ;
2390  ;
2400 1EF7 CDC21E IFNEXT  CMP NEXTKY    IS IT THE "NEXT" KEY?
2410 1EEA D004  BNE IFPREV    IF NOT, PERFORM NEXT TEST.
2420  ;
2430 1EFC 20781F  JSR NEXTCH    IF SO, ADVANCE TO NEXT
2440  ;
2450 1EFF 60  RTS    CHARACTER...
2460  ;
2470  ;
2480 1F00 CDC31E IFPREV  CMP PREVKY    ...AND RETURN.
2490 1F03 D004  BNE IF.RUB    IS IT THE "PREVIOUS" KEY?
2500 1F05 20871F  JSR PREVL    IF NOT, PERFORM NEXT TEST.
2510 1F08 60  RTS    IF SO, BACK UP TO PREVIOUS
2520  ;
2530  ;
2540 1F09 CDC51E IF.RUB  CMP RUBKEY    CHARACTER AND RETURN.
2550 1F0C D004  BNE IF.RT    IS IT THE "RUBOUT" KEY?
2560 1F0E 20001F  JSR DELETE    IF NOT, PERFORM NEXT TEST.
2570 1F11 60  RTS    IF SO, DELETE CURRENT
2580  ;
2590  ;
2600 1F12 CDC41E IF.RT  CMP PRTKEY    CHARACTER AND RETURN.
2610 1F15 D004  BNE IFFLSH    IS IT THE "PRINT" KEY?
2620 1F17 20C51F  JSR PRTBUF    IF NOT, PERFORM NEXT TEST.
2630 1F1A 60  RTS    IF SO, PRINT THE BUFFER...
2640  ;
2650  ;
2660  ;
2670 1F1B CDC01E IFFLSH  CMP FLSHKY    ...AND RETURN.
2680 1F1E D004  BNE CHARKY    IS IT THE "FLUSH" KEY?
2690  ;
2700 1F20 20B41F  JSR FLUSH    IF NOT, IT MUST BE A CHARACTER
2710 1F23 60  RTS    KEY.
2720  ;
2730  ;
2740  ;
2750  ;
2760  ;
2770  ;
2780  ;
2790  ;
2800  ;
2810 1F24 AE011E CHARKY  LDX EMDMDE    ARE WE IN OVERSTRIKE MODE?
2820 1F27 F004  BEQ STRIKE    IF SO, OVERSTRIKE THE CURRENT
2830  ;
2840 1F29 20341F  JSR INSERT    CHARACTER.
2850 1F2C 60  RTS    IF NOT, INSERT THE CHARACTER.
2860  ;
2870 1F2D 202D13 STRIKE JSR PUT SL    RETURN.
2880  ;
2890  ;
2900  ;
2910  ;
2920  ;
2930  ;
2940  ;
2950  ;
2960  ;
2970  ;
2980  ;
2990  ;
3000  ;
3010  ;
3020  ;
3030  ;
3040  ;
3050  ;
3060  ;
3070  ;
3080  ;
3090  ;
3100  ;
3110  ;
3120  ;
3130  ;
3140  ;
3150  ;
3160  ;
3170  ;
3180  ;
3190  ;
3200  ;
3210  ;
3220  ;
3230  ;
3240  ;
3250  ;
3260  ;
3270  ;
3280  ;
3290  ;
3300  ;
3310  ;
3320  ;
3330  ;
3340  ;
3350  ;
3360  ;
3370  ;
3380  ;
3390  ;
3400  ;
3410  ;
3420  ;
3430  ;
3440  ;
3450  ;
3460  ;
3470  ;
3480  ;
3490  ;
3500  ;
3510  ;
3520  ;
3530  ;
3540  ;
3550  ;
3560  ;
3570  ;
3580  ;
3590  ;
3600  ;
3610  ;
3620  ;
3630  ;
3640  ;
3650  ;
3660  ;
3670  ;
3680  ;
3690  ;
3700  ;
3710  ;
3720  ;
3730  ;
3740  ;
3750  ;
3760  ;
3770  ;
3780  ;
3790  ;
3800  ;
3810  ;
3820  ;
3830  ;
3840  ;
3850  ;
3860  ;
3870  ;
3880  ;
3890  ;
3900  ;
3910  ;
3920  ;
3930  ;
3940  ;
3950  ;
3960  ;
3970  ;
3980  ;
3990  ;
4000  ;
4010  ;
4020  ;
4030  ;
4040  ;
4050  ;
4060  ;
4070  ;
4080  ;
4090  ;
4100  ;
4110  ;
4120  ;
4130  ;
4140  ;
4150  ;
4160  ;
4170  ;
4180  ;
4190  ;
4200  ;
4210  ;
4220  ;
4230  ;
4240  ;
4250  ;
4260  ;
4270  ;
4280  ;
4290  ;
4300  ;
4310  ;
4320  ;
4330  ;
4340  ;
4350  ;
4360  ;
4370  ;
4380  ;
4390  ;
4400  ;
4410  ;
4420  ;
4430  ;
4440  ;
4450  ;
4460  ;
4470  ;
4480  ;
4490  ;
4500  ;
4510  ;
4520  ;
4530  ;
4540  ;
4550  ;
4560  ;
4570  ;
4580  ;
4590  ;
4600  ;
4610  ;
4620  ;
4630  ;
4640  ;
4650  ;
4660  ;
4670  ;
4680  ;
4690  ;
4700  ;
4710  ;
4720  ;
4730  ;
4740  ;
4750  ;
4760  ;
4770  ;
4780  ;
4790  ;
4800  ;
4810  ;
4820  ;
4830  ;
4840  ;
4850  ;
4860  ;
4870  ;
4880  ;
4890  ;
4900  ;
4910  ;
4920  ;
4930  ;
4940  ;
4950  ;
4960  ;
4970  ;
4980  ;
4990  ;
5000  ;
5010  ;
5020  ;
5030  ;
5040  ;
5050  ;
5060  ;
5070  ;
5080  ;
5090  ;
5100  ;
5110  ;
5120  ;
5130  ;
5140  ;
WITH NEW CHARACTER.
SELECT NEXT CHARACTER.
RETURN.
SAVE THE CHARACTER TO BE
INSERTED, WHILE WE MAKE ROOM
FOR IT IN THE BUFFER...
SAVE THE CURRENT ADDRESS.
SAVE THE BUFFER’S ADDRESS.
SAVE BUFFER’S END ADDRESS.
SET SA=SELECT, SO CURRENT
LOCATION WILL BE START OF
THE BLOCK WE’LL MOVE.
ADVANCE TO NEXT CHARACTER
POSITION IN THE BUFFER.
IF WE’RE AT THE END OF THE
BUFFER, WE’LL OVERSTRIKE
INSTEAD OF INSERTING.
SET DEST=SELECT.
DESTINATION OF BLOCK MOVE
WILL BE ONE BYTE ABOVE
BLOCK’S INITIAL LOCATION.
DECREMENT END ADDRESS
OPEN UP ONE BYTE OF SPACE
AT CURRENT CHARACTER’S
LOCATION, BY MOVING TO DEST
THE BLOCK SPECIFIED BY SA, EA.
RESTORE EA SO IT POINTS
TO END OF BUFFER.
3460 1F65 68  PLA
3470 1F66 ED5515 STA EA+1
3480 ;
3490 ;
3500 1FE9 68  PLA RESTORE SA SO IT POINTS TO
3510 1FEA ED5215 STA SA START OF BUFFER.
3520 1FED 68  PLA
3530 1FEE BD5315 STA SA+1
3540 ;
3550 ;
3560 1F71 202B15 JSR POP.SL RESTORE SELECT SO IT POINTS
3570 ; TO CURRENT CHARACTER POSITION.
3580 ;
3590 ;
3600 1F74 68  PLA RESTORE NEW CHARACTER TO
3610 ; ACCUMULATOR. WE'VE CREATED
3620 ; A ONE-BYTE SPACE FOR IT, SO
3630 1F75 202D1F JSR STRIKE WE NEED ONLY OVERWRITE IT
3640 1F78 60 RTS AND RETURN.
3650 1F79 20B212 NEXTCH JSR GET.SL GET CURRENT CHARACTER.
3660 1F7C 20FF CMP #$FF IS IT END OF TEXT CHARACTER?
3670 1F7E F804 BEQ AN.ETX IF SO, RETURN TO CALLER,
3680 ; BEARING A NEGATIVE RETURN CODE.
3690 ;
3700 1FB0 20B317 JSR NEXTSL IF NOT, SELECT NEXT BYTE IN
3710 ; BUFFER.
3720 1F83 60 RTS RETURN PLUS IF WE INCREMENTED
3730 ; SELECT; MINUS IF SELECT
3740 ; ALREADY EQUALLED EA.
3750 ;
3760 1F84 A9FF AN.ETX LDA #$FF SINCE WE'RE ON AN ETX, WE
3770 1F85 60 RTS WILL RETURN MINUS, WITHOUT
3780 ; INCREMENTING SELECT.
3790 ;
3800 ;
3810 ;
3820 ;
3830 1F87 3B PREVSL SEC PREPARE TO COMPARE.
3840 1F88 AD5315 LDA SA+1 IS SELECT IN A HIGHER PAGE
3850 1F8B CD0612 CMP SELECT+1 THAN START OF BUFFER?
3860 1F8E 800C BCC SL.OK IF SO, SELECT MAY BE DECREMENTED
3870 1F90 D010 BNE NOT.OK IF SELECT IS IN A LOWER
3880 ; PAGE THAN SA, IT'S NOT OK.
3890 ;
3900 ;
3910 1F92 AD5215 LDA SA SELECT IS IN SAME PAGE AS SA.
3920 1F95 CD0512 CMP SELECT IS SELECT>SA?
3930 1F98 F017 BEQ NO.DEC IF SELECT=SA, DON'T DECREMENT
3940 ; SELECT.
3950 1F9A B006 BCS NOT.OK IF SELECT<SA, DON'T DECREMENT
3960 ; SELECT.
3970 1F9C 20A13 SL.OK JSR DEC.SL SELECT>SA, SO WE MAY
3980 ; DECREMENT SELECT AND IT
3990 ; WILL REMAIN IN THE BUFFER.
4000 1F9F 8300 LDA #0 SET A POSITIVE RETURN CODE...
4010 1FA1 60 RTS ...AND RETURN.
4040 1FA2 AD5215 NOT.OK LDA SA  
4050 1FA5 8D0512 STA SELECT  
4060 1FA8 AD5315 LDA SA+1  
4070 1FA8 8D0512 STA SELECT+1  
4080 1FAE 8D00 LDA #$0  
4090 1FB0 60 RTS  

SINCE SELECT(SA), IT IS NOT  
EVEN IN THE EDIT BUFFER. SO  
MAKE SELECT LEGAL, BY SETTING  
IT EQUAL TO SA.  
SET A POSITIVE RETURN CODE...  
...AND RETURN.

4100 ;  
4110 ;  
4120 1FB1 ASFF NO.DEC LDA #$FF  
4130 1FB3 60 RTS  
SELECT=SA, SO CHANGE  
NOTHING. RETURN WITH  
NEGATIVE RETURN CODE.

4140 ;  
4150 ;  
4160 ;  
4170 ;  

4180 1FB4 20A017 FLUSH JSR GOTOSA  
4190 1FB7 ASFF FLOOP LDA #$ETX  
SET SELECT=SA.  
PUT AN ETX CHARACTER  
INTO THE BUFFER.

4200 1FB9 202D13 JSR PUT.SL  
ADVANCE TO NEXT POSITION IN  
BUFFER.

4210 1FBC 208317 JSR NEXTS1  

4220 ;  
4230 1FBE 10F6 BPL FLOOP  
IF WE HAVEN'T REACHED END  
OF BUFFER, PUT AN ETX INTO  
THIS POSITION, TOO.

4240 ;  
4250 ;  
4260 ;  

4270 1FC1 20A017 JSR GOTOSA  
HAVING FILLED BUFFER WITH  
ETC CHARACTERS, RESET SELECT  
TO BEGINNING OF BUFFER.

4280 ;  
4290 ;  

4300 1FC4 60 RTS  
RETURN.

4310 1FC5 20A017 PRTBUF JSR GOTOSA  
SET SELECT TO START OF BUFFER  
SELECT PRINTER FOR OUTPUT.

4320 1FC8 201414 JSR PR.ON  
GET CURRENT CHARACTER.

4330 1FCB 209412 PRLOOP JSR GET.SL  
IS IT ETX?

4340 1FCE CSFF CMP #$ETX  
IF SO, WE'RE DONE.

4350 1FDF 0008 BEO ENDPRT  
IF NOT, PRINT IT.

4360 1FD2 204014 JSR PR.CHRS  
SELECT NEXT CHARACTER  
IF WE HAVEN'T REACHED THE  
END OF THE BUFFER, HANDLE  
THE CURRENT CHARACTER AS BEFORE.

4370 1FD5 203817 JSR NEXTSL  
HAVING REACHED END OF MESSAGE  
OR END OF BUFFER, RETURN TO  
CALLER OF EDITIT, Deselecting  
THE PRINTER AS WE DO SO.

4380 ;  
4400 ;  

4410 1FDA 4C1A14 ENDPRT JMP PR.OFF  
SAVE CURRENT ADDRESS.  
SAVE BUFFER'S START ADDRESS.

4420 ;  
4430 ;  
4440 ;  
4450 ;  
4460 ;  

4470 1FDD 201215 DELETE JSR PUSHL  
SET DEST=SELECT, BECAUSE  
WE'LL MOVE A BLOCK OF TEXT  
DOWN TO HERE, TO CLOSE UP  
THE BUFFER AT THE CURRENT  
CHARACTER.

4480 1FEB AD5315 LDA SA+1  
ADVANCE BY ONE BYTE THROUGH  
BUFFER, IF POSSIBLE.

4490 ;  
4500 1FEE 206716 JSR SAHERE  
SET SA=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.
MOVE BLOCK SPECIFIED BY
SA, EA TO DEST.

RESTORE INITIAL SA (WHICH
IS THE START ADDRESS OF THE
TEXT BUFFER, NOT OF THE BLOCK
WE JUST MOVED.)
RESTORE CURRENT ADDRESS.
RETURN TO CALLER.
Appendix C12:
Extending the Visible Monitor
APPENDIX C12: ASSEMBLER LISTING OF VISIBLE MONITOR EXTENSIONS

SEE CHAPTER 12 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

; EXTERNAL ADDRESSES

PRPAGE=$1400 STARTING PAGE OF PRINT UTILITIES.
PRINTR=PRPAGE
USER =PRPAGE+2

HEX.PG=$1500 ADDRESS OF PAGE IN WHICH HEXDUMP CODE STARTS.

TVDUMP=HEX.PG+$57
PRDUMP=HEX.PG+$AC

DSPAGE=$1900 STARTING PAGE OF DISASSEMBLER
**$1060**

***************

EXTENSIONS TO THE VISIBLE MONITOR

***************

1060 C950 EXTEND CMP $'P' IS IT THE 'P' KEY?

1062 D009 BNE IF.U IF NOT, PERFORM NEXT TEST.

1064 AD0014 LDA PRINTR IF SO, TOGGLE THE PRINTER

1067 49FF EOR $FF FLAG...

1069 BD0014 STA PRINTR AND RETURN TO CALLER.

106C 60 RTS

950 108F D009 BNE IF.H IF NOT, PERFORM NEXT TEST.

1081 AD0214 LDA USER IF SO, TOGGLE THE USER-

1084 49FF EOR $FF PROVIDED OUTPUT FLAG...

1087 BD0214 STA USER AND RETURN.

950 1089 60 RTS

1090 108A C940 IF.H CMP $'H' IS IT THE 'H' KEY?

1092 D00D BNE IF.M IF NOT, PERFORM NEXT TEST.

1094 AD0014 LDA PRINTR IS THE PRINTER SELECTED?

1097 D004 BNE NEXT.1 IF SO, PRINT A HEXDUMP...

109A 205715 JSR TUDUMP AND RETURN.

109D 60 RTS PRINT A HEXDUMP...

10A0 10DA 60 ...AND RETURN.

10A9 10BD C940 IF.M CMP $'M' IS IT THE 'M' KEY?

10BD D004 BNE IF.DIS IF NOT, PERFORM NEXT TEST.

10BF 20B417 JSR MOVER IF SO, LET USER SPECIFY AND

10C0 10E2 60 RTS AND MOVE A BLOCK OF MEMORY.

10C3 10E3 C93F IF.DIS CMP $'?' IS IT THE '?' KEY?
1160 10E5 D00D   BNE IF.T   IF NOT, PERFORM NEXT TEST.
1170 10E7 A00014  LDA PRINTR  IS THE PRINTER SELECTED?
1180 10EA D004   BNE NEXT.2  IF SO, PRINT A DISASSEMBLY.
1190 10EC 200919  JSR TV.DIS  IF NOT, DISASSEMBLE TO THE
1200 10EF 60    RTS          SCREEN AND RETURN.
1210 10F0 202619 NEXT.2 JSR PR.DIS  PRINT A DISASSEMBLY...
1220 10F3 60    RTS          AND RETURN.
1230                    ;
1240 10F4 C954   IF.T   CMP 'T   IS IT THE 'T' KEY?
1250 10F6 D004   BNE EXIT   IF NOT, RETURN.
1260 10F8 20021E  JSR EDITOR  IF SO, CALL THE SIMPLE
1270 10FB 60    RTS          TEXT EDITOR AND RETURN.
1280                    ;
1290 10FC 60  EXIT  RTS      EXTEND THE VISIBLE MONITOR
1300                    ;      EVEN FURTHER BY REPLACING
1310                    ;      THIS 'RTS' WITH A 'JMP' TO
1320                    ;      MORE TEST-AND-BRANCH CODE.

353
Appendix C13:
System Data Block for the Ohio Scientific C-1P
APPENDIX C13: ASSEMBLER LISTING OF SYSTEM DATA BLOCK FOR THE OHIO SCIENTIFIC C-1P

SEE APPENDIX B1 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 5502 PERSONAL COMPUTER

BY KEN SKIER

; *************************************************************

; SCREEN PARAMETERS

; *************************************************************

; HOME  .WORD $D065

; THIS IS THE ADDRESS OF THE CHARACTER IN THE UPPER LEFT CORNER OF THE SCREEN. THE ADDRESS OF HOME WILL VARY AS A FUNCTION OF YOUR VIDEO MONITOR. IF YOU CAN'T SEE THE VISIBLE MONITOR DISPLAY, ADJUST THE LOW BYTE.

; ROWINC .BYTE 32

; ADDRESS DIFFERENCE FROM ONE ROW TO THE NEXT.

; NUMCOLS .BYTE $18

; NUMBER OF COLUMNS ON SCREEN, COUNTING FROM ZERO.

; NUMROWS .BYTE $18

; NUMBER OF ROWS ON SCREEN, COUNTING FROM ZERO.
580 1805 D3 HIPAGE .BYTE $D3 HIGHEST PAGE IN SCREEN MEMORY.
590 1806 20 BLANK .BYTE $20 OSI DISPLAY CODE FOR A BLANK.
590 1807 10 ARROW .BYTE $10 OSI DISPLAY CODE FOR AN UP-ARROW
610 ;
620 ;
630 ;
640 ;
650 ;
660 ;
670 ;
680 ;
690 ;
700 ;
710 ;
720 ;
730 ;
740 ;
750 ;
760 ;
770 ;
780 ;
790 ;
800 ;
810 1808 ED FE ROMKEY .WORD $FEED "INPUT/OUTPUT VECTORS"
820 ;
830 ;
840 ;
850 ;
860 ;
870 ;
880 ;
890 180A 20 D7F ROMTUT .WORD $F2D POINTER TO ROUTINE THAT GETS AN ASCII CHARACTER FROM THE KEYBOARD. (NOTE: $FEED IS THE GENERAL CHARACTER-INPUT ROUTINE FOR OSI BASIC-IN-ROM COMPUTERS.)
900 ;
910 ;
920 ;
930 ;
940 ;
950 ;
960 180C BF F8 ROMPRT .WORD $FCB1 POINTER TO ROUTINE TO PRINT AN ASCII CHARACTER ON THE SCREEN (NOTE: $FEEF IS THE CHARACTER-OUTPUT ROUTINE FOR OSI BASIC-IN-ROM COMPUTERS.)
970 ;
980 ;
990 ;
1000 ;
1010 180E 1810 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.)
1020 ;
1030 ;
1040 ;
1050 ;
1060 ;
1070 ;
1080 1810 60 DUMMY RTS THIS IS A DUMMY SUBROUTINE. IT DOES NOTHING BUT RETURN.
1090 ;
1100 ;
1110 ;
1120 ;
1130 ;
1140 ;
1150 ;

358 BEYOND GAMES
CONVERT ASCII CHARACTER TO DISPLAY CODE

SINCE OSI DISPLAY CODES ARE THE SAME AS THE CORRESPONDING ASCII CHARACTERS, NO CONVERSION IS NECESSARY; FIXCHR IS A DUMMY.
Appendix C14:
System Data Block for the PET 2001
APPENDIX C14: ASSEMBLER LISTING OF
SYSTEM DATA BLOCK
FOR THE PET 2001

SEE APPENDIX B2 OF BEYOND GAMES: SYSTEM
SOFTWARE FOR YOUR 5502 PERSONAL COMPUTER

BY KEN SKIER

; *********************************************************
; SCREEN PARAMETERS
; *********************************************************
350 1000 *=1000

; HOME .WORD $8000 THIS IS THE ADDRESS OF THE
; CHARACTER IN THE UPPER LEFT
; CORNER OF THE SCREEN.
410 1000 0060

; ROWINC .BYTE $20 ADDRESS DIFFERENCE FROM ONE
; ROW TO THE NEXT.
470 1002 28

; TVCOLS .BYTE 39 NUMBER OF COLUMNS ON SCREEN,
; COUNTING FROM ZERO.
490 1003 27

; TVROWS .BYTE 24 NUMBER OF ROWS ON SCREEN,
; COUNTING FROM ZERO.
510 1004 18

; HIPAGE .BYTE $83 HIGHEST PAGE IN SCREEN MEMORY.
530 1005 83

; BLANK .BYTE $20 PET DISPLAY CODE FOR A BLANK.
540 1006 20 (IN NORMAL VIDEO MODE.)

; ARROW .BYTE $1E PET DISPLAY CODE FOR UP-ARROW.
560 1007 1E

363
ROMKEY .WORD PETFKEY  ; POINTER TO ROUTINE THAT GETS
                       ; AN ASCII CHARACTER FROM THE
                       ; KEYBOARD. (NOTE: PETFKEY
                       ; CALLS A ROM SUBROUTINE, BUT
                       ; PETFKEY IS NOT A PET ROM
                       ; SUBROUTINE.)

ROMVUT .WORD $FFD2  ; POINTER TO ROUTINE TO PRINT
                       ; AN ASCII CHARACTER ON THE SCREEN

ROMPRT .WORD DUMMY  ; POINTER TO ROUTINE TO SEND AN
                       ; ASCII CHARACTER TO THE PRINTER
                       ; (SET TO DUMMY UNTIL YOU MAKE
                       ; IT POINT TO THE CHARACTER-
                       ; OUTPUT ROUTINE THAT DRIVES
                       ; YOUR PRINTER.)

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.)

DUMMY RTS  ; THIS IS A DUMMY SUBROUTINE.
            ; IT DOES NOTHING BUT RETURN.
1130  ;
1200  ;
1210  ;
1220  1011 297F  FIXCHR AND #$7F  CLEAR BIT 7, TO MAKE IT
1230  ;  A LEGAL ASCII CHARACTER.
1240  1013 38  SEC  PREPARE TO COMPARE.
1250  1014 C940  CMP #$40  IS IT LESS THAN $40? (IS
1260  ;  IT A NUMBER OR PUNCTUATION
1270  ;  MARK?)
1280  1016 9011  BCC FIXEND  IF SO, NO CONVERSION NEEDED.
1290  ;
1300  1016 C960  CMP #$60  IS IT BETWEEN $40 AND $60?
1310  ;  IF SO, SUBTRACT $40 TO
1320  101A 900A  BCC SUB.40  CONVERT FROM ASCII TO PET.
1330  ;
1340  ;
1350  ;
1360  101C A20E  LDX #14  IT'S >= $60, SO WE MUST
1370  101E 8D4CE8  STA $946B  SET PET DISPLAY MODE FOR
1380  ;  CHARACTER SET THAT INCLUDES
1390  ;  LOWER CASE ALPHA CHARACTERS.
1400  1021 E920  SBC #$20  SUBTRACT $20 TO CONVERT
1410  ;  LOWER CASE ASCII TO PET CODE.
1420  1023 18  CLC
1430  1024 9003  BCC FIXEND
1435  ;
1440  1026 38  SUB.40 SEC  PREPARE TO SUBTRACT.
1450  1027 E940  SBC #$40  SUBTRACT $40 TO CONVERT ASCII
1460  ;  UPPER CASE CHAR TO PET CODE.
1470  1029 60  FIXEND RTS  RETURN, WITH A HOLDING
1480  ;  PET DISPLAY CODE FOR ASCII
1490  ;  ORIGINALLY IN A.
1500  ;
1510  ;
1520  ;
1530  ;
1540  ;
1550  ;  ************************************************************
1560  ;
1570  ;  GET AN ASCII CHARACTER FROM THE KEYBOARD
1580  ;
1590  ;  ************************************************************
1600  ;
1610  ;
1620  ;
1630  ;
1640  102A 20E4FF PETKEY JSR $FFE4  SCAN THE PET KEYBOARD
1650  102D 297F  AND #$7F  CLEAR BIT 7, TO BE SURE
1660  ;  IT'S A LEGAL ASCII CHARACTER.
1670  102F F0F9  BEQ PETKEY  ZERO MEANS NO KEY, SO
1680  ;  SCAN AGAIN.
1690  ;
1700  1031 60  RTS  RETURN WITH ASCII CHARACTER
1710  ;  FROM THE KEYBOARD.
Appendix C15:
System Data Block for the Apple II
APPENDIX C15: ASSEMBLER LISTING OF
SYSTEM DATA BLOCK
FOR THE APPLE II

SEE APPENDIX B3 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

BY KEN SKIER

SCREEN PARAMETERS

10  #1000

410  #1000 0000  HOME .WORD $0400

420

430

440

450

460

470  #1002 00  ROWINC .BYTE $80

480

490  #1003 27  TUCOLS .BYTE 39

500

510  #1004 07  TUROWS .BYTE 7

520

530  #1005 07  HIPAGE .BYTE $07

540

550  #1006 00  BLANK .BYTE $A0

560

570

THIS IS THE ADDRESS OF THE CHARACTER IN THE UPPER LEFT CORNER OF THE SCREEN. (WHEN YOU ARE DISPLAYING LOW-RESOLUTION GRAPHICS AND TEXT PAGE 1.) ADDRESS DIFFERENCE FROM ONE ROW TO THE NEXT. NUMBER OF COLUMNS ON SCREEN, COUNTING FROM ZERO. NUMBER OF ROWS ON SCREEN, COUNTING FROM ZERO. HIGHEST PAGE IN SCREEN MEMORY. (WITH LOW-RES PAGE 1 SELECTED.) APPLE II DISPLAY CODE FOR A BLANK: A DARK BOX, USED AS A SPACE WHEN APPLE II IS IN
NORMAL DISPLAY MODE (WHITE CHARACTERS ON A DARK BACKGROUND.)

APPLE II DISPLAY CODE FOR A CARAT (USED BECAUSE APPLE II HAS NO UP-ARROW.)

INPUT/OUTPUT VECTORS

ROMKEY .WORD APLKEY  POINTER TO ROUTINE THAT GETS AN ASCII CHARACTER FROM THE KEYBOARD. (NOTE: APLKEY CALLS A ROM SUBROUTINE, BUT APLKEY IS NOT AN APPLE ROM SUBROUTINE.)

ROMTUT .WORD APLTUT  POINTER TO ROUTINE TO PRINT AN ASCII CHARACTER ON THE SCREEN

ROMPRD .WORD DUMMY  POINTER TO ROUTINE TO SEND AN ASCII CHARACTER TO THE PRINTER (SET TO DUMMY UNTIL YOU MAKE IT POINT TO THE CHARACTER-OUTPUT ROUTINE THAT DRIVES YOUR PRINTER.)

YOU MAY WANT TO SET ROMPRD SO IT POINTS TO $FDED, THE APPLE II'S GENERAL CHARACTER OUTPUT ROUTINE. $FDED WILL PRINT TO A PRINTER IF YOU TELL YOUR APPLE II ROM SOFTWARE TO SELECT YOUR PRINTER AS AN OUTPUT DEVICE. DO THAT IN BASIC BY TYPING "PR #n", WHERE N IS THE NUMBER OF THE SLOT HOLDING THE CIRCUIT CARD THAT DRIVES YOUR PRINTER.
1150 ;
1160 ;
1170 ;
1180 100E 1010 USROUT .WORD DUMMY POINTER TO USER-WRITTEN OUTPUT
1190 ; ROUTINE. (SET HERE TO DUMMY
1200 ; UNTIL YOU SET IT TO POINT
1210 ; TO YOUR OWN CHARACTER-OUTPUT
1220 ; ROUTINE.)
1230 ;
1240 ;
1250 1010 60 DUMMY RTS THIS IS A DUMMY SUBROUTINE.
1260 ; IT DOES NOTHING BUT RETURN.
1270 ;
1280 ;
1290 ;
1300 ;
1310 ;
1320 ;
1330 ;*****************************************************************
1340 ;
1350 ; CONVERT ASCII CHARACTER TO DISPLAY CODE
1360 ;
1370 ;*****************************************************************
1380 ;
1390 ;
1400 ;
1410 ;
1420 ;
1430 1011 0380 FIXCHR ORA #$80 SET BIT 7, SO CHARACTER
1440 ; WILL DISPLAY IN NORMAL MODE.
1450 1013 60 RTS RETURN.
1460 ;
1470 ;
1480 ;
1490 ;
1500 ;
1510 ;*****************************************************************
1520 ;
1530 ; GET AN ASCII CHARACTER FROM THE KEYBOARD
1540 ;
1550 ;*****************************************************************
1560 ;
1570 ;
1580 ;
1590 ;
1600 1014 203FD APLKEY JSR #$D35 GET KEYBOARD CHARACTER WITH
1610 ; BIT 7 SET.
1620 1017 297F AND #$7F CLEAR BIT 7.
1630 ;
1640 1019 60 RTS RETURN WITH ASCII CHARACTER
1650 ; FROM THE KEYBOARD.
1660 ;
1670 ;
1680 ;
1690 ;
1700 ;
1710 ;
1720 ;

371
1730 ; ************************************************************************
1740 ; PRINT AN ASCII CHARACTER ON THE SCREEN
1750 ; ************************************************************************
1760 ;
1770 ;
1780 ;
1790 ;
1800 ;
1810 ;
1820 ;
1830 ;
1840 101A 0300 AFLVT ORA #$00 SET BIT 7 SO CHARACTER WILL
1850 ; PRINT IN NORMAL MODE.
1860 181C 20FD $00 JSR $FBFD CALL APPLE II ROM ROUTINE TO
1870 ; PRINT A CHARACTER TO SCREEN.
1880 181F 60 RTS RETURN TO CALLER.
Appendix C16:
System Data Block for the Atari 800
APPENDIX C16: ASSEMBLER LISTING OF
SYSTEM DATA BLOCK
FOR THE ATARI 800

SEE APPENDIX B4 OF BEYOND GAMES: SYSTEM
SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

BY KEN SKIER

373 0003= TV_PTR=0
378
398 1100= TUSUBS=$1100
400 1113= CLR.XY=TUSUBS+$13
410 112B= TVHOME=TUSUBS+$2B
420 113C= TUTOXY=TUSUBS+$3C
430 1175= TUDOWN=TUSUBS+$75
440 11C4= TUPUSH=TUSUBS+$C4
450 11D3= TV.POP=TUSUBS+$D3
460 117C= VUCHAR=TUSUBS+$7C
470
480 1500= HEX.PG=$1500
490 1552= SA=HEX.PG+$52
500 1554= EA=SA+2
510
520 1700= MOV.PG=$1700
530 17B2= BEST=MOV.PG+$B2
540 17B6= MOV.EA=MOV.PG+$D6
550
558
570
SCREEN PARAMETERS

ADDRESS OF THE
CHARACTER IN THE UPPER LEFT
CORNER OF THE SCREEN.
(FOR AN ATARI 800 W/32K RAM,
in screen mode 0.)
YOU MUST USE SCREEN MODE 0.
APPENDIX B4 INCLUDES A BASIC
PROGRAM TO START THE VISIBLE
MONITOR. IT SETS HOME FOR
YOUR SYSTEM.

NOTE:
IF HOME IS LESS THAN $2000
(B192 DECIMAL), THE SCREEN
WILL INTERFERE WITH THE
SOFTWARE IN THIS BOOK.

IF YOU TRY TO RUN THIS
SOFTWARE ON AN 8K SYSTEM, DON'T
USE THE DISASSEMBLER OR THE
SIMPLE TEXT EDITOR, BECAUSE
SCREEN OPERATIONS WILL WRITE
OVER THEM, AND THEY'LL CRASH.

ADDRESS DIFFERENCE FROM ONE
ROW TO THE NEXT.
NUMBER OF COLUMNS ON SCREEN,
COUNTING FROM ZERO.
NUMBER OF ROWS ON SCREEN,
COUNTING FROM ZERO.
HIGHEST PAGE IN SCREEN
MEMORY. LIKE HOME, HIPIAGE
VARIES ACCORDING TO THE
AMOUNT OF RAM IN YOUR ATARI.
HIPIAGE IS SET FOR YOUR SYSTEM
WHEN YOU RUN THE BASIC PROGRAM
IN APPENDIX B4 TO START
THE VISIBLE MONITOR.

ATARI DISPLAY CODE FOR A BLANK
ATARI DISPLAY CODE FOR
AN UP-ARROW.
1160  ;
1170  ;
1180  ;
1190  ;
1200  ;
1210  ;
1220  ;
1230  ;
1240  ;
1250  ;
1260  ;
1270  ;
1280  ;
1290  ;
1300  ;
1310  ;
1320  ;
1330  ;
1340  ;
1350  1000 2810  ROMKEY .WORD ATRLKEY  POINTER TO ROUTINE THAT GETS
1360  ;  AN ASCII CHARACTER FROM THE
1370  ;  KEYBOARD.
1380  ;
1390  ;
1400  100A 3610  ROMOUT .WORD TUTSIN  POINTER TO ROUTINE TO PRINT
1410  ;  AN ASCII CHARACTER ON THE SCREEN
1420  ;
1430  ;
1440  100C 1010  ROMPRT .WORD DUMMY  POINTER TO ROUTINE TO SEND AN
1450  ;  ASCII CHARACTER TO THE PRINTER
1460  ;  (SET TO DUMMY UNTIL YOU MAKE
1470  ;  IT POINT TO THE CHARACTER-
1480  ;  OUTPUT ROUTINE
1490  ;  THAT DRIVES YOUR PRINTER.
1500  ;
1510  ;
1520  ;
1530  100E 1010  USROUT .WORD DUMMY  POINTER TO USER-WRITTEN OUTPUT
1540  ;  ROUTINE. (SET HERE TO DUMMY
1550  ;  UNTIL YOU SET IT TO POINT
1560  ;  TO YOUR OWEN CHARACTER-OUTPUT
1570  ;  ROUTINE.)
1580  ;
1590  ;
1600  1010 60  DUMMY RTS  THIS IS A DUMMY SUBROUTINE.
1610  ;  IT DOES NOTHING BUT RETURN.
1620  ;
1630  ;
1640  ;
1650  ;
1660  ;
1670  ;
1680  ;
1690  ;
1700  ;  CONVERT ASCII CHARACTER TO DISPLAY CODE
1710  ;
1720  ;
1730  ;

377
1748 ;
1750 ;
1760 ;
1770 ;
1780 1011 297F FIXCHR AND #$7F CLEAR BIT 7 SO CHARACTER IS
1790 ; A LEGITIMATE ASCII CHARACTER.
1800 1013 38 SEC PREPARE TO COMPARE.
1810 1014 C920 CMP #$20 IS CHARACTER < $20?
1820 1016 9008 BCC BADCHR IF SO, IT'S NOT A VIEWABLE
1830 ; ASCII CHARACTER, SO RETURN
1840 ; A BLANK.
1850 ;
1860 1018 C950 CMP #$50 IS CHARACTER < $50?
1870 101A 9008 BCC SUB.20 IF SO, SUBTRACT $20 AND RETURN.
1880 101C C97B CMP #$7B CHARACTER < $7B?
1890 101E 9007 BCC FIXEND IF SO, NO CONVERSION IS NEEDED.
1900 ;
1910 1020 AD610 BADCHR LDA BLANK THE CHARACTER IS NOT A
1920 ; VIEWABLE ASCII CHARACTER,
1930 1023 60 RTS SO RETURN A BLANK.
1940 1024 3B SUB.20 SEC PREPARE TO SUBTRACT.
1950 1025 E920 SBC #$20 SUBTRACT $20 TO CONVERT ASCII
1960 ; TO ATARI DISPLAY CODE.
1970 1027 60 FIXEND RTS RETURN WITH ATARI DISPLAY
1980 ; CODE FOR ORIGINAL ASCII
1990 ; CHARACTER.
2000 ;
2010 ;
2020 ;
2030 ;
2040 ;
2050 ;
2060 ;
2070 ;
2080 ;
2090 ;
2100 ; GET AN ASCII CHARACTER FROM THE KEYBOARD
2110 ;
2120 ;
2130 ;
2140 ;
2150 ;
2160 ;
2170 ;
2180 ;
2190 ;
2200 ;
2210 1028 ADFO2 ATRKEY LDA $02FC HAS A KEY BEEN DEPRESSED?
2220 1028 CSFF CMP #$FF $FF MEANS NO KEY.
2230 102D F0F9 BEQ ATRKEY IF NOT, LOOK AGAIN.
2240 ;
2250 ;
2260 ;
2270 ;
2280 102F 8B TAY A KEY HAS GONE DOWN.
2290 ; ACCUMULATOR HOLDS ITS
2300 ; HARDWARE KEY-CODE.
2310 ; PREPARE TO USE THAT CODE AS

378 BEYOND GAMES
LDA ATR KYS,Y
LOOK UP CHARACTER FOR THAT
KEY AND SHIFT STATE.

RTS
RETURN WITH ASCII CHARACTER
FOR THAT KEY AND SHIFT STATE.

; ***********************
PRINT AN ASCII CHARACTER ON THE SCREEN
; ***********************

; CR=0D
ASCII CARRIAGE RETURN.

; LF=0A
ASCII LINEFEED CHARACTER.

; TUCHAR .BYTE 0
THIS BYTE HOLDS CHARACTER
TO BE DISPLAYED. (ALSO,
CHARACTER MOST RECENTLY
DISPLAYED, USING TUTSIM.)

; TV.COL .BYTE 0
THIS BYTE HOLDS COLUMN IN
WHICH CHARACTER WILL NEXT
APPEAR. WE MAY THINK OF IT
AS THE POSITION OF AN
ELECTRONIC "PRINT-HEAD".

; TUTSIM CMP #$CR
IS CHARACTER AN ASCII
CARRIAGE RETURN?

; BNE LFTEST
IF NOT, PERFORM NEXT TEST.

; LDA #$0
RESET TV COLUMN TO

; STA TV.COL
LEFT MARGIN AND

; RTS
RETURN.

; LFTEST CMP #$LF
IS IT A LINEFEED CHARACTER?

; BNE CHSAVE
IF NOT, HANDLE IT AS A CHARACTER
SCROLL TEXT UP FOR A LINEFEED.

; JMP SCROLL

; SINCE IT'S NOT CR OR LF,
LET'S SAVE IT.

; JSR TUPUSH
SAVE ZERO PAGE BYTES WE'LL USE.

; LDY TUROWS
SET TV.PTR TO CURRENT

; LDX TV.COL
POSITION OF "PRINT-HEAD".

; JSR TUTOXY

; LDA TUCHAR
GET CHARACTER TO BE DISPLAYED.

; JSR VUCHAR
SHOW IT.

; INC TV.COL
ADVANCE "PRINT-HEAD" TO NEXT
2900 ;
2910 ;
2920 105F AD3510 LDA T.V.COLL.
2930 1052 CD310 CMP T.V.COLS
2940 ;
2950 1055 D066 BNE TUTEND
2960 1037 203A10 JSR RESET
2970 105A 20600E JSR SCROLL
2980 105D 20D311 TUTEND JSR TV.POP
2990 ;
3000 1070 60 RTS
3010 ;
3020 ;
3030 ;
3040 ;
3050 ;
3060 ;
3070 ;
3080 ;
3090 ;
3100 ;
3110 ;
3120 ;
3130 ;
3140 ;
3150 ;
3160 0E80 /*=OEB0
3170 ;
3180 ;
3190 ;
3200 ;
3210 ;
3220 0E80 20C411 SCROLL JSR TUPUSH
3230 ;
3240 ;
3250 ;
3260 ;
3270 ;
3280 ;
3290 ;
3300 ;
3310 0E83 ADB317 LDA DEST+1
3320 0E56 43 PHA
3330 0E97 ADE217 LDA DEST
3340 0E9A 48 PHA
3350 0E9B AD515 LDA EA+1
3360 0E9E 48 PHA
3370 0E9F AD5415 LDA EA
3380 0E92 48 PHA
3390 0E93 AD5315 LDA SA+1
3400 0E96 48 PHA
3410 0E97 AD5215 LDA SA
3420 0E9A 48 PHA
3430 ;
3440 ;
3450 0E9B 202B11 JSR TVHOME
3460 0E9E A500 LDA TV.PTR
3470 0E9A 8DB217 STA DEST

SCREEN POSITION.
HAS "PRINT-HEAD" REACHED RIGHT EDGE OF SCREEN?
IF NOT, PREPARE TO RETURN.
IF SO, RESET "PRINT-HEAD" TO LEFT MARGIN AND SCROLL TEXT.
RESTORE ZERO PAGE BYTES WE USED, AND RETURN.

SCROLL TEXT UP ON SCREEN

SAVE ZERO PAGE BYTES WE'LL USE.
SCROLLING IS SIMPLY MOVING THE CONTENTS OF SCREEN MEMORY UP BY ONE ROW. BEFORE WE MOVE ANYTHING, HOWEVER, LET'S SAVE SA, EA, AND DEST--THE MOVE PARAMETERS.

NOW SA, EA, AND DEST ARE SAVED.
SET TV.PTR TO HOME POSITION.
SET DEST=HOME, SINCE HE'LL MOVE THE CONTENTS OF SCREEN
LDA TV.PTR+1
STA DEST+1
JSR TUDOWN
LDA TV.PTR
STA SA
LDA TV.PTR+1
STA SA+1
LDX TUCOLS
LDY TROWS
JSR TUTOXY
LDA TV.PTR
STA EA
LDA TV.PTR+1
STA EA+1
JSR MOV.EA
LDY TROWS
LDX #$0
JSR TUTOXY
LDX TUCOLS
LDY #$1
JSR CLR.XY
PLA
STA SA
PLA
STA SA+1
PLA
STA EA
PLA
STA EA+1
PLA
STA DEST
PLA
STA DEST+1
JSR TV.POP
RESTORE ZERO PAGE BYTES WE
USED.
RTS
RETURN.

bloque de definiciones de teclado
4069 ;
4370 ;
4390 ;
4090 ;
4100 ;
4110 0F00 ;
4120 ;
4130 ;
4140 ;
4150 ;
4160 ;
4170 027= APOSTR=$27 ASCII APOSTROPHE.
4180 025E= CARAT=$5E ASCII CARAT.
4190 001B= ESC=$1B ASCII ESCAPE CHARACTER.
4200 0020= SPACE=$20 ASCII SPACE.
4210 0009= TAB=$9 ASCII TAB CHARACTER.
4220 005B= BACKSL=$5B ASCII BACKSLASH CHARACTER.
4230 0058= BACKSP=$8 ASCII BACKSPACE CHARACTER.
4240 007A= LBRKT=$7A ASCII LEFT BRACKET.
4250 005D= RBRKT=$5D ASCII RIGHT BRACKET.
4260 007F= DELETE=$7F ASCII DELETE CHARACTER.
4270 ;
4280 ;
4290 ;
4300 0F00 6C ATRKYS .BYTE 'l,j;/,0,0,'k+*o'0,'pu' ,CR,'i='
4300 0F01 6A
4300 0F02 3B
4300 0F03 00
4300 0F04 00
4300 0F05 6B
4300 0F06 2B
4300 0F07 2A
4300 0F08 6F
4300 0F09 00
4300 0F0A 70
4300 0F0B 75
4300 0F0C 00
4300 0F0D 69
4300 0F0E 2D
4300 0F0F 3D
4310 0F10 76
4310 0F11 00
4310 0F12 63
4310 0F13 00
4310 0F14 00
4310 0F15 62
4310 0F16 70
4310 0F17 7A
4310 0F18 34
4310 0F19 00
4310 0F1A 53
4310 0F1B 36
4310 0F1C 1B
4310 0F1D 35
4310 0F1E 32
4310 0F1F 31
4320 0F20 2C
4320 0F21 20
4320 0F22 20
382 BEYOND GAMES
4320 0F22 2E
4320 0F23 6E
4320 0F24 00
4320 0F25 6D
4320 0F26 2F
4320 0F27 00
4320 0F28 72
4320 0F29 00
4320 0F2A 65
4320 0F2B 79
4320 0F2C 09
4320 0F2D 74
4320 0F2E 77
4320 0F2F 71
4330 0F30 3B
4330 0F31 00
4330 0F32 30
4330 0F33 37
4330 0F34 08
4330 0F35 38
4330 0F36 3C
4330 0F37 3E
4330 0F38 66
4330 0F39 60
4330 0F3A 64
4330 0F3B 00
4330 0F3C 00
4330 0F3D 67
4330 0F3E 73
4330 0F3F 61
4340
4350
4360
4370
4380
4390
4400 0F40 4C
4400 0F41 4A
4400 0F42 3A
4400 0F43 00
4400 0F44 20
4400 0F45 4B
4400 0F46 5B
4400 0F47 5E
4410 0F48 4F
4410 0F49 00
4410 0F4A 50
4410 0F4B 55
4410 0F4C 0D
4410 0F4D 49
4410 0F4E 2D
4410 0F4F 3D
4420 0F50 56
4420 0F51 00
4420 0F52 43
4420 0F53 00
4420 0F54 68
4420 0F55 42

.BYTE '9',0,'07',BACKSP,'8<fhd',0,0,'gca'

.FOLLOWING 64 BYTES CONTAIN ASCII CODES FOR SHIFTED KEYS.

.BYTE 'LJ:',0,0,'K',BACKSL,CARAT

.BYTE 'O',0,' Pu',CR,'I='

.BYTE 'U',0,'C',0,0,'EX24',0,'35',ESC.'%'

383
4420 0F66 58
4420 0F57 5A
4420 0F58 34
4420 0F59 00
4420 0F5A 33
4420 0F5B 36
4420 0F5C 18
4420 0F5D 25
4420 0F5E 22
4420 0F5F 21
4430 0F60 5A
4430 0F61 20
4430 0F62 5D
4430 0F63 4E
4430 0F64 00
4430 0F65 40
4430 0F66 3F
4430 0F67 00
4440 0F68 52
4440 0F69 00
4440 0F6A 45
4440 0F6B 59
4440 0F6C 09
4440 0F6D 54
4440 0F6E 57
4440 0F6F 51
4450 0F70 28
4450 0F71 00
4450 0F72 29
4450 0F73 27
4450 0F74 7F
4450 0F75 40
4450 0F76 00
4450 0F77 00
4460 0F78 46
4460 0F79 48
4460 0F7A 44
4460 0F7B 00
4460 0F7C 00
4460 0F7D 47
4460 0F7E 53
4460 0F7F 41
4470
4480
4490
4500
4510
4520
4530
4540 0F80 00
4540 0F81 00
4540 0F82 00
4540 0F83 00
4540 0F84 00
4540 0F85 00
4540 0F86 00
4540 0F87 00
4540 0F88 00

.BYTE LBRAKT, SPACE, RBRAKT, 'N', 0, 'M?', 0

.BYTE 'R', 0, 'EY', TAB, 'THQ'

.BYTE '(' , 0, ')' , APOSTR, DELETE, 'O', 0, 0

.BYTE 'FHD', 0, 0, 'GRS'

THE FOLLOWING 128 BYTES CONTAIN CHARACTER CODES FOR CONTROL SHIFTED KEYS. EDITOR FUNCTION KEYS ARE DEFINED.

.BYTE 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
.BYTE 0,0,3,0,0,0,0,0,0,0,0,0,0,0,0

.BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

.BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

.BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4590</td>
<td>0FC3</td>
</tr>
<tr>
<td>4590</td>
<td>0FC4</td>
</tr>
<tr>
<td>4590</td>
<td>0FC5</td>
</tr>
<tr>
<td>4590</td>
<td>0FC6</td>
</tr>
<tr>
<td>4590</td>
<td>0FC7</td>
</tr>
<tr>
<td>4590</td>
<td>0FC8</td>
</tr>
<tr>
<td>4590</td>
<td>0FC9</td>
</tr>
<tr>
<td>4590</td>
<td>0FCA</td>
</tr>
<tr>
<td>4590</td>
<td>0FCB</td>
</tr>
<tr>
<td>4590</td>
<td>0FCC</td>
</tr>
<tr>
<td>4590</td>
<td>0FCD</td>
</tr>
<tr>
<td>4590</td>
<td>0FCE</td>
</tr>
<tr>
<td>4590</td>
<td>0FCF</td>
</tr>
<tr>
<td>4590</td>
<td>0FD0</td>
</tr>
<tr>
<td>4590</td>
<td>0FD1</td>
</tr>
<tr>
<td>4590</td>
<td>0FD2</td>
</tr>
<tr>
<td>4590</td>
<td>0FD3</td>
</tr>
<tr>
<td>4590</td>
<td>0FD4</td>
</tr>
<tr>
<td>4590</td>
<td>0FD5</td>
</tr>
<tr>
<td>4590</td>
<td>0FD6</td>
</tr>
<tr>
<td>4590</td>
<td>0FD7</td>
</tr>
<tr>
<td>4590</td>
<td>0FD8</td>
</tr>
<tr>
<td>4590</td>
<td>0FD9</td>
</tr>
<tr>
<td>4590</td>
<td>0FDA</td>
</tr>
<tr>
<td>4590</td>
<td>0FDB</td>
</tr>
<tr>
<td>4590</td>
<td>0FDC</td>
</tr>
<tr>
<td>4590</td>
<td>0FDD</td>
</tr>
<tr>
<td>4590</td>
<td>0FDE</td>
</tr>
<tr>
<td>4590</td>
<td>0DFD</td>
</tr>
<tr>
<td>4590</td>
<td>0FED</td>
</tr>
<tr>
<td>4590</td>
<td>0FE0</td>
</tr>
<tr>
<td>4590</td>
<td>0FE1</td>
</tr>
<tr>
<td>4590</td>
<td>0FE2</td>
</tr>
<tr>
<td>4590</td>
<td>0FE3</td>
</tr>
<tr>
<td>4590</td>
<td>0FE4</td>
</tr>
<tr>
<td>4590</td>
<td>0FE5</td>
</tr>
<tr>
<td>4590</td>
<td>0FE6</td>
</tr>
<tr>
<td>4590</td>
<td>0FE7</td>
</tr>
<tr>
<td>4590</td>
<td>0FE8</td>
</tr>
<tr>
<td>4590</td>
<td>0FE9</td>
</tr>
<tr>
<td>4590</td>
<td>0FEA</td>
</tr>
<tr>
<td>4590</td>
<td>0FEB</td>
</tr>
<tr>
<td>4590</td>
<td>0FEC</td>
</tr>
<tr>
<td>4590</td>
<td>0FED</td>
</tr>
<tr>
<td>4590</td>
<td>0FEE</td>
</tr>
<tr>
<td>4590</td>
<td>0FEF</td>
</tr>
<tr>
<td>4610</td>
<td>0F00</td>
</tr>
<tr>
<td>4610</td>
<td>0F01</td>
</tr>
<tr>
<td>4610</td>
<td>0F02</td>
</tr>
<tr>
<td>4610</td>
<td>0F03</td>
</tr>
<tr>
<td>4610</td>
<td>0F04</td>
</tr>
<tr>
<td>4610</td>
<td>0F05</td>
</tr>
<tr>
<td>4610</td>
<td>0F06</td>
</tr>
<tr>
<td>4610</td>
<td>0F07</td>
</tr>
<tr>
<td>4610</td>
<td>0F08</td>
</tr>
<tr>
<td>4610</td>
<td>0F09</td>
</tr>
<tr>
<td>4610</td>
<td>0F0A</td>
</tr>
<tr>
<td>4610</td>
<td>0F0B</td>
</tr>
<tr>
<td>4610</td>
<td>0F0C</td>
</tr>
</tbody>
</table>

```
.BYTE 0,0,0,0,0,0,0,0,0,0,0,0
.BYTE 0,0,0,0,0,0,0,0,0,0,0,0
.BYTE 0,0,0,0,0,0,0,0,0,0,0,0
```
Appendix D1:
Screen Utilities

APPENDIX D1: SCREEN UTILITIES

SEE CHAPTER 5 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER.

DUMPING $1100-$11FF

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

1100 20 C4 11 20 2B 11 AE 03 10 AC 04 10 20 13 11 20
1110 D3 11 60 6E 2A 11 98 AA AD 06 10 AC 2A 11 91 00
1120 88 10 FB 20 76 11 CA 10 EF 60 19 A2 00 AD 00 18
1130 90 0A AD 04 10 4A A8 AD 03 10 4A AA 38 EC 03 10
1140 90 03 AE 03 10 38 CC 04 10 90 03 AC 04 10 AD 00
1150 10 05 00 AD 01 10 95 01 00 DB 8A 18 65 00 90 03
1160 E6 01 18 C0 00 F0 0B 18 6D 02 10 90 02 E6 01 88
1170 DB F5 05 00 28 60 AD 02 10 18 90 05 20 9B 11 A9
1180 01 0B DB 18 65 00 90 02 E6 01 05 00 38 AD 05 10
1190 C5 01 EC 05 AD 01 10 05 01 28 60 20 11 10 A0 00
11A0 91 00 60 4A 4A 4A 4A 20 B6 11 20 7C 11 68 20
11B0 B6 11 20 7C 11 60 08 DB 29 0F C9 0A 30 02 69 06
11C0 69 30 28 6A 6A 6A 6A A5 01 48 A5 00 48 88 48
11D0 6A 4A 60 6A AA 6A AB 6A 65 00 63 65 01 9B 49 8A
11E0 49 60 00 00 00 00 00 00 00 00 00 00 00 00 00
11F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```
Appendix D2:
Visible Monitor (Top Level and Display Subroutines)

**APPENDIX D2:**  
THE VISIBLE MONITOR (TOP LEVEL AND DISPLAY SUBROUTINES)

SEE CHAPTER 6 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

DUMPING $1200-$12DF

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1200</td>
<td>00 0C 00 00 31 05 12 08 09 20 12 12 20 E3 12 18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1210</td>
<td>90 FC 20 C4 11 20 25 12 20 34 12 20 5C 12 20 AF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1220</td>
<td>12 20 D3 11 60 A2 02 0A 02 20 3C 11 A0 00 0C 51 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1230</td>
<td>20 13 11 60 A2 00 A0 02 20 3C 11 A0 00 0C 51 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1240</td>
<td>03 52 12 20 7C 11 EE 51 12 AC 51 12 C0 0A 0D F0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1250</td>
<td>60 0A 41 20 20 5B 20 20 59 20 20 50 A2 02 A0 03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1260</td>
<td>20 3C 11 AD 06 12 20 A3 11 AD 05 12 20 A3 11 20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1270</td>
<td>7F 11 20 94 12 48 20 A3 11 20 7F 11 68 20 7C 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1280</td>
<td>20 7F 11 62 00 BD 01 12 20 A3 11 20 7F 11 68 E0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1290</td>
<td>04 00 F2 60 A5 02 48 A6 03 AD 05 12 65 02 AD 06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12A0</td>
<td>12 85 03 A0 00 B1 02 A6 68 52 02 86 03 58 68 A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12B0</td>
<td>02 A0 04 20 3C 11 AC 00 12 38 C0 07 90 05 A0 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12C0</td>
<td>8C 00 12 89 CD 12 A0 AD 07 10 91 00 60 03 05 08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12D0</td>
<td>00 0E 11 14 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

390 BEYOND GAMES
Appendix D3:
Visible Monitor (Update Subroutine)

APPENDIX D3:  THE VISIBLE MONITOR (UPDATE SUBROUTINE)

SEE CHAPTER 6 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER.

DUMPING $12E0-$13FF

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12E0</td>
<td>6C</td>
<td>B8</td>
<td>10</td>
<td>20</td>
<td>E0</td>
<td>12</td>
<td>C9</td>
<td>3E</td>
<td>D0</td>
<td>10</td>
<td>EE</td>
</tr>
<tr>
<td>12F0</td>
<td>C9</td>
<td>07</td>
<td>D0</td>
<td>05</td>
<td>A9</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>12</td>
<td>60</td>
<td>C9</td>
</tr>
<tr>
<td>1300</td>
<td>12</td>
<td>10</td>
<td>05</td>
<td>A9</td>
<td>00</td>
<td>00</td>
<td>12</td>
<td>60</td>
<td>C9</td>
<td>20</td>
<td>D0</td>
</tr>
<tr>
<td>1310</td>
<td>D0</td>
<td>03</td>
<td>EE</td>
<td>06</td>
<td>12</td>
<td>60</td>
<td>C9</td>
<td>00</td>
<td>D0</td>
<td>0C</td>
<td>AD</td>
</tr>
<tr>
<td>1320</td>
<td>06</td>
<td>12</td>
<td>CE</td>
<td>05</td>
<td>12</td>
<td>60</td>
<td>AE</td>
<td>00</td>
<td>12</td>
<td>E0</td>
<td>02</td>
</tr>
<tr>
<td>1330</td>
<td>48</td>
<td>A5</td>
<td>01</td>
<td>AD</td>
<td>05</td>
<td>12</td>
<td>BS</td>
<td>00</td>
<td>AD</td>
<td>06</td>
<td>12</td>
</tr>
<tr>
<td>1340</td>
<td>91</td>
<td>08</td>
<td>EE</td>
<td>01</td>
<td>68</td>
<td>68</td>
<td>00</td>
<td>68</td>
<td>C9</td>
<td>47</td>
<td>D0</td>
</tr>
<tr>
<td>1350</td>
<td>02</td>
<td>12</td>
<td>AD</td>
<td>04</td>
<td>12</td>
<td>48</td>
<td>AD</td>
<td>01</td>
<td>12</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>1360</td>
<td>12</td>
<td>EE</td>
<td>02</td>
<td>12</td>
<td>0C</td>
<td>03</td>
<td>12</td>
<td>60</td>
<td>0D</td>
<td>04</td>
<td>12</td>
</tr>
<tr>
<td>1370</td>
<td>20</td>
<td>05</td>
<td>13</td>
<td>30</td>
<td>48</td>
<td>A8</td>
<td>68</td>
<td>90</td>
<td>AE</td>
<td>00</td>
<td>12</td>
</tr>
<tr>
<td>1380</td>
<td>06</td>
<td>B6</td>
<td>12</td>
<td>CE</td>
<td>06</td>
<td>12</td>
<td>CA</td>
<td>10</td>
<td>F6</td>
<td>98</td>
<td>00</td>
</tr>
<tr>
<td>1390</td>
<td>60</td>
<td>E0</td>
<td>01</td>
<td>D0</td>
<td>18</td>
<td>29</td>
<td>0F</td>
<td>48</td>
<td>20</td>
<td>04</td>
<td>12</td>
</tr>
<tr>
<td>13A0</td>
<td>F0</td>
<td>80</td>
<td>AC</td>
<td>13</td>
<td>68</td>
<td>00</td>
<td>AC</td>
<td>13</td>
<td>28</td>
<td>2D</td>
<td>13</td>
</tr>
<tr>
<td>13B0</td>
<td>A8</td>
<td>03</td>
<td>16</td>
<td>1E</td>
<td>01</td>
<td>12</td>
<td>EE</td>
<td>10</td>
<td>F9</td>
<td>1D</td>
<td>01</td>
</tr>
<tr>
<td>13C0</td>
<td>68</td>
<td>C9</td>
<td>7F</td>
<td>D0</td>
<td>04</td>
<td>28</td>
<td>00</td>
<td>11</td>
<td>68</td>
<td>C9</td>
<td>51</td>
</tr>
<tr>
<td>13D0</td>
<td>60</td>
<td>20</td>
<td>E8</td>
<td>18</td>
<td>60</td>
<td>30</td>
<td>E9</td>
<td>30</td>
<td>90</td>
<td>0F</td>
<td>C9</td>
</tr>
<tr>
<td>13E0</td>
<td>C9</td>
<td>18</td>
<td>E0</td>
<td>05</td>
<td>30</td>
<td>C9</td>
<td>0A</td>
<td>00</td>
<td>03</td>
<td>A9</td>
<td>FF</td>
</tr>
<tr>
<td>13F0</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
Appendix D4:
Print Utilities

APPENDIX D4: PRINT UTILITIES
SEE CHAPTER 7 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUT

BUMPING $1400-$154F

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1400</td>
<td>FF</td>
<td>FF</td>
<td>00</td>
<td>20</td>
<td>00</td>
<td>00</td>
<td>0C</td>
<td>15</td>
<td>A9</td>
<td>FF</td>
<td>0D</td>
<td>01</td>
<td>14</td>
<td>60</td>
<td>A9</td>
</tr>
<tr>
<td>1410</td>
<td>8D</td>
<td>01</td>
<td>14</td>
<td>60</td>
<td>A9</td>
<td>FF</td>
<td>0D</td>
<td>00</td>
<td>14</td>
<td>60</td>
<td>A9</td>
<td>00</td>
<td>0D</td>
<td>00</td>
<td>14</td>
</tr>
<tr>
<td>1420</td>
<td>A9</td>
<td>FF</td>
<td>0D</td>
<td>02</td>
<td>14</td>
<td>60</td>
<td>A9</td>
<td>00</td>
<td>0D</td>
<td>02</td>
<td>14</td>
<td>60</td>
<td>20</td>
<td>00</td>
<td>14</td>
</tr>
<tr>
<td>1430</td>
<td>14</td>
<td>14</td>
<td>20</td>
<td>20</td>
<td>14</td>
<td>60</td>
<td>20</td>
<td>0E</td>
<td>14</td>
<td>20</td>
<td>1A</td>
<td>14</td>
<td>20</td>
<td>25</td>
<td>14</td>
</tr>
<tr>
<td>1440</td>
<td>C9</td>
<td>00</td>
<td>F0</td>
<td>24</td>
<td>6D</td>
<td>03</td>
<td>14</td>
<td>AD</td>
<td>01</td>
<td>14</td>
<td>F0</td>
<td>06</td>
<td>AD</td>
<td>03</td>
<td>14</td>
</tr>
<tr>
<td>1450</td>
<td>69</td>
<td>14</td>
<td>AD</td>
<td>00</td>
<td>14</td>
<td>F0</td>
<td>06</td>
<td>AD</td>
<td>03</td>
<td>14</td>
<td>20</td>
<td>6C</td>
<td>14</td>
<td>AD</td>
<td>02</td>
</tr>
<tr>
<td>1460</td>
<td>F0</td>
<td>06</td>
<td>AD</td>
<td>03</td>
<td>14</td>
<td>20</td>
<td>6F</td>
<td>14</td>
<td>60</td>
<td>6C</td>
<td>0A</td>
<td>18</td>
<td>6C</td>
<td>0C</td>
<td>18</td>
</tr>
<tr>
<td>1470</td>
<td>0E</td>
<td>10</td>
<td>A9</td>
<td>0D</td>
<td>20</td>
<td>4B</td>
<td>14</td>
<td>A9</td>
<td>0A</td>
<td>20</td>
<td>4B</td>
<td>14</td>
<td>60</td>
<td>A9</td>
<td>20</td>
</tr>
<tr>
<td>1480</td>
<td>4B</td>
<td>14</td>
<td>60</td>
<td>48</td>
<td>4A</td>
<td>4A</td>
<td>4A</td>
<td>20</td>
<td>06</td>
<td>11</td>
<td>28</td>
<td>40</td>
<td>14</td>
<td>68</td>
<td>20</td>
</tr>
<tr>
<td>1490</td>
<td>06</td>
<td>11</td>
<td>28</td>
<td>40</td>
<td>14</td>
<td>68</td>
<td>A9</td>
<td>20</td>
<td>0E</td>
<td>04</td>
<td>14</td>
<td>49</td>
<td>AE</td>
<td>04</td>
<td>14</td>
</tr>
<tr>
<td>14A0</td>
<td>0A</td>
<td>CE</td>
<td>04</td>
<td>14</td>
<td>20</td>
<td>48</td>
<td>14</td>
<td>60</td>
<td>18</td>
<td>50</td>
<td>F0</td>
<td>06</td>
<td>60</td>
<td>60</td>
<td>CE</td>
</tr>
<tr>
<td>14B0</td>
<td>AE</td>
<td>04</td>
<td>14</td>
<td>F0</td>
<td>05</td>
<td>CE</td>
<td>04</td>
<td>14</td>
<td>20</td>
<td>72</td>
<td>14</td>
<td>18</td>
<td>90</td>
<td>F2</td>
<td>60</td>
</tr>
<tr>
<td>14C0</td>
<td>05</td>
<td>14</td>
<td>B5</td>
<td>01</td>
<td>48</td>
<td>B5</td>
<td>00</td>
<td>48</td>
<td>AC</td>
<td>05</td>
<td>14</td>
<td>A1</td>
<td>00</td>
<td>C9</td>
<td>FF</td>
</tr>
<tr>
<td>14D0</td>
<td>0C</td>
<td>F6</td>
<td>00</td>
<td>00</td>
<td>02</td>
<td>F6</td>
<td>01</td>
<td>20</td>
<td>4B</td>
<td>14</td>
<td>18</td>
<td>90</td>
<td>EB</td>
<td>68</td>
<td>95</td>
</tr>
<tr>
<td>14E0</td>
<td>68</td>
<td>95</td>
<td>01</td>
<td>60</td>
<td>68</td>
<td>0A</td>
<td>68</td>
<td>0A</td>
<td>20</td>
<td>12</td>
<td>15</td>
<td>8C</td>
<td>05</td>
<td>12</td>
<td>8C</td>
</tr>
<tr>
<td>14F0</td>
<td>12</td>
<td>20</td>
<td>0E</td>
<td>13</td>
<td>20</td>
<td>0D</td>
<td>13</td>
<td>20</td>
<td>S4</td>
<td>12</td>
<td>C9</td>
<td>FF</td>
<td>F0</td>
<td>05</td>
<td>28</td>
</tr>
<tr>
<td>1500</td>
<td>14</td>
<td>18</td>
<td>90</td>
<td>F0</td>
<td>AE</td>
<td>05</td>
<td>12</td>
<td>AC</td>
<td>06</td>
<td>12</td>
<td>20</td>
<td>2B</td>
<td>15</td>
<td>98</td>
<td>49</td>
</tr>
<tr>
<td>1510</td>
<td>48</td>
<td>60</td>
<td>68</td>
<td>0D</td>
<td>06</td>
<td>14</td>
<td>68</td>
<td>80</td>
<td>07</td>
<td>14</td>
<td>AD</td>
<td>06</td>
<td>12</td>
<td>48</td>
<td>AD</td>
</tr>
<tr>
<td>1520</td>
<td>12</td>
<td>48</td>
<td>AD</td>
<td>07</td>
<td>14</td>
<td>48</td>
<td>AD</td>
<td>06</td>
<td>14</td>
<td>48</td>
<td>60</td>
<td>68</td>
<td>0D</td>
<td>06</td>
<td>14</td>
</tr>
<tr>
<td>1530</td>
<td>6D</td>
<td>07</td>
<td>14</td>
<td>68</td>
<td>80</td>
<td>05</td>
<td>12</td>
<td>68</td>
<td>80</td>
<td>05</td>
<td>12</td>
<td>AD</td>
<td>07</td>
<td>14</td>
<td>48</td>
</tr>
<tr>
<td>1540</td>
<td>06</td>
<td>14</td>
<td>48</td>
<td>60</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

392 BEYOND GAMES
Appendix D5:
Two Hexdump Tools

APPENDIX D5: TWO HEXDUMP TOOLS

SEE CHAPTER 8 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

DUMPING $1550-$17AF

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

1550 00 04 50 15 AF 17 00 20 09 14 AD 61 15 80 50 15
1560 AD 05 12 20 FD 80 05 12 20 72 14 20 72 14 20 A1
1570 15 20 72 14 20 7D 14 20 9A 15 20 0D 13 AD 65 12
1580 29 07 D0 F0 20 72 14 AD 05 12 20 0F D0 03 20 72
1590 14 CE 50 15 D0 D0 0E 14 60 20 94 12 20 63 14
15A0 60 AD 06 12 20 63 14 AD 05 12 20 63 14 60 20 C9
15B0 15 20 E9 15 20 A9 17 20 14 14 20 CB 16 20 42 17
15C0 10 FB 20 72 14 20 1A 14 60 20 0A 11 20 08 14 20
15D0 E4 14 7F 00 50 50 50 49 4E 54 49 4E 47 20 48 45 58
15E0 44 55 4D 50 00 0A 0A FF 60 20 08 14 20 44 14 7F
15F0 0D 0A 53 45 54 20 53 54 41 52 54 49 4E 47 20 41
1600 44 44 52 45 53 53 20 41 4E 44 20 50 52 45 53 53
1610 20 22 51 22 2C FF 20 07 12 20 0A 0D 12 0D 53 15 90 24
1620 E4 14 7F 0D 0A 53 45 54 20 45 4E 44 20 41 44 44
1630 52 45 53 53 20 41 4E 44 20 50 52 45 53 53 20 22
1640 51 22 2C FF 20 07 12 30 AD 06 12 0D 53 15 90 24
1650 D8 08 AD 05 12 CD 52 15 90 1A AD 06 12 AD 55 15
1660 AD 05 12 0D 54 15 60 AD 06 12 0D 53 15 AD 05 12
1670 8D 52 15 00 20 E4 14 7F 0D 0A 0A 0A 20 45 52 52
1680 4F 52 21 21 20 45 4E 44 20 41 44 44 52 45 53
1690 53 20 4C 45 53 53 20 54 48 41 4E 20 53 54 41 52
16A0 54 20 41 44 44 52 45 53 53 2C 20 57 48 49 43 48
16B0 20 49 53 20 FF 20 BB 16 4C 1C 16 A9 24 20 40 14
16C0 AD 53 15 20 03 14 AD 52 15 20 03 14 60 A9 24 20
16D0 48 14 AD 55 15 20 03 14 AD 54 15 20 03 14 60 20
16E0 BB 16 A9 20 20 40 14 20 CD 16 60 20 E4 14 7F 0D
16F0 0A 0A 44 55 4D 50 49 4E 47 20 FF 20 D6 16 20 72
1700 14 20 E4 14 7F 0A 0A 20 20 20 20 20 20 20 20 30
1710 20 20 31 20 20 32 20 20 33 20 20 34 20 20 35 20
Appendix D6:
Table-Driven Disassembler (Top Level and Utility Subroutines)

APPENDIX D6: TABLE-DRIVEN DISASSEMBLER (TOP LEVEL AND UTILITY SUBROUTINES)

SEE CHAPTER 9 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

DUMPING $1980-$1A3F

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

1900 05 00 00 5A 40 1A FF 04 10 20 00 14 AD 00 19 BD
1910 01 19 A9 FF 0D 54 15 60 55 15 20 72 14 20 7D 19
1920 CE 01 19 DD F8 60 20 1A 14 20 08 14 20 E4 14 7F
1930 BD 0A 20 20 20 20 20 50 52 49 4E 54 49 4E 47 20
1940 44 49 53 41 53 53 45 4D 42 4C 45 52 2E 00 0A FF
1950 20 E9 15 20 14 14 20 E4 14 7F BD 0A 44 49 53 41
1960 53 53 45 4D 42 4C 49 4E 47 20 FF 20 DF 16 20 A0
1970 17 20 72 14 20 7D 19 10 FB 20 1A 14 60 20 94 12
1980 48 20 92 19 20 7D 14 60 20 AF 19 20 01 1A 20 83
1990 17 60 62 03 8E 02 19 AA BD 00 1C AA BD 50 1B 8C
19A0 03 19 20 40 14 AE 03 19 EA CE 02 19 DD EE 60 AA
19B0 BD 00 1D AA 20 8D 19 60 BD 18 1B BD 04 19 EB BD
19C0 1B 18 00 05 19 EC 04 19 20 0D 13 20 9A 15 60 20
19D0 0D 13 20 94 12 48 20 0D 13 20 9A 15 60 20 83 14
19E0 60 A9 28 D0 02 A9 29 20 40 14 60 A9 2C 20 40 14
19F0 A5 58 20 40 14 60 A9 2C 20 40 14 A5 59 20 40 14
1A00 60 00 07 19 BE 06 19 CA 30 06 20 1A 13 CA 10 FA
1A10 08 00 39 AD 08 18 ES 04 ED 07 19 28 AA 20 95 14
1A20 28 A1 15 20 7D 14 20 9A 15 20 0D 13 CE 06 19 10
1A30 F2 20 1A 13 20 72 14 60 00 00 00 00 00 00 00 00
Appendix D7:
Table-Driven Disassembler
(Addressing Mode Subroutines)

See Chapter 9 of Beyond Games: System Software for Your 6502 Personal Computer

DUMPING $1A40-$1B4F

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

1A40  20 CF 19 A2 02 A9 04 60 20 40 1A 20 EB 19 A2 02
1A50  A9 06 60 20 40 1A 20 FE 19 A2 02 A9 06 60 A9 41
1A60  20 40 1A 00 A9 01 60 A2 00 A9 00 60 A9 29 29
1A70  40 1A A9 24 20 40 1A 20 CB 19 A2 01 A9 04 60 20
1A80  E1 19 20 40 1A 20 ES 19 A9 05 A2 02 60 20 E1 19
1A90  20 EB 1A 20 ES 19 A2 01 A9 08 60 20 E1 19 20 DB
1AA0  1A 20 ES 19 20 F6 19 A2 01 A9 08 60 20 DB 19 20
1AB0  12 15 20 94 12 49 20 0D 13 68 C9 00 10 03 CE 06
1AC0  12 06 0A 18 6D 05 12 90 03 EE 06 12 6D 05 12 28
1AD0  20 A1 15 20 2B 15 A2 01 A9 04 60 A9 00 20 03 14
1AE0  20 CB 19 A2 01 A9 04 60 20 DB 1A 20 EB 19 A2 01
1AF0  A9 06 60 20 DB 1A 20 FE 19 A2 01 A9 06 60 60 60
1B00  68 68 2B 03 17 30 00 20 94 12 CF FF F0 06 20 4B
1B10  14 18 98 EE 20 72 1A 20 83 17 60 60 1A EE 1A 6D
1B20  1A DB 1A ES 1A FE 1A 40 1A 40 1A 53 1A 68 1A AC
1B30  1A BD 1A BD 1A BD 1A FE 1A 00 00 00 00 00 00 00 00
1B40  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

396 Beyond Games
## Appendix D8:
Table-Driven Disassembler (Tables)

**APPENDIX D8:**
**TABLE-DRIVEN DISASSEMBLER (TABLES)**

SEE CHAPTER 9 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

**DUMPING $1B50-$1DFF**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1B50</td>
<td>7</td>
<td>F</td>
<td>42</td>
<td>41</td>
<td>44</td>
<td>41</td>
<td>44</td>
<td>43</td>
<td>41</td>
<td>4E</td>
<td>44</td>
<td>41</td>
<td>53</td>
<td>4C</td>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td>1B60</td>
<td>42</td>
<td>43</td>
<td>53</td>
<td>42</td>
<td>45</td>
<td>51</td>
<td>42</td>
<td>49</td>
<td>54</td>
<td>42</td>
<td>4D</td>
<td>49</td>
<td>42</td>
<td>4E</td>
<td>45</td>
<td>42</td>
</tr>
<tr>
<td>1B70</td>
<td>50</td>
<td>4C</td>
<td>42</td>
<td>52</td>
<td>48</td>
<td>42</td>
<td>56</td>
<td>43</td>
<td>42</td>
<td>56</td>
<td>53</td>
<td>43</td>
<td>4C</td>
<td>43</td>
<td>43</td>
<td>4C</td>
</tr>
<tr>
<td>1B80</td>
<td>44</td>
<td>43</td>
<td>4C</td>
<td>49</td>
<td>43</td>
<td>4C</td>
<td>56</td>
<td>43</td>
<td>4D</td>
<td>50</td>
<td>43</td>
<td>50</td>
<td>58</td>
<td>43</td>
<td>50</td>
<td>53</td>
</tr>
<tr>
<td>1B90</td>
<td>44</td>
<td>45</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>58</td>
<td>44</td>
<td>45</td>
<td>59</td>
<td>45</td>
<td>4F</td>
<td>52</td>
<td>49</td>
<td>4E</td>
<td>43</td>
<td>49</td>
</tr>
<tr>
<td>1BA0</td>
<td>4E</td>
<td>58</td>
<td>49</td>
<td>4E</td>
<td>59</td>
<td>4A</td>
<td>4D</td>
<td>50</td>
<td>4A</td>
<td>53</td>
<td>52</td>
<td>4C</td>
<td>44</td>
<td>41</td>
<td>4C</td>
<td>44</td>
</tr>
<tr>
<td>1BB0</td>
<td>58</td>
<td>4C</td>
<td>44</td>
<td>59</td>
<td>4C</td>
<td>53</td>
<td>52</td>
<td>4E</td>
<td>50</td>
<td>4F</td>
<td>52</td>
<td>41</td>
<td>50</td>
<td>48</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>1BC0</td>
<td>50</td>
<td>48</td>
<td>50</td>
<td>4C</td>
<td>41</td>
<td>50</td>
<td>4C</td>
<td>50</td>
<td>52</td>
<td>4F</td>
<td>4C</td>
<td>52</td>
<td>4F</td>
<td>52</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>1BD0</td>
<td>54</td>
<td>49</td>
<td>52</td>
<td>54</td>
<td>53</td>
<td>53</td>
<td>52</td>
<td>43</td>
<td>53</td>
<td>46</td>
<td>53</td>
<td>45</td>
<td>44</td>
<td>45</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>1BE0</td>
<td>49</td>
<td>53</td>
<td>54</td>
<td>41</td>
<td>53</td>
<td>54</td>
<td>50</td>
<td>53</td>
<td>54</td>
<td>53</td>
<td>54</td>
<td>41</td>
<td>50</td>
<td>54</td>
<td>41</td>
<td>59</td>
</tr>
<tr>
<td>1BF0</td>
<td>54</td>
<td>53</td>
<td>58</td>
<td>54</td>
<td>58</td>
<td>41</td>
<td>54</td>
<td>58</td>
<td>53</td>
<td>54</td>
<td>59</td>
<td>41</td>
<td>54</td>
<td>45</td>
<td>59</td>
<td>FF</td>
</tr>
<tr>
<td>1C00</td>
<td>22</td>
<td>EA</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>6A</td>
<td>0A</td>
<td>01</td>
<td>70</td>
<td>6A</td>
<td>0A</td>
<td>01</td>
<td>6A</td>
<td>0A</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>1C10</td>
<td>1F</td>
<td>6A</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>6A</td>
<td>0A</td>
<td>01</td>
<td>28</td>
<td>6A</td>
<td>01</td>
<td>01</td>
<td>6A</td>
<td>0A</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>1C20</td>
<td>58</td>
<td>07</td>
<td>01</td>
<td>01</td>
<td>16</td>
<td>07</td>
<td>79</td>
<td>01</td>
<td>76</td>
<td>07</td>
<td>79</td>
<td>01</td>
<td>16</td>
<td>07</td>
<td>79</td>
<td>01</td>
</tr>
<tr>
<td>1C30</td>
<td>19</td>
<td>07</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>07</td>
<td>79</td>
<td>01</td>
<td>88</td>
<td>07</td>
<td>01</td>
<td>01</td>
<td>07</td>
<td>79</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>1C40</td>
<td>07</td>
<td>09</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>49</td>
<td>64</td>
<td>01</td>
<td>60</td>
<td>49</td>
<td>64</td>
<td>01</td>
<td>55</td>
<td>49</td>
<td>64</td>
<td>01</td>
</tr>
<tr>
<td>1C50</td>
<td>25</td>
<td>49</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>49</td>
<td>64</td>
<td>01</td>
<td>31</td>
<td>49</td>
<td>01</td>
<td>01</td>
<td>49</td>
<td>64</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>1C60</td>
<td>42</td>
<td>04</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>04</td>
<td>7C</td>
<td>01</td>
<td>73</td>
<td>04</td>
<td>7C</td>
<td>01</td>
<td>55</td>
<td>04</td>
<td>7C</td>
<td>01</td>
</tr>
<tr>
<td>1C70</td>
<td>28</td>
<td>04</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>04</td>
<td>7C</td>
<td>01</td>
<td>8E</td>
<td>04</td>
<td>01</td>
<td>01</td>
<td>04</td>
<td>7C</td>
<td>AC</td>
<td></td>
</tr>
<tr>
<td>1C80</td>
<td>01</td>
<td>91</td>
<td>01</td>
<td>01</td>
<td>57</td>
<td>91</td>
<td>94</td>
<td>01</td>
<td>45</td>
<td>91</td>
<td>93</td>
<td>01</td>
<td>91</td>
<td>94</td>
<td>91</td>
<td>94</td>
</tr>
<tr>
<td>1C90</td>
<td>00</td>
<td>91</td>
<td>01</td>
<td>01</td>
<td>97</td>
<td>91</td>
<td>94</td>
<td>01</td>
<td>93</td>
<td>91</td>
<td>93</td>
<td>01</td>
<td>91</td>
<td>91</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>1CA0</td>
<td>01</td>
<td>58</td>
<td>0E</td>
<td>01</td>
<td>61</td>
<td>58</td>
<td>5E</td>
<td>01</td>
<td>60</td>
<td>58</td>
<td>9A</td>
<td>01</td>
<td>61</td>
<td>58</td>
<td>5E</td>
<td>01</td>
</tr>
<tr>
<td>1CB0</td>
<td>01</td>
<td>58</td>
<td>0E</td>
<td>01</td>
<td>61</td>
<td>58</td>
<td>5E</td>
<td>01</td>
<td>34</td>
<td>58</td>
<td>9E</td>
<td>01</td>
<td>61</td>
<td>58</td>
<td>5E</td>
<td>01</td>
</tr>
<tr>
<td>1CC0</td>
<td>3D</td>
<td>37</td>
<td>01</td>
<td>01</td>
<td>3D</td>
<td>37</td>
<td>40</td>
<td>01</td>
<td>52</td>
<td>37</td>
<td>43</td>
<td>01</td>
<td>3D</td>
<td>37</td>
<td>40</td>
<td>01</td>
</tr>
<tr>
<td>1CD0</td>
<td>2C</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>37</td>
<td>40</td>
<td>01</td>
<td>2C</td>
<td>37</td>
<td>01</td>
<td>01</td>
<td>37</td>
<td>40</td>
<td>01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1CE0</td>
<td>3A</td>
<td>05</td>
<td>01</td>
<td>01</td>
<td>3A</td>
<td>05</td>
<td>4C</td>
<td>01</td>
<td>01</td>
<td>4F</td>
<td>05</td>
<td>67</td>
<td>01</td>
<td>3A</td>
<td>05</td>
<td>4C</td>
</tr>
<tr>
<td>1CF0</td>
<td>13</td>
<td>05</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>05</td>
<td>4C</td>
<td>01</td>
<td>01</td>
<td>05</td>
<td>01</td>
<td>01</td>
<td>05</td>
<td>4C</td>
<td>01</td>
</tr>
<tr>
<td>1D000</td>
<td>12</td>
<td>16</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>12</td>
<td>04</td>
<td>02</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>1D10</td>
<td>14</td>
<td>18</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>12</td>
<td>10</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>16</td>
<td>16</td>
<td>00</td>
</tr>
</tbody>
</table>

397
| 1D20 | 0C 16 00 00 06 06 06 00 12 04 02 00 6C 6C 6C 00 |
| 1D30 | 14 18 00 00 00 00 00 12 10 00 00 00 0E 0E 00 |
| 1D40 | 12 16 00 00 00 06 06 00 12 0C 02 00 6C 6C 6C 00 |
| 1D50 | 14 18 00 00 00 00 00 12 10 00 00 00 0E 0E 00 |
| 1D60 | 12 16 00 00 00 06 06 00 12 04 02 00 1A 6C 6C 00 |
| 1D70 | 14 18 00 00 00 00 00 12 10 00 00 00 0E 0E 1C |
| 1D80 | 00 16 00 00 00 06 06 00 12 00 12 00 6C 6C 6C 00 |
| 1D90 | 14 18 00 00 00 00 00 0A 00 12 10 12 00 00 0E 00 00 |
| 1DA0 | 04 16 04 00 06 06 05 00 12 04 12 00 6C 6C 6C 00 |
| 1DB0 | 14 18 00 00 00 00 00 0A 00 14 10 12 00 0E 0E 10 00 |
| 1DC0 | 04 16 00 00 00 06 06 00 12 04 12 00 6C 6C 6C 00 |
| 1DD0 | 14 18 00 00 00 00 00 0A 00 12 10 00 00 00 0E 0E 00 |
| 1DE0 | 04 16 00 00 00 06 06 00 12 04 12 00 6C 6C 6C 00 |
| 1DF0 | 14 18 00 00 00 00 00 00 12 10 00 00 00 0E 0E 00 |

398 BEYOND GAMES
Appendix D9:
Move Utilities

APPENDIX D9: MOVE UTILITIES

SEE CHAPTER 10 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

DUMPING $17B0-$18FF

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>17B0</td>
<td>C7 00 39 04 20 B0 14 20 E4 14 7F 00 0A 20 20 20</td>
<td>17C0</td>
<td>20 20 4D 4F 56 45 20 54 4F 4F 4C 2E 00 0A 0A FF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17D0</td>
<td>20 20 15 20 B9 18 AE 55 15 3B AD 54 15 ED 52 15</td>
<td>17E0</td>
<td>80 B0 17 B0 02 CA 38 BA ED 53 15 BD B1 17 B0 03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17F0</td>
<td>A9 00 00 00 03 B9 00 00 48 08 10 F9 30 AD 53 15</td>
<td>1800</td>
<td>CD B3 17 90 40 D0 18 AD 52 15 CD B2 17 90 35 D0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1810</td>
<td>0E AD 00 00 00 00 00 B0 C8 C0 04 D0 F7 A9 FF 60 20</td>
<td>1820</td>
<td>A4 18 AD 00 AE B1 17 F0 0E B1 00 00 02 CB D0 F9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1830</td>
<td>F6 B1 06 03 CA 00 F2 08 CB 00 00 02 CC B0 17</td>
<td>1840</td>
<td>D0 F6 4C 11 18 AD B1 17 F0 48 AC B1 17 AD B0 17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1850</td>
<td>3B E9 FF B0 01 00 0A 04 03 0A 18 6D 52 15 05 00</td>
<td>1860</td>
<td>90 01 CB 99 6D 53 15 05 01 08 18 6D B2 17 05 02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1870</td>
<td>S0 02 E6 03 AS 03 6D B3 17 05 04 AE B1 17 A0 FF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1880</td>
<td>B1 00 91 02 00 00 B9 B1 00 01 02 CB 01 C5 03 CA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1890</td>
<td>00 EC 20 A4 18 AC B0 17 B1 00 01 02 08 C0 FF 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18A0</td>
<td>F7 4C 11 18 AD 52 15 05 00 AD 53 15 05 01 AD B2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18B0</td>
<td>17 0B 02 AD B3 17 65 03 60 20 00 14 20 E4 14 7F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18C0</td>
<td>00 0A 53 45 54 20 44 45 53 54 49 4E 41 54 49 4F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18D0</td>
<td>4E 20 41 4E 44 28 50 52 45 53 53 20 51 2E FF 28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18E0</td>
<td>07 12 AD 05 12 00 B2 17 AD 05 12 00 B3 17 60 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18F0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

399
Appendix D10:
Simple Text Editor

APPENDIX D10:  A SIMPLE TEXT EDITOR

SEE CHAPTER 11 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTERS
BY KEN SKIERN

DUMPING $1E00-$1FFF

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

1E00  FF 01 20 0F 1E 20 37 1E 20 C9 1F 1B 18 10 90 F6 20
1E10  0B 14 20 E4 14 7F 0D 0A 0A 53 45 54 20 55 50 20
1E20  45 44 49 54 20 42 55 46 46 46 52 2E 0D 0A 0A FF
1E30  20 E9 15 20 A0 17 60 20 C4 11 20 2B 11 AE 03 10
1E40  A0 03 20 13 11 20 2B 11 20 76 11 20 C4 11 20 SE
1E50  1E 20 D3 11 20 76 11 20 69 1E 20 D3 11 60 20 12
1E60  15 AD 03 18 4A AA CA CA 20 1A 13 CA 10 FA AD 03
1E70  10 BD 00 1E 20 94 12 20 9B 11 20 7F 11 20 BD 13
1E80  CE 00 1E 10 EF 20 29 15 60 AD 03 10 4A E9 02 20
1E90  B1 11 AD 01 1E C9 01 D0 05 A9 49 1B 09 02 A9 4F
1EA0  20 B8 11 A9 02 20 81 11 AD 07 10 20 98 11 A9 02
1EB0  20 B1 11 AD 06 12 20 A3 11 AD 05 12 20 A3 11 60
1EC0  06 03 3E 3C 10 7F 51 00 20 E0 12 CD C6 1E D0 17
1ED0  4B 20 E0 12 CD C6 1E D0 00 68 68 60 60 CD C7 1E
1EE0  68 20 E7 1E AD C7 1E CD C1 1E D0 00 CD 01 1E 10
1EF0  05 A9 01 8D 01 1E 60 CD C2 1E D0 00 79 1F 6F
1F00  CD C3 1E D0 04 20 BE 1F 68 CD C5 1E D0 04 20 DD
1F10  1F 60 CD C4 1E D0 04 20 C5 1F 68 CD C0 1E D0 04
1F20  20 B4 1F 60 AE 01 1E F0 04 20 34 1F 60 20 2D 13
1F30  20 B3 17 60 48 20 12 15 AD 53 15 48 AD 52 15 48
1F40  AD 55 15 48 AD 54 15 48 20 67 16 20 83 17 30 11
1F50  20 E2 18 AD 54 15 00 04 CE 55 15 CE 54 15 20 D6
1F60  17 ED BD 54 15 60 BD 55 15 60 BD 52 15 60 BD 53
1F70  15 20 28 15 60 20 2D 1F 60 20 94 12 C9 FF F0 04
1F80  20 BD 17 60 AD 9F FF 60 38 AD 53 15 CD 06 12 90 0C
1F90  D0 10 AD 52 15 CD 05 12 F0 17 D0 06 20 1A 13 99
1FA0  00 60 AD 52 15 BD 05 12 AD 53 15 BD 06 12 A9 00
1FB0  60 99 FF 60 20 AD 17 AD 9F FF 20 2D 13 20 83 17 10
1FC0  F6 20 A0 17 60 20 A0 17 20 14 14 28 94 12 C9 FF
1FD0  F0 00 20 40 14 20 BD 17 10 F1 4C 1A 14 20 12 15
1FE0  AD 53 15 48 AD 52 15 48 20 E2 10 20 83 17 20 67
1FF0  16 20 D6 17 60 60 52 15 60 60 53 15 20 2B 15 60
```

400 BEYOND GAMES
Appendix D11:
Extending the Visible Monitor

APPENDIX D11:  EXTENDING THE VISIBLE MONITOR

SEE CHAPTER 12 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER.

DUMPING $10B0-$10FF

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

10B0  C9 50 D0 09 AD 00 14 49 FF 8D 00 14 60 C9 55 D0
10C0  09 AD 02 14 49 FF 8D 02 14 60 C9 48 D0 0D AD 00
10D0  14 D0 04 20 57 15 60 20 AC 15 60 C9 4D D0 04 20
10E0  B4 17 60 C9 3F D0 0D AD 00 14 D0 04 20 09 19 60
10F0  20 26 19 60 C9 54 D0 04 20 02 1E 60 60 00 00 00
Appendix E1:
Screen Utilities

APPENDIX E1
SCREEN UTILITIES

THE FOLLOWING DATA STATEMENTS CONTAIN DECIMAL OBJECT CODE AND CHECKSUMS FOR MEMORY FROM 4352 TO 4607 SUITABLE FOR LOADING WITH THE BASIC OBJECT CODE LOADER.

1000 DATA 4352, 32, 196, 17, 32, 43, 17, 174, 3, 4966
1001 DATA 4360, 16, 172, 4, 16, 32, 19, 17, 32, 4668
1002 DATA 4368, 211, 17, 96, 142, 42, 17, 152, 170, 5215
1003 DATA 4376, 173, 6, 16, 172, 42, 17, 145, 0, 4947
1004 DATA 4384, 136, 16, 251, 32, 118, 17, 202, 16, 5172
1005 DATA 4392, 239, 96, 25, 162, 0, 160, 0, 24, 5096
1006 DATA 4400, 144, 10, 173, 4, 16, 74, 160, 173, 6162
1007 DATA 4408, 3, 16, 74, 170, 56, 236, 3, 16, 4982
1008 DATA 4416, 144, 3, 174, 3, 16, 56, 204, 0, 5820
1009 DATA 4424, 16, 144, 3, 172, 4, 16, 173, 0, 4952
1010 DATA 4432, 16, 133, 0, 173, 1, 16, 133, 1, 4985
1011 DATA 4440, 8, 216, 138, 24, 101, 0, 144, 3, 5074
1012 DATA 4448, 230, 1, 24, 192, 0, 248, 11, 24, 5170
1013 DATA 4456, 109, 2, 16, 144, 2, 230, 1, 136, 5096
1014 DATA 4464, 208, 245, 133, 0, 40, 96, 173, 2, 5361
1015 DATA 4472, 16, 24, 144, 5, 32, 155, 17, 163, 5034
1016 DATA 4480, 1, 8, 216, 24, 101, 0, 144, 2, 4976
1017 DATA 4488, 230, 1, 133, 0, 56, 173, 5, 16, 5102
1018 DATA 4496, 197, 1, 176, 5, 173, 1, 16, 133, 5198
1019 DATA 4504, 1, 40, 96, 32, 17, 16, 160, 0, 4866
1020 DATA 4512, 145, 0, 96, 72, 74, 74, 74, 74, 5121
1021 DATA 4520, 32, 182, 17, 32, 124, 17, 104, 32, 5060
1022 DATA 4528, 182, 17, 32, 124, 17, 96, 8, 216, 5220
1023 DATA 4536, 41, 15, 281, 10, 48, 2, 105, 6, 4964
1024 DATA 4544, 185, 48, 40, 96, 104, 170, 104, 166, 5379
1025 DATA 4552, 165, 1, 72, 165, 0, 72, 152, 72, 5251
1026 DATA 4560, 138, 72, 96, 104, 170, 104, 168, 104, 5516
1027 DATA 4568, 133, 0, 104, 133, 1, 152, 72, 138, 5301
1028 DATA 4576, 72, 96, 0, 0, 0, 0, 0, 4744
Appendix E2:

Visible Monitor (Top Level and Display Subroutines)

THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM 4608 TO 4931
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.

1100 DATA   4608, 0, 12, 8, 8, 49, 177, 252, 8, 5106
1101 DATA   4616, 216, 32, 10, 16, 32, 227, 18, 24, 5201
1102 DATA   4624, 144, 246, 32, 186, 17, 32, 37, 18, 5346
1103 DATA   4632, 32, 32, 18, 32, 52, 18, 32, 175, 5683
1104 DATA   4640, 18, 32, 211, 17, 96, 162, 2, 160, 5338
1105 DATA   4648, 2, 32, 60, 17, 162, 25, 160, 3, 5109
1106 DATA   4656, 32, 19, 17, 96, 162, 13, 160, 2, 5157
1107 DATA   4664, 32, 60, 17, 160, 0, 140, 81, 18, 5172
1108 DATA   4672, 185, 82, 18, 32, 124, 17, 238, 81, 5449
1109 DATA   4680, 18, 172, 81, 18, 192, 10, 208, 240, 5619
1110 DATA   4688, 96, 10, 85, 32, 32, 88, 32, 32, 5075
1111 DATA   4696, 69, 32, 32, 80, 162, 2, 160, 3, 5266
1112 DATA   4704, 32, 60, 17, 173, 6, 18, 32, 163, 5205
1113 DATA   4712, 17, 173, 5, 18, 32, 163, 17, 32, 5169
1114 DATA   4720, 127, 17, 32, 148, 18, 72, 32, 163, 5329
1115 DATA   4728, 17, 32, 127, 17, 104, 32, 124, 17, 5198
1116 DATA   4736, 32, 127, 17, 162, 0, 189, 1, 18, 5282
1117 DATA   4744, 32, 163, 17, 32, 127, 17, 232, 224, 5588
1118 DATA   4752, 4, 208, 242, 96, 165, 2, 72, 166, 5707
1119 DATA   4760, 3, 173, 5, 18, 133, 2, 173, 6, 5273
1120 DATA   4768, 18, 133, 3, 180, 0, 177, 2, 168, 5423
1121 DATA   4776, 104, 133, 2, 134, 3, 152, 96, 162, 5562
1122 DATA   4784, 2, 160, 4, 32, 60, 17, 172, 0, 5231
1123 DATA   4792, 18, 56, 152, 7, 144, 5, 160, 0, 5374
1124 DATA   4800, 140, 0, 18, 185, 205, 18, 168, 173, 5707
1125 DATA   4808, 7, 16, 145, 0, 96, 3, 6, 8, 5089
1126 DATA 4816, 11, 14, 17, 20, 0, 0, 0, 0, 4870
1127 DATA 4824, 0, 0, 0, 0, 0, 0, 0, 0, 4824
1128 END
Appendix E3:
Visible Monitor (Update Subroutine)

APPENDIX E3
VISIBLE MONITOR (UPDATE SUBROUTINE)

THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM 4832 TO 5119
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.

1200 DATA 4832, 108, 8, 16, 32, 224, 18, 201, 62, 5501
1201 DATA 4840, 208, 16, 238, 0, 18, 173, 0, 18, 5511
1202 DATA 4848, 201, 7, 208, 5, 169, 0, 141, 0, 5579
1203 DATA 4856, 18, 96, 201, 60, 208, 11, 206, 0, 5656
1204 DATA 4864, 18, 16, 5, 169, 6, 141, 0, 18, 5237
1205 DATA 4872, 96, 201, 32, 208, 9, 238, 5, 18, 5679
1206 DATA 4880, 208, 3, 238, 6, 18, 96, 201, 13, 5663
1207 DATA 4888, 208, 12, 173, 5, 18, 206, 3, 206, 5721
1208 DATA 4896, 6, 18, 206, 5, 18, 96, 174, 0, 5419
1209 DATA 4904, 18, 224, 2, 208, 27, 168, 165, 0, 5716
1210 DATA 4912, 72, 165, 1, 173, 5, 18, 133, 0, 5460
1211 DATA 4920, 173, 6, 18, 133, 1, 152, 163, 0, 5563
1212 DATA 4928, 145, 0, 134, 1, 104, 133, 0, 96, 5541
1213 DATA 4936, 201, 71, 208, 35, 172, 3, 18, 174, 5819
1214 DATA 4944, 2, 18, 173, 4, 18, 72, 173, 1, 5405
1215 DATA 4952, 18, 40, 32, 100, 19, 0, 141, 1, 5319
1216 DATA 4960, 18, 142, 2, 18, 198, 3, 18, 104, 5405
1217 DATA 4968, 141, 4, 18, 96, 108, 5, 18, 72, 5405
1218 DATA 4976, 32, 213, 19, 49, 75, 158, 104, 152, 5787
1219 DATA 4984, 174, 0, 18, 208, 20, 162, 3, 24, 5593
1220 DATA 4992, 14, 5, 18, 46, 6, 18, 202, 16, 5317
1221 DATA 5000, 246, 152, 13, 5, 10, 141, 5, 18, 5506
1222 DATA 5008, 96, 224, 1, 206, 24, 41, 15, 72, 5689
1223 DATA 5016, 32, 148, 18, 10, 10, 18, 41, 5295
1224 DATA 5024, 249, 141, 172, 19, 104, 13, 172, 19, 5904
1225 DATA 5032, 32, 45, 19, 95, 16, 202, 202, 202, 5846
1226 DATA 5040, 168, 3, 24, 30, 1, 18, 136, 15, 5428
1227 DATA 5048, 249, 29, 1, 18, 157, 1, 18, 96, 5617
1228 DATA 5056, 104, 201, 127, 208, 4, 32, 0, 17, 5749
<table>
<thead>
<tr>
<th>1229 DATA</th>
<th>5054, 96, 201, 81, 208, 4, 104, 104, 40, 5602</th>
</tr>
</thead>
<tbody>
<tr>
<td>1230 DATA</td>
<td>5072, 96, 32, 16, 16, 96, 56, 233, 48, 5665</td>
</tr>
<tr>
<td>1231 DATA</td>
<td>5060, 144, 15, 201, 10, 144, 14, 233, 7, 5648</td>
</tr>
<tr>
<td>1232 DATA</td>
<td>5068, 201, 16, 176, 5, 56, 201, 10, 176, 5929</td>
</tr>
<tr>
<td>1233 DATA</td>
<td>5056, 3, 169, 255, 96, 162, 0, 96, 0, 5877</td>
</tr>
<tr>
<td>1234 DATA</td>
<td>5104, 0, 0, 0, 0, 0, 0, 5104</td>
</tr>
<tr>
<td>1235 DATA</td>
<td>5112, 0, 0, 0, 0, 0, 0, 0, 0, 5112</td>
</tr>
<tr>
<td>1236 END</td>
<td></td>
</tr>
</tbody>
</table>
Appendix E4: Print Utilities

APPENDIX E4 PRINT UTILITIES

THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKS FOR MEMORY FROM 5120 TO 5455
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.

1300 DATA 5120, 0, 255, 0, 0, 0, 0, 0, 0, 5375
1301 DATA 5120, 169, 255, 141, 1, 20, 96, 169, 0, 5979
1302 DATA 5136, 141, 1, 20, 96, 169, 255, 141, 0, 5959
1303 DATA 5144, 20, 96, 169, 0, 141, 0, 20, 96, 5606
1304 DATA 5152, 169, 255, 141, 2, 20, 96, 169, 0, 6004
1305 DATA 5160, 141, 2, 20, 96, 32, 0, 20, 32, 5511
1306 DATA 5168, 20, 26, 32, 32, 26, 96, 32, 14, 5434
1307 DATA 5176, 20, 32, 26, 28, 32, 30, 26, 96, 5460
1308 DATA 5184, 201, 0, 240, 35, 141, 3, 20, 173, 5988
1309 DATA 5192, 1, 20, 240, 6, 173, 3, 20, 32, 5887
1310 DATA 5200, 105, 20, 173, 0, 20, 240, 6, 173, 5937
1311 DATA 5208, 3, 20, 32, 109, 20, 173, 2, 20, 5586
1312 DATA 5216, 240, 6, 173, 3, 20, 32, 111, 20, 5821
1313 DATA 5224, 96, 109, 16, 18, 109, 12, 16, 109, 5699
1314 DATA 5232, 14, 16, 169, 13, 32, 64, 20, 169, 5729
1315 DATA 5240, 10, 32, 64, 20, 96, 169, 32, 32, 5695
1316 DATA 5248, 64, 20, 96, 72, 74, 74, 74, 74, 5796
1317 DATA 5256, 32, 102, 17, 32, 64, 20, 104, 32, 5739
1318 DATA 5264, 102, 17, 32, 64, 20, 96, 169, 32, 5876
1319 DATA 5272, 142, 4, 20, 72, 174, 4, 20, 240, 5948
1320 DATA 5280, 0, 206, 4, 20, 32, 64, 20, 104, 5748
1321 DATA 5288, 24, 144, 240, 104, 96, 142, 4, 20, 6062
1322 DATA 5296, 174, 4, 20, 240, 9, 206, 4, 20, 5973
1323 DATA 5304, 32, 114, 20, 24, 144, 242, 96, 142, 5118
1324 DATA 5312, 5, 20, 181, 1, 72, 181, 0, 72, 5644
1325 DATA 5320, 174, 5, 20, 161, 0, 201, 255, 240, 6376
1326 DATA 5328, 12, 246, 0, 208, 2, 246, 1, 32, 6975
1327 DATA 5336, 64, 20, 24, 144, 235, 104, 149, 0, 6076
1328 DATA 5344, 104, 149, 1, 96, 104, 170, 104, 163, 6240
1329 DATA 5362, 32, 18, 21, 142, 5, 18, 140, 6, 5734
1330 DATA 5360, 18, 32, 13, 19, 32, 13, 19, 32, 5538
1331 DATA 5368, 148, 18, 201, 255, 240, 6, 32, 64, 6332
1332 DATA 5376, 20, 24, 144, 240, 174, 5, 18, 172, 6173
1333 DATA 5384, 6, 18, 32, 43, 21, 152, 72, 138, 5866
1334 DATA 5392, 72, 96, 104, 141, 6, 20, 104, 141, 6076
1335 DATA 5400, 7, 20, 173, 6, 18, 72, 173, 5, 5874
1336 DATA 5408, 18, 72, 173, 7, 20, 104, 141, 5949
1337 DATA 5416, 20, 72, 96, 104, 141, 6, 20, 104, 5973
1338 DATA 5424, 141, 7, 20, 104, 141, 5, 18, 104, 5964
1339 DATA 5432, 141, 6, 18, 173, 7, 20, 72, 173, 6042
1340 DATA 5440, 6, 20, 72, 96, 0, 0, 0, 0, 0, 0, 0, 0, 5634
1341 DATA 5448, 0, 0, 0, 0, 0, 0, 0, 0, 5448
1342 END

410 BEYOND GAMES
Appendix E5: Two Hexdump Tools

The following data statements contain decimal object code and checksums for memory from 5456 to 6053 suitable for loading with the basic object code loader.

1400 DATA 5456, 0, 4, 0, 0, 255, 255, 0, 32, 6002
1401 DATA 5464, 8, 20, 173, 81, 21, 141, 80, 21, 6009
1402 DATA 5472, 173, 5, 18, 41, 248, 141, 5, 18, 6121
1403 DATA 5480, 32, 114, 20, 32, 114, 20, 32, 161, 6005
1404 DATA 5488, 21, 32, 114, 20, 32, 125, 20, 32, 5994
1405 DATA 5496, 154, 21, 32, 13, 19, 173, 5, 18, 5931
1406 DATA 5504, 41, 7, 208, 240, 32, 114, 20, 173, 6339
1407 DATA 5512, 5, 18, 41, 15, 208, 3, 32, 114, 5948
1408 DATA 5520, 20, 206, 80, 21, 208, 216, 32, 14, 6317
1409 DATA 5528, 20, 96, 32, 148, 18, 32, 131, 20, 6025
1410 DATA 5536, 96, 173, 6, 18, 32, 131, 20, 173, 6185
1411 DATA 5544, 5, 18, 32, 131, 20, 96, 32, 201, 6079
1412 DATA 5552, 21, 32, 233, 21, 32, 180, 23, 32, 6166
1413 DATA 5560, 20, 20, 32, 235, 22, 32, 66, 23, 6010
1414 DATA 5568, 16, 251, 32, 114, 20, 32, 25, 20, 6079
1415 DATA 5576, 96, 32, 0, 17, 32, 8, 20, 32, 5813
1416 DATA 5584, 228, 20, 127, 13, 80, 82, 73, 79, 6285
1417 DATA 5592, 84, 73, 78, 71, 32, 72, 69, 88, 6159
1418 DATA 5596, 69, 95, 77, 80, 13, 10, 10, 255, 6198
1419 DATA 5600, 96, 32, 8, 20, 32, 228, 20, 127, 6171
1420 DATA 5608, 13, 18, 83, 69, 84, 32, 63, 64, 6074
1421 DATA 5616, 65, 82, 84, 73, 78, 71, 32, 65, 6174
1422 DATA 5624, 69, 69, 82, 69, 83, 83, 32, 65, 6182
1423 DATA 5632, 78, 78, 78, 78, 78, 78, 78, 78, 6215
1424 DATA 5640, 32, 34, 81, 34, 46, 255, 32, 7, 6159
1425 DATA 5648, 18, 32, 103, 22, 32, 8, 20, 32, 5923
1426 DATA 5656, 228, 20, 127, 13, 10, 8, 9, 84, 6298
1427 DATA 5664, 32, 69, 78, 68, 32, 65, 68, 68, 6152
1428 DATA 5672, 82, 69, 83, 83, 32, 65, 78, 68, 6240
Appendix E6:
Table-Driven Disassembler (Top Level and Utility Subroutines)

THE FOLLOWING DATA STATEMENTS CONTAIN DECIMAL OBJECT CODE AND CHECKSUMS FOR MEMORY FROM 6400 TO 6719 SUITABLE FOR LOADING WITH THE BASIC OBJECT CODE LOADER.

| 1500 DATA | 6400, 5, 0, 0, 0, 0, 0, 0, 6405 |
| 1501 DATA | 6408, 16, 32, 8, 20, 173, 0, 25, 141, 6823 |
| 1502 DATA | 6416, 1, 25, 169, 255, 141, 84, 21, 141, 7253 |
| 1503 DATA | 6424, 65, 21, 32, 114, 20, 32, 125, 25, 6878 |
| 1504 DATA | 6432, 206, 1, 25, 200, 248, 96, 32, 26, 7274 |
| 1505 DATA | 6440, 20, 32, 8, 20, 32, 228, 20, 127, 6927 |
| 1506 DATA | 6448, 13, 10, 32, 32, 32, 32, 80, 6711 |
| 1507 DATA | 6456, 82, 73, 73, 84, 73, 78, 71, 32, 7027 |
| 1508 DATA | 6464, 68, 73, 83, 65, 83, 83, 69, 77, 7055 |
| 1509 DATA | 6472, 66, 75, 69, 82, 46, 13, 10, 255, 7093 |
| 1510 DATA | 6480, 32, 233, 21, 32, 20, 20, 32, 228, 7098 |
| 1511 DATA | 6488, 20, 127, 13, 10, 69, 73, 83, 65, 6947 |
| 1512 DATA | 6496, 93, 83, 69, 77, 66, 75, 73, 79, 7101 |
| 1513 DATA | 6504, 71, 32, 255, 32, 223, 22, 32, 160, 7331 |
| 1514 DATA | 6512, 23, 32, 114, 20, 32, 125, 25, 16, 6899 |
| 1515 DATA | 6520, 251, 32, 26, 20, 96, 32, 143, 18, 7143 |
| 1516 DATA | 6528, 72, 32, 146, 25, 32, 125, 20, 104, 7084 |
| 1517 DATA | 6536, 32, 175, 25, 32, 1, 25, 32, 131, 6930 |
| 1518 DATA | 6544, 23, 96, 162, 3, 142, 2, 25, 170, 7167 |
| 1519 DATA | 6552, 189, 0, 20, 170, 189, 0, 27, 142, 7377 |
| 1520 DATA | 6560, 3, 25, 32, 64, 20, 174, 3, 25, 6906 |
| 1521 DATA | 6568, 232, 200, 2, 25, 200, 232, 96, 170, 7745 |
| 1522 DATA | 6576, 189, 0, 29, 170, 32, 104, 25, 96, 7301 |
| 1523 DATA | 6584, 169, 27, 27, 141, 4, 25, 232, 169, 7418 |
| 1524 DATA | 6592, 27, 27, 141, 5, 25, 108, 4, 25, 6954 |
| 1525 DATA | 6600, 32, 13, 19, 32, 154, 21, 98, 32, 6999 |
1526 DATA 6608, 13, 19, 32, 149, 18, 72, 32, 13, 6355
1527 DATA 6616, 19, 32, 154, 21, 164, 32, 131, 20, 7129
1528 DATA 6624, 96, 163, 40, 208, 2, 169, 41, 32, 7381
1529 DATA 6632, 64, 20, 96, 169, 44, 32, 64, 20, 7141
1530 DATA 6640, 169, 99, 32, 64, 20, 96, 169, 44, 7322
1531 DATA 6648, 32, 64, 20, 159, 69, 32, 64, 20, 7138
1532 DATA 6656, 96, 141, 7, 25, 142, 6, 25, 202, 7300
1533 DATA 6664, 48, 6, 32, 25, 19, 202, 16, 250, 7263
1534 DATA 6672, 8, 216, 56, 173, 8, 25, 233, 4, 7395
1535 DATA 6680, 237, 7, 25, 40, 170, 32, 150, 20, 7361
1536 DATA 6688, 32, 151, 21, 32, 125, 20, 32, 154, 7265
1537 DATA 6696, 21, 32, 13, 19, 206, 6, 25, 16, 7034
1538 DATA 6704, 242, 32, 25, 19, 32, 114, 20, 96, 7285
1539 DATA 6712, 0, 0, 0, 0, 0, 0, 0, 0, 6712
1540 END
Appendix E7:
Table-Driven Disassembler
(Addressing Mode Subroutines)

THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM 6720 TO 6991
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.

1600 DATA 6720, 32, 207, 25, 162, 2, 169, 4, 96, 7417
1601 DATA 6728, 32, 64, 26, 32, 235, 25, 162, 2, 7306
1602 DATA 6736, 169, 6, 97, 32, 64, 26, 32, 246, 7407
1603 DATA 6744, 25, 162, 2, 169, 6, 96, 169, 65, 7438
1604 DATA 6752, 32, 64, 20, 162, 0, 169, 1, 96, 7295
1605 DATA 6760, 162, 0, 169, 0, 96, 169, 35, 32, 7423
1606 DATA 6768, 64, 20, 169, 36, 32, 64, 20, 32, 7205
1607 DATA 6776, 200, 25, 162, 1, 169, 4, 96, 32, 7465
1608 DATA 6784, 225, 25, 32, 64, 26, 32, 229, 25, 7442
1609 DATA 6792, 169, 6, 162, 2, 96, 32, 225, 25, 7509
1610 DATA 6800, 32, 232, 26, 32, 229, 25, 162, 1, 7539
1611 DATA 6808, 169, 8, 95, 32, 225, 25, 32, 219, 7614
1612 DATA 6816, 26, 32, 229, 25, 32, 246, 25, 162, 7593
1613 DATA 6824, 1, 169, 8, 96, 32, 13, 19, 32, 7194
1614 DATA 6832, 19, 21, 32, 148, 18, 72, 32, 13, 7186
1615 DATA 6840, 19, 104, 201, 0, 16, 3, 206, 5, 7395
1616 DATA 6848, 19, 8, 216, 24, 109, 5, 18, 144, 7390
1617 DATA 6856, 3, 238, 6, 18, 141, 5, 18, 40, 7325
1618 DATA 6864, 32, 151, 21, 32, 43, 21, 162, 1, 7337
1619 DATA 6872, 169, 4, 96, 169, 0, 32, 131, 20, 7493
1620 DATA 6880, 32, 206, 25, 162, 1, 169, 4, 96, 7569
1621 DATA 6888, 32, 213, 25, 32, 235, 25, 162, 1, 7620
1622 DATA 6896, 169, 6, 95, 32, 213, 26, 32, 246, 7722
1623 DATA 6904, 25, 162, 1, 169, 6, 96, 104, 104, 7571
1624 DATA 6912, 104, 104, 32, 131, 23, 48, 13, 32, 7399
1625 DATA 6920, 148, 18, 201, 255, 240, 6, 32, 64, 7084
1626 DATA 6928, 20, 24, 144, 238, 32, 114, 20, 32, 7552
1627 DATA 6936, 131, 23, 56, 104, 26, 94, 26, 109, 7545
1628 DATA 6944, 26, 219, 26, 232, 26, 243, 26, 64, 7506
1629 DATA 6952, 26, 72, 26, 83, 26, 104, 26, 172, 7467
1630 DATA 6960, 26, 141, 26, 155, 26, 127, 26, 254, 7741
1631 DATA 6968, 26, 0, 0, 0, 0, 0, 0, 6994
1632 DATA 6976, 0, 0, 0, 0, 0, 0, 6976
1633 DATA 6984, 0, 0, 0, 0, 0, 0, 6984
1634 END
Appendix E8:
Table-Driven Disassembler (Tables)

The following data statements contain decimal object code and checksums for memory from 6592 to 7679 suitable for loading with the basic object code loader.

1700 DATA 6592, 127, 65, 65, 65, 68, 69, 67, 65, 7583
1701 DATA 7000, 75, 55, 65, 83, 75, 66, 67, 65, 7570
1702 DATA 7008, 65, 67, 83, 66, 69, 81, 65, 73, 7579
1703 DATA 7016, 84, 65, 77, 73, 66, 78, 69, 65, 7555
1704 DATA 7024, 80, 76, 65, 82, 75, 66, 67, 67, 7562
1705 DATA 7032, 65, 65, 83, 67, 76, 67, 67, 76, 7620
1706 DATA 7040, 65, 67, 76, 73, 67, 76, 67, 67, 7620
1707 DATA 7048, 77, 90, 65, 70, 65, 76, 67, 67, 7576
1708 DATA 7056, 65, 69, 67, 68, 69, 69, 66, 69, 7622
1709 DATA 7064, 89, 69, 79, 82, 73, 78, 67, 73, 7674
1710 DATA 7072, 72, 65, 89, 73, 79, 69, 74, 77, 68, 7709
1711 DATA 7080, 74, 83, 82, 76, 69, 65, 76, 68, 7672
1712 DATA 7088, 88, 76, 68, 89, 76, 83, 82, 79, 7728
1713 DATA 7096, 79, 80, 79, 82, 65, 80, 72, 65, 7598
1714 DATA 7104, 80, 72, 80, 80, 76, 65, 80, 76, 7713
1715 DATA 7112, 80, 82, 79, 76, 82, 79, 82, 82, 7754
1716 DATA 7120, 84, 73, 82, 84, 83, 83, 89, 87, 7742
1717 DATA 7128, 83, 89, 87, 83, 83, 89, 87, 7749
1718 DATA 7135, 73, 83, 84, 65, 83, 84, 98, 83, 7779
1719 DATA 7144, 84, 99, 84, 65, 84, 56, 69, 7792
1720 DATA 7152, 84, 83, 88, 84, 88, 65, 84, 88, 7816
1721 DATA 7160, 83, 84, 89, 84, 69, 89, 89, 255, 7977
1722 DATA 7168, 34, 105, 1, 1, 105, 10, 1, 7428
1723 DATA 7176, 112, 105, 10, 1, 1, 105, 10, 1, 7523
1724 DATA 7184, 31, 105, 1, 1, 105, 10, 1, 7441
1725 DATA 7192, 43, 105, 1, 1, 105, 10, 1, 7461
1726 DATA 7200, 90, 7, 1, 1, 22, 7, 121, 1, 7448
1727 DATA 7208, 98, 7, 1, 22, 7, 121, 1, 7606
1728 DATA 7216, 25, 7, 1, 1, 7, 121, 1, 7390
Appendix E9:
Move Utilities

THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM 6054 TO 6399
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.

1600 DATA 6054, 0, 0, 0, 0, 32, 28, 32, 6156
1601 DATA 6072, 228, 20, 127, 13, 18, 32, 32, 32, 6566
1602 DATA 6098, 32, 32, 77, 79, 69, 32, 84, 6571
1603 DATA 6088, 79, 79, 76, 46, 13, 10, 10, 255, 6566
1604 DATA 6096, 32, 233, 21, 32, 185, 24, 174, 85, 6882
1605 DATA 6104, 21, 56, 173, 84, 21, 237, 82, 21, 6799
1606 DATA 6112, 141, 176, 23, 176, 2, 202, 56, 138, 7026
1607 DATA 6120, 237, 83, 21, 141, 177, 23, 176, 3, 6991
1608 DATA 6128, 169, 0, 96, 160, 3, 105, 0, 0, 6741
1609 DATA 6136, 72, 136, 16, 249, 56, 173, 83, 21, 6942
1610 DATA 6144, 205, 179, 23, 144, 64, 208, 24, 173, 7164
1611 DATA 6152, 82, 21, 205, 178, 23, 144, 54, 208, 7067
1612 DATA 6160, 14, 160, 0, 104, 153, 0, 8, 208, 6791
1613 DATA 6168, 192, 4, 208, 247, 159, 255, 96, 32, 7371
1614 DATA 6176, 164, 24, 160, 0, 174, 177, 23, 240, 7138
1615 DATA 6184, 14, 177, 0, 145, 2, 200, 208, 249, 7179
1616 DATA 6192, 230, 1, 230, 9, 202, 208, 242, 136, 7444
1617 DATA 6200, 200, 177, 0, 145, 2, 204, 176, 23, 7127
1618 DATA 6208, 200, 246, 76, 17, 24, 173, 177, 23, 7152
1619 DATA 6216, 248, 72, 172, 177, 23, 173, 176, 23, 7272
1620 DATA 6224, 56, 233, 255, 176, 1, 136, 170, 132, 7383
1621 DATA 6232, 3, 136, 24, 108, 62, 21, 133, 0, 6742
1622 DATA 6240, 144, 1, 200, 152, 109, 83, 21, 133, 7083
1623 DATA 6248, 1, 136, 24, 108, 178, 23, 133, 2, 6856
1624 DATA 6256, 144, 2, 230, 3, 165, 3, 189, 179, 7091
1625 DATA 6264, 23, 133, 3, 174, 177, 23, 160, 255, 7212
1626 DATA 6272, 177, 0, 145, 2, 136, 208, 249, 177, 7366
1627 DATA 6280, 0, 145, 2, 198, 1, 198, 3, 202, 7029
1628 DATA 6288, 200, 236, 32, 164, 24, 172, 176, 23, 7323
1829 DATA 6256, 177, 0, 145, 2, 136, 192, 255, 203, 7411
1830 DATA 6304, 247, 76, 17, 24, 173, 82, 21, 133, 7877
1831 DATA 6312, 0, 173, 63, 21, 133, 1, 173, 178, 7074
1832 DATA 6320, 23, 133, 2, 173, 178, 23, 133, 3, 6309
1833 DATA 6328, 35, 32, 8, 20, 32, 228, 20, 127, 6891
1834 DATA 6336, 13, 10, 93, 69, 94, 32, 68, 69, 6754
1835 DATA 6344, 83, 84, 73, 78, 65, 94, 73, 79, 6963
1836 DATA 6352, 78, 32, 65, 78, 68, 32, 80, 82, 6867
1837 DATA 6360, 69, 83, 32, 81, 46, 255, 32, 7041
1838 DATA 6368, 7, 10, 173, 5, 18, 141, 178, 23, 6331
1839 DATA 6376, 173, 5, 18, 141, 173, 23, 96, 0, 7012
1840 DATA 6384, 0, 0, 0, 0, 0, 0, 0, 0, 6384
1841 DATA 6392, 0, 0, 0, 0, 0, 0, 0, 0, 6392
1842 END

420 BEYOND GAMES
THE FOLLOWING DATA STATEMENTS CONTAIN DECIMAL OBJECT CODE AND CHECKSUMS FOR MEMORY FROM 7680 TO 8191 SUITABLE FOR LOADING WITH THE BASIC OBJECT CODE LOADER.

1900 DATA 7680, 255, 1, 32, 15, 30, 32, 55, 30, 8130
1901 DATA 7688, 32, 200, 30, 24, 24, 144, 246, 32, 8420
1902 DATA 7696, 8, 28, 32, 228, 20, 127, 13, 10, 8154
1903 DATA 7704, 10, 83, 69, 84, 32, 85, 80, 32, 8179
1904 DATA 7712, 69, 68, 73, 84, 32, 66, 85, 70, 8259
1905 DATA 7720, 70, 69, 82, 46, 13, 10, 10, 255, 8275
1906 DATA 7728, 32, 233, 21, 32, 160, 23, 96, 32, 9357
1907 DATA 7736, 196, 17, 32, 43, 17, 174, 3, 16, 8234
1908 DATA 7744, 160, 3, 32, 19, 17, 32, 43, 17, 8067
1909 DATA 7752, 32, 118, 17, 32, 196, 17, 32, 94, 8290
1910 DATA 7760, 30, 32, 211, 17, 32, 118, 17, 32, 8249
1911 DATA 7768, 137, 30, 32, 211, 17, 96, 32, 18, 8341
1912 DATA 7776, 21, 173, 3, 16, 74, 170, 202, 202, 8537
1913 DATA 7784, 32, 26, 19, 202, 16, 250, 173, 3, 8505
1914 DATA 7792, 16, 141, 0, 30, 32, 148, 18, 32, 8209
1915 DATA 7800, 155, 17, 32, 127, 17, 32, 13, 19, 9212
1916 DATA 7808, 206, 0, 30, 15, 239, 32, 43, 21, 8395
1917 DATA 7816, 96, 173, 3, 16, 74, 233, 2, 32, 8445
1918 DATA 7824, 129, 17, 173, 1, 30, 201, 1, 208, 8584
1919 DATA 7832, 5, 169, 73, 24, 144, 2, 169, 79, 8497
1920 DATA 7840, 32, 155, 17, 159, 2, 32, 129, 17, 8393
1921 DATA 7848, 173, 7, 16, 32, 155, 17, 169, 2, 8419
1922 DATA 7856, 32, 129, 17, 173, 6, 18, 32, 163, 8426
1923 DATA 7864, 17, 173, 5, 16, 32, 153, 17, 96, 8385
1924 DATA 7872, 6, 3, 62, 60, 16, 127, 81, 0, 8227
1925 DATA 7880, 32, 224, 16, 205, 198, 30, 208, 23, 8818
1926 DATA 7888, 72, 32, 224, 16, 205, 198, 30, 209, 8875
1927 DATA 7896, 4, 104, 104, 104, 96, 141, 199, 30, 8679
1928 DATA 7904, 164, 52, 231, 30, 173, 199, 30, 205, 8969
1929 DATA 7912, 193, 30, 260, 11, 206, 1, 30, 16, 8607
1930 DATA 7520, 5, 169, 1, 141, 1, 30, 36, 205, 8968
1931 DATA 7528, 194, 30, 205, 4, 32, 121, 31, 96, 8644
1932 DATA 7536, 205, 195, 30, 206, 4, 32, 135, 31, 8776
1933 DATA 7544, 95, 205, 197, 30, 205, 4, 32, 221, 8937
1934 DATA 7552, 31, 96, 205, 196, 30, 205, 4, 32, 8754
1935 DATA 7560, 197, 31, 96, 205, 192, 30, 208, 4, 8923
1936 DATA 7568, 32, 188, 31, 96, 174, 1, 30, 240, 8752
1937 DATA 7576, 4, 32, 52, 31, 96, 32, 45, 19, 8287
1938 DATA 7584, 32, 131, 23, 96, 72, 32, 18, 21, 8409
1939 DATA 7592, 173, 83, 21, 72, 173, 82, 21, 72, 8669
1940 DATA 8000, 173, 83, 21, 72, 173, 84, 21, 72, 8701
1941 DATA 8008, 32, 103, 22, 32, 131, 23, 48, 17, 8416
1942 DATA 8016, 32, 226, 24, 173, 84, 21, 208, 4, 8788
1943 DATA 8024, 206, 85, 21, 205, 84, 21, 32, 214, 8893
1944 DATA 8032, 23, 104, 141, 84, 21, 104, 141, 85, 8735
1945 DATA 8040, 21, 104, 141, 82, 21, 104, 141, 83, 8737
1946 DATA 8048, 21, 32, 43, 21, 104, 32, 45, 31, 8377
1947 DATA 8056, 96, 32, 148, 18, 201, 255, 240, 4, 8050
1948 DATA 8064, 32, 131, 23, 96, 169, 255, 96, 56, 8222
1949 DATA 8072, 173, 83, 21, 205, 6, 18, 144, 12, 8734
1950 DATA 8080, 208, 16, 173, 82, 21, 205, 5, 18, 8808
1951 DATA 8088, 240, 23, 176, 6, 32, 26, 19, 169, 8779
1952 DATA 8096, 0, 96, 173, 82, 21, 141, 5, 18, 8632
1953 DATA 8104, 173, 83, 21, 141, 6, 18, 169, 0, 8715
1955 DATA 8120, 255, 32, 45, 19, 32, 131, 23, 16, 8673
1957 DATA 8136, 32, 29, 20, 32, 148, 18, 201, 255, 8662
1958 DATA 8144, 240, 8, 32, 64, 20, 32, 131, 23, 8594
1959 DATA 8152, 16, 241, 76, 26, 20, 32, 19, 21, 8622
1960 DATA 8160, 173, 83, 21, 72, 173, 82, 21, 72, 8857
1961 DATA 8168, 32, 226, 24, 32, 131, 23, 32, 103, 8771
1962 DATA 8176, 22, 32, 214, 23, 104, 141, 82, 21, 8815
1963 DATA 8184, 104, 141, 83, 21, 32, 43, 21, 96, 8725
1964 END

422 BEYOND GAMES
Appendix E11:
Extending the Visible Monitor

THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUNS FOR MEMORY FROM 4272 TO 4351
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.

2000 DATA 4272, 201, 60, 208, 9, 173, 0, 20, 73, 5036
2001 DATA 4280, 255, 141, 0, 20, 96, 201, 85, 208, 5286
2002 DATA 4298, 9, 173, 2, 20, 73, 255, 141, 2, 4963
2003 DATA 4296, 20, 96, 201, 72, 208, 13, 173, 0, 5079
2004 DATA 4304, 20, 208, 4, 32, 21, 96, 32, 4004
2005 DATA 4312, 174, 21, 96, 201, 77, 208, 4, 32, 5125
2006 DATA 4320, 160, 23, 96, 201, 63, 208, 13, 173, 5277
2007 DATA 4328, 0, 20, 208, 4, 32, 9, 25, 96, 4722
2008 DATA 4336, 32, 25, 96, 201, 80, 208, 4, 5024
2009 DATA 4344, 32, 2, 30, 96, 96, 0, 0, 0, 4500
2010 END
Appendix E12:

System Data Block for the Ohio Scientific C-1P

APPENDIX E12 SYSTEM DATA BLOCK FOR OSI C1P

THE FOLLOWING DATA STATEMENTS CONTAIN DECIMAL OBJECT CODE AND CHECKSUMS FOR MEMORY FROM 4096 TO 4119 SUITABLE FOR LOADING WITH THE BASIC OBJECT CODE LOADER.

2100 DATA 4096, 101, 208, 32, 24, 24, 211, 32, 16, 4744
2101 DATA 4104, 237, 254, 45, 131, 177, 252, 16, 16, 5292
2102 DATA 4112, 96, 96, 0, 0, 0, 0, 0, 0, 4384
2103 END

OK
Appendix E13:
System Data Block for the PET 2001

APPENDIX E13  SYSTEM DATA BLOCK FOR THE PET 2001

THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM 4096 TO 4151
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.

2100 DATA  4096, 0, 128, 40, 39, 24, 131, 32, 30, 4520
2101 DATA  4104, 42, 16, 219, 255, 16, 16, 16, 16, 4591
2102 DATA  4112, 96, 41, 127, 55, 201, 64, 144, 17, 4658
2103 DATA  4120, 201, 96, 144, 10, 162, 14, 141, 76, 4964
2104 DATA  4128, 232, 233, 32, 24, 144, 3, 56, 233, 5085
2105 DATA  4136, 64, 96, 32, 228, 255, 41, 127, 240, 5219
2106 DATA  4144, 249, 96, 0, 0, 0, 0, 0, 0, 4489
2107 END

OK
Appendix E14:
System Data Block for the Apple II

APPENDIX E14        SYSTEM DATA BLOCK FOR THE APPLE II

THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUNS FOR MEMORY FROM 4036 TO 4127
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.

2100 DATA  4036, 0, 4, 128, 39, 7, 7, 160, 222, 4653
2101 DATA  4104, 29, 16, 26, 16, 16, 16, 16, 16, 4246
2102 DATA  4112, 96, 9, 128, 96, 32, 12, 253, 41, 4779
2103 DATA  4120, 127, 96, 9, 128, 32, 253, 251, 96, 5112
2104 END

OK
Appendix E15:
System Data Block for the Atari 800

THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM 3712 TO 4223
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.

2100 DATA 3712, 32, 196, 17, 173, 179, 23, 72, 173, 4577
2101 DATA 3728, 179, 23, 72, 173, 85, 21, 72, 173, 4517
2102 DATA 3728, 84, 21, 72, 173, 83, 21, 72, 173, 4427
2103 DATA 3736, 82, 21, 72, 32, 43, 17, 165, 0, 4168
2104 DATA 3744, 141, 178, 23, 165, 1, 141, 179, 23, 4595
2105 DATA 3752, 32, 118, 17, 165, 0, 141, 82, 21, 4328
2106 DATA 3759, 165, 1, 141, 83, 21, 174, 3, 16, 4364
2107 DATA 3768, 172, 4, 16, 32, 60, 17, 165, 0, 4234
2108 DATA 3776, 141, 84, 21, 165, 1, 141, 85, 21, 4435
2109 DATA 3784, 32, 21, 23, 172, 4, 16, 162, 0, 4407
2110 DATA 3792, 32, 60, 17, 174, 3, 16, 160, 1, 4255
2111 DATA 3800, 32, 13, 17, 104, 141, 82, 21, 104, 4320
2112 DATA 3808, 141, 83, 21, 104, 141, 84, 21, 104, 4597
2113 DATA 3816, 141, 85, 21, 104, 141, 178, 23, 104, 4613
2114 DATA 3824, 141, 179, 23, 32, 211, 17, 96, 0, 4523
2115 DATA 3832, 0, 0, 0, 0, 0, 0, 0, 0, 3832
2116 DATA 3840, 108, 106, 59, 0, 0, 107, 43, 42, 4305
2117 DATA 3848, 111, 0, 112, 117, 13, 105, 45, 61, 4412
2118 DATA 3856, 118, 0, 99, 0, 0, 98, 120, 122, 4413
2119 DATA 3864, 52, 0, 51, 54, 27, 53, 50, 49, 4200
2120 DATA 3872, 44, 32, 45, 110, 0, 103, 47, 0, 4260
2121 DATA 3880, 114, 0, 101, 121, 9, 116, 119, 113, 4573
2122 DATA 3888, 57, 0, 48, 55, 0, 56, 60, 62, 4234
2123 DATA 3896, 102, 104, 106, 0, 0, 103, 115, 97, 4517
2124 DATA 3904, 76, 74, 59, 0, 0, 75, 91, 94, 4372
2125 DATA 3912, 79, 0, 80, 85, 13, 73, 45, 61, 4348
2126 DATA 3920, 86, 0, 67, 0, 0, 66, 88, 90, 4317
2127 DATA 3928, 52, 0, 51, 54, 27, 37, 34, 33, 4216
2128 DATA 3936, 90, 32, 93, 78, 0, 77, 63, 0, 4369
Index

ABSLUT 123
ABS.X 124
ABS.Y 124
absolute mode 16, 123
ACC 124
accumulator 3, 124
ADC 38
addressing
absolute indexed 21
base 20, 33
description 16-17
index 20-21
indirect 27, 45, 119, 126
pointer 45
relative 26, 127
zero page 17, 45, 129
zero page indexed 21
addition 38
ALL.OFF 85
ALL.ON 85
AND 39
Apple computer 2, 6, 7, 44
arithmetic 23, 34
arrow line 67, 72
ASCII 11, 15, 46, 52, 54, 62, 145, 159
assemblers 12, 45
assembly language 1, 8
Atari computer 2, 6, 7, 44
BAD 117
BASIC 7, 26, 28
BCC 24
BCS 24
BEQ 23-24
binary 9, 36
bits 4, 8, 41, 45
bit twiddling 39
BLANK 58
BMI 24
BNE 21, 23-24
BPL 24
break flag 22
bug 49
BVC 24
BVS 24
byte 8
call 28
carry 23, 35, 36, 38
carry flag 22
cartesian coordinates 49
CENTER 55
current graphics 44
CHARS 90
CLD 39
clear screen 57
CLR.TV 57
CLR.XY 58
CMP 22
comma 21
COMMENT 14
compare 22
conditional branch 23
constant 15-16, 22
CPX 21-22
CPY 22
CR.LF 89
CR.LFS 90
data line 67, 69
data mode 61
debugging 49
DEC 37
decimal 15, 39
decimal flag 22, 39
decrement 37
delete 158
DEST 154
DEX 37
directives 18
disassembler 114, 160
display-memory 47, 49
divide 36
documentation 14
DSTYPE 117
DUMMY 162
dummy subroutine 30
DUMPSL 102

EA 154
EDITIT 152
EDITOR 146
EDM.units 148, 153
8080 7
equate 18
error-checking 52
ETX 92, 129, 157
EXTEND 161

fetch 12, 27
FINISH 131
FIXCHR 46
flag 33, 42, 67, 85, 160
FLASHKY 153
flush buffer 157
front end 135
function keys 133

GETKEY 76, 152
GOSUB 28
GOTO 26
graphic 46, 64

hand 44
HEADER 107
hierarchy 73
hexadecimal 9-12, 19, 52, 62, 98, 114
HEXDUMP 160
hexdump 13, 98
high byte 45, 119, 121
HOME 48
hook 30, 84

immediate mode 14-15, 22, 125

IMMEDT 125
IMPLID 125
implied mode 125
INC 37
increment 21, 24, 37
index 25, 33, 118-119
INDRCT 126
IND.X 126
IND.Y 127
input/output 5
input ports 4
INSCHR 149
INSERT 154
insert 149
instruction:
cycle 11-12
set 7
interpreter 7
interrupt flag 22
INX 21, 37
INY 37

JMP 28, 62
JSR 28, 63, 119
juggling 8, 15

key 74

LABEL 14
label 14, 18, 21, 45
label line 67, 69
LDA 14-15, 20
LDX 14, 20
LDY 14
least-significant:
byte 17
bit 45, 54
LIFO 30
listing 159
logical operations 23, 39
loop 22, 24, 66, 139
low byte 45, 121
LPARENT 121

machine language 12, 60
mask 39
MCODES 118
memory 3, 44
memory-mapped display 44
message 84
microprocessor 3, 7
M NAMES 118
MNEMON 118
MNEMONIC 14
mnemonic 14, 114
mode 148-149
MODEKY 153
MODES 119
monitor 60-61
most-significant:
  byte 17
  bit 45, 54
MOVDN 139
move 134
MOV.EA 135, 154
MOVER 143, 160
MOVE TOOL 142
MOVNUM 135
multiply 36
negative 23, 26
negative flag 22
NEXTCH 155
next character 155
NEXTKY 153
NEXTSL 111
nybbles 10, 39

object code 10, 12, 15
Ohio Scientific (OSI) computer 2, 6, 7, 44, 47, 61, 159
ONEBYT 120
opcodes 11-12, 14-15, 27, 114
OPERAND 14
operand 14, 16, 17, 21, 24-28, 38, 114
operating systems 6
OPERND 119
ORA 39
output:
  port 4
  print 84
  vectors 88
overflow flag 22
overlap 135
overstrike 149
OVRCHR 149

P register 22
page 17
PAGE-DOWN 141
PAGE-UP 141
parentheses 119
PC 11-12, 27
PET computer 2, 6-7, 44
PHA 30
PLA 30
pockets 8, 31
pointers 27, 45, 119
POINTER 45
pop 30
POP.SL 95, 155
positive 26
PR.ADR 102
PR.BYT 89
PR.CHR 87
PR.DIS 132, 160
PRDUMP 103, 160
PR.EA 107
previous character 156
PREVKY 153
PREVCH 156
PRINT: 93
print 157
print utilities 84
PRLINE 109
PR.MSG 91
PR.OFF 85
program counter 11, 27
programmable memory 4, 7
PR.ON 85
PR.SA 106
PRTBUF 157
PRTKEY 153
pseudo-addressing mode 129
pseudo-mnemonic 117
push 30
PUSHSL 95, 154

QUITKY 152

RANGE 107
registers:
  A 3, 8, 15
  compare 22
description 3

INDEX 431
index 20
processor status (P) 22, 64
transfer 33
X register 3, 8, 20, 24, 49, 56
Y register 3, 8, 21, 49, 56
RELATV 127
relocate 28
RETURN 28
return 29
ROM 4, 6
ROMPRT 88
ROMTMT 88
rotate 36
ROWINC 48
RPAREN 121
RTS 28, 62, 136
RUBKEY 153

SA 154
SAHERE 154
screen 44
screen utilities 58
SED 39
SELECT 94, 117
set 39
SETBUF 147
SET.DA 143
shift 35
SHOWIT 148
6502 8, 11
6800 7
source code 13, 18
SPACE 89-90
space bar 65
STA 16, 20
stack 30
status 150
STRIKE 153
string 129, 145
SUBPTR 119
subroutines 30, 62, 119
SUBS 120, 130
subtraction 38
SYSTEM DATA 88

table 32, 118-119
TAX 33

TAY 33
TEST 24
TEX 92, 130
text:
buffer 146
description 130, 145, 150
editor 145, 160
title 142
toggle 41
tool 142
truth table 39
TVCOLS 49, 150
TV.DISL 132, 160
TVDOWN 51
TVDUMP 98, 160
TVHOME 55
TVPLUS 51
TVPOP 56, 59
TVPRTR 45
TVPUSH 56, 59
TV.PUT 46
TVROWS 49
TVSKIP 51
TVT 84
TVT.OFF 85
TVT.ON 85
TVTOXY 48
TWOBYT 121
TXA 33
TXMODE 130
TYA 33

UPDATE 74, 162
USR.OFF 85
USR.ON 85
USRROUT 88
utilities 120, 134

Visible Monitor 63, 160
VBYTE 52, 54
VUCHAR 52

XINDEX 122
XOR 39, 41

YINDEX 122

Z80 7
zero 22
zero flag 22
ZEROPG 128
ZERO.X 129
ZERO.Y 129
Beyond Games: Systems Software for Your 6502 Personal Computer

By Ken Skier

Use your 6502 personal computer for more than games! Learn how it works and how to make it work for you. This book, for Apple, Atari, Ohio Scientific and PET 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?, Beyond Games 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, systems analyst for Wang Laboratories, Inc, designs software for word processing and other applications concerning the office of the future. A Massachusetts Institute of Technology graduate, he co-founded the M.I.T. Writing Program, where he teaches science fiction writing. He lives in Cambridge, Massachusetts, with his wife Cynthia and a nameless white cat.