The Visible Computer: 6502

Machine Language Teaching System

Commodore 64

The Visible Computer: 6502

Machine Language Teaching System

Commodore 64 Version

Software Masters™
3330 Hillcroft/Suite BB
Houston, Texas 77057

Copyright © 1984 Software Masters

The Visible Computer: 6502 Program is copyrighted and all rights are reserved by Software Masters. Only you, as original purchaser, may use The Visible Computer: 6502 computer program and only on a single computer system.

The Visible Computer: 6502 User Manual is copyrighted and all rights are reserved by Software Masters. This manual may not be copied, in whole or in part, by any means, without the express written permission of Software Masters.

Warranty

For a period of ninety days after purchase, Software Masters will replace defective Visible Computer: 6502 program disks free of charge. Replacement cost after ninety days is $5.00.

Program and Manual by Charles Anderson

Commodore 64 Version by J.I. Blackshear, Jr.

Introduction

The Visible Computer: 6502 Machine Language Teaching System combines this manual with a diskette containing a 6502 simulator program to provide a systematic way to learn machine language programming on the Commodore 64 computer.

This manual is a tutorial on 6502 machine language and the related concepts of binary and hexadecimal numbering systems. The program is a simulation of the 6502 microprocessor, slowed down and opened up for close inspection. Note: From a programming standpoint, the 6510 processor used in the C-64 is identical to the 6502; learning one is learning the other.

Prerequisites

This manual assumes some familiarity on the reader's part with the Basic programming language. Programming is programming, and the more experience you have with any form of it, the better. It presupposes no prior exposure to machine language, and includes preliminary chapters on alternate numbering systems and hardware fundamentals.

Hardware Requirements

To run The Visible Computer, you will need a Commodore 64 computer with 1541 disk drive. A printer is optional. Sound capability, through either your TV speaker, display monitor, or an external amplifier, is assumed, though not absolutely necessary.

Scope

Most of the books that profess to teach 6502 machine language work so hard at thoroughness, with endless chapters on floating point arithmetic and control programs for hypothetical daisy wheel printers, that they skimp on the fundamental job of delivering the concepts. The Visible Computer is designed to get you over the initial hurdles of machine language programming, not to present algorithms for controlling elevator systems.

Learning everything in this manual will not qualify you to work at Microsoft writing 6502 Cobol compilers. However, if you apply yourself, it will get you to the point where you will be able to develop independently in your area of interest, be it arcade games, chess programs, or new and wonderful operating systems. And who knows, someday the Microsoft recruiter might just give you a call.

About This Manual

Chapters 1, 2, and 3 are the standard introductory fare of Hex, Binary, and Computer Block Diagrams. They may be skipped by those who have already been through eleven discussions of hex and binary (and if they see one more block diagram of a computer, they'll scream).

The TVC program disk is not used until Chapter 4. It would be a good idea to skip there quickly right now and make sure that your disk works correctly.

The heart of the course is Chapters 6 through 16, where you'll work through a series of progressively more difficult 6502 machine language programs contained on the TVC disk. By the end of Chapter 16 you will have read about, and seen demonstrated, nearly every 6502 instruction, and will have earned the honorary title of *TVC Master*.

Chapter 17 shows how to write machine language programs, by taking ideas and working through the design and coding phases to produce working programs. The role of the assembler is discussed.

Chapter 18 is a rundown on the options available in assemblers, with tips on debugging, techniques for interfacing Basic and machine language, and a suggested reading list.

Table of Contents

1. What Is Machine Language? 1

2. Alternate Numbering Systems 4
Positional Numbering Systems
Binary and Hexadecimal
The Logical Operators
Self Test

3. Hardware 18
Computer Block Diagram
The 6502/6510 Microprocessor
Memory Types
C–64 Memory Map
How Machine Language Works

4. Getting Started 28
Booting Up
The TVC Display
Talking to the Monitor
TVC Calculator
ERASE RESTORE WINDOW BASE

5. Working With Memory 34
TVC Memory Map
Displaying and Altering Memory
Writing to Registers

6. First Programs 39
The Registers
PROG1: Loading the Accumulator
The 6502 Simulator
Microsteps
PROG2: Storing the Accumulator
PROG3: Loading and Storing X and Y
PROG4: The Transfer Instructions

7. Processor Status Register 48
P Register Flags
Setting and Clearing the Flags
Conditioning Z and N
PROG5: Disassembly
PRINTER Command

8. Branches: Decision Making 53
Decrement/Increment Instructions
BNE: The Branch instructions
PROG6: Looping
The Step Command: Simulator Control

9. Addressing Modes 57
PROG7: Zero Page
PROG8: Absolute

10. Subroutines: The Stack 61
PROG9: JMP
The Stack
Pushes and Pulls
PROG10: JSR/RTS
PROG11: Stack Pitfalls: POP
PROG12: JMP Indirect

11. Instructions That Work: ADC/SBC 69
ADC: The Accumulator
The Carry Flag
PROG13: ADC
PROG14: Multiprecision Addition
SBC: The Borrow Flag
PROG15: SBC
PROG16: Multiprecision Subtraction
PROG17: Multiplication
PROG18: Division

12. Beyond Adding and Subtracting 75
The Shift Instructions
PROG20: Multiprecision shift

The Logical Operators
Reading the Joystick
FCHECKPROG
PROG21: AND/OR

13. Indexing: Special Uses for X and Y 82
PROG22: Block Move
Arrays
TOTALPROG
PROG23: Zero Page indexing

14. The Kernal: Canned Subroutines 87
Kernal Functions
GETIN
RDTIM
MASTER Mode
Sharing Page Zero

15. Indexing, Part II 92
Indirect, Indexed
CLEARPROG
GO Command
Indexed, Indirect
BTABLEPROG

16. Some Fine Points 98
NOP
Interrupts
RTI
INTPROG
Signed Numbers/Two's Complement
PRACTICEPROG
Binary Coded Decimal

17. Putting It All Together 107
Writing Machine Language Programs
Bubble Sort Algorithm
SORTPROG
ASCII Organ

Making Sound with SID
Color Memory
ASCII Organ Listing

18. Where Do I Go From Here? 128
Buy an Assembler
Basic and Machine Language Hybrids
What is Basic?
Suggested Reading

Appendices

A. Behind the Scenes of TVC 133
B. TVC Character Set/Standard Color Chart 135
C. Monitor Commands Reference 137
D. Simulator Reference 144
E. Error Messages 146
F. 6502 Reference 148

1.
What is Machine Language?

And If It's So hard, Why Do People Use It? This is a fair question, and if you haven't asked it yet you probably should have. Before we get into the *hows* of machine language we're going to touch on the *whys*.

Although Basic gets all the publicity, the language closest to a C–64's silicon heart is 6502 machine language. Later chapters of this book will develop a more formal definition; for now, it suffices to say that 6502 machine language is the fundamental language of the Commodore 64 computer. Not a moment passes during a C–64's powered-on lifetime when it is *not* executing 6502 machine language. In fact, languages like Basic are nothing but clever ruses to save poor humans from the wicked binary ways of 6502 processors.

As to the widespread rumor that machine language programming is more difficult than programming in Basic, consider these two sets of instructions for building a cedar fence in your backyard.

Basic

> Using 6' by 6" cedar slats, with supporting posts every 8 feet, build a fence enclosing your back yard.

Machine Language

> Drive to lumber yard. Purchase 722 6' by 6" cedar slats. Load into truck. Drive home. Unload truck. Start at northeast corner of back yard. Dig a hole three feet deep. Get post from pile. Insert post into hole. Cement post. Move 8 feet west. If not yet at corner, dig a hole three feet deep. Get post from pile. Insert post into hole. . .

Is the second set of instructions more difficult than the first? Not really. It *looks* more involved, and certainly took longer to write down, but the individual jobs that make up the second paragraph are simplic-

ity itself: "Move 8 feet west", "Get post from pile". So it is with machine language. Working from a limited palette of about 50 simple instructions, we achieve complex results by combining them cleverly.

Machine language programmers have to take smaller steps to get where they're going. That means it takes more time to write programs. Longer to design, longer to code, and much longer to debug. As a rule of thumb, 10 times longer than working in Basic. Furthermore, almost anything you can do in machine language can be done in Basic.

So why do people knock themselves out learning and writing machine language programs? Two main reasons: 1.*For speed.* 2.*For more speed.* Machine language programs execute ten to one hundred times faster than similar programs written in Basic. (Purists and other curmudgeons will object to this statement, and there *is* something to be said for the fact that unless someone had written the machine language program named Commodore Basic, Basic would not exist, even as an alternative.) Is speed that important? It depends.

In an accounting program, where the computer spends most of its time waiting for the operator to hit a key, or the printer to finish, or a disk drive to get something, blinding speed is not important. We hear phrases like "printer bound" and "floppy bound". A program that is printer bound can only be speeded up by buying a faster printer. Writing accounting programs in assembly language, then, results in programs that wait for user input at very high speed, and cost 10 times as much to develop as acceptably speedy programs written in Basic. Clearly, an idea whose time has not come.

But sometimes speed is desirable, even critical. Most arcade game programs could not function written in Basic. Animation makes tremendous demands on a computer system: moving objects around the screen requires the carefully coordinated movement of thousands of numbers. At Basic speeds games like Pac Man would be exercises in patience, with sluggish ghosts that take minutes to get from one side of the maze to the other. So game programs, especially the arcade type, are one place where we need the speed of machine language.

Many times the best tactic is a combination of Basic and machine language. Take the accounting application from a minute ago. Most of it can be written in slow-to-execute, but fast-to-program Basic. Certain time consuming jobs can be handled in machine language. For instance, sorting.

Sorting programs written in Basic, for those of you who have avoided learning about such things thus far in your programming careers (and your time is coming), are slow. *Really* slow. Sorting a list of 3,000 employee numbers into a stack with the biggest at the bottom and the smallest at the top takes at least two minutes, and maybe as many as 10, depending on what method we tackle the problem with. (The methods available range from the crude—read easy—to the complex. Graduate students as yet unborn will earn their degrees with programs that sort .01 percent more efficiently than some other program.)

Two minutes is an important length of time to the operator of an accounting package, and ten minutes is an eternity. The strategy followed by the smart programmer, then, is to use Basic for everything except the sort itself—and pass that job to a hard-to-write, but breathtakingly fast machine language program. After 10 seconds (or one or two, depending on how fancy a method we use), the Basic program is handed on a silver platter a sorted list of employee numbers.

Sharing the work between machine language and Basic is a good technique, employed by countless programs, including TVC itself. Mostly Basic, machine language where you need the speed.

To sum up: The best reason for programming a Commodore 64 in machine language is to speed up a process that would be too slow otherwise. Conversely, except as a learning exercise, it is a waste of time to use machine language for something that would be acceptably fast written in Basic.

2.
Alternate Numbering Systems

If you bought The Visible Computer with the hope that it would somehow save you the effort of climbing Mount Hexadecimal, picking you up magically and dropping you safely into the valley of machine language programming on the other side, sorry, no can do.

People don't use binary and hexadecimal numbers to make machine language programming easy; they use these weird numbering systems to make machine language *feasible.* Although it is arguable, barely, that one could learn some machine language without ever learning hex, a person who went that route would find himself working twice as hard as someone who learned the tools of the trade first and the programming second.

A Twelve is a 12 is a 1100

Most people agree that the symbols "1" and "2", printed together, like this:

12

have a certain numeric meaning. Specifically, "12" represents the quantity of periods printed here:

. . . .

Or this many commas:

, , , , , , , , , , , ,

But there is nothing intrinsically "12-like" about these symbols sitting next to each other. If we wanted to form a club that said from now on, "*" would stand for 12 and "#" for 17, we could. Let's do that. You and I will be the charter members of the "* = 12 and # = 17" Club.

Until further notice, "*" represents this many things:

,, ,, ,,, ,,,, ,

and "#", this many:

, ,, , , , ,,, ,,,,, , , ,

How many eggs in a dozen? Very good, * is correct. What fab group had a 1964 hit called *She was Just #*? Right again, the Beatles. (Fooled you—the song's title was really *I Saw Her Standing There*.) Although we'd have to work fairly hard at it the first couple of months, eventually it would become almost as natural as the old way.

But not when we're doing math. What's * times #? Even for people as smart and good looking as members of the club, getting that answer is pretty tough. Whereas everyone else's notation, "12 times 17", lends itself to computational tricks like carrying and partial products, our representation gives not a clue to the answer. We'd have to either memorize all the combinations of multiplications and divisions for * and #, or give up comparison shopping forever.

This situation isn't as farfetched as you might imagine. Consider the Roman Empire. For all its accomplishments, Rome's state-of-the-art method for representing numbers was what we now call Roman numerals. (Although I suppose they simply referred to them as "numbers"). As with our club's method, Roman numerals are okay for some things, (like the names of popes and book report outlines), and lousy for others, like calculations.

It's a wonder they built Bridge I considering how hard their engineers had to work to do this simple division:

XXIX / XIV

Stop and think about it. If you had to solve this problem you'd probably proceed like this: Convert both parts into "normal" notation. Divide using conventional techniques. Finally, convert the answer back to Roman numerals. Unfortunately, "normal" notation hadn't been invented yet, and wouldn't for another 500 years.

Around 500 AD an Arabian astronomer devised a better system. Not only was the new Arabic notation better for representing long numbers than the Roman method, it greatly facilitated arithmetic calculations. Let's see why the Arabic method is so powerful. Numbers written with this system can be methodically broken into their component parts.

The value of a digit depends on its *position* the number. The value is always ten times the value of the same digit one position to the right, and one-tenth the value of the same digit one position to the left. This positional chart shows how numbers can be broken down into their components.

	Fourth Digit	Third Digit	Second Digit	First Digit
	10^3 1000	10^2 100	10^1 10	10^0 1
3,479 breaks into:	3 * 1000 3000 +	4 * 100 400 +	7 * 10 70 +	9 * 1 9
209 is:		2 * 100 200 +	0 * 10 0 +	9 * 1 9

The biggest problem keeping previous designers of numerical representation schemes from implementing a system like this was that they never saw a need for a character to represent *zero*, the quantity nothing. Without zero to serve as a placeholder, you can't have positional representation.

The usefulness of the Arabic positional system has nothing to do with the symbols that form the counting alphabet. *6*'s and *7*'s aren't any better or worse than *V*'s and *X*'s. It's the positional concept that makes it better. We will refer to this ingenious and familiar scheme not as "Arabic Positional", but as decimal (from the Latin *decem*, ten), because 10 is the value that each position is based on.

But this quantity:

* * * * * * * * * *

is by no means magic in the grand scheme of the universe. No more "round" or "even" than this number:

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

So why do we use 10 as the base value of our positional notation? Class? Anyone have a guess? Right. In all probability, because people have 10 fingers, and for millions of years, fingers were all we had for representing numbers. On a planet where beings have two hands of four fingers each, we can reasonably predict that their positional numbering system is based on the number:

* * * * * * * *

The decimal numbering system has remained unchanged for 1,500 years because it is an extremely useful way of representing numbers. There exist computational methods that allow 12-year-olds to calculate five digit square roots with nothing but paper and pencil. And in all probability it will be popular 1,500 years from now, even though the advent of the $4 calculator makes some of its best features (ease of manual calculation) moot. If Roman numerals could have hung in there until the Age of Cheap Calculators, they would have been in good shape.

Unfortunately, for programming computers, especially in machine language, decimal falls flat on its well known face. Because of the way they work, computers have a working vocabulary of only two digits. It's easy to make an electronic device store a 1 or a 0, and hard to make one that can store zero 0 through 9. A sensor that can detect whether a light bulb is on or off is trivial. A device that can consistently detect 10 discrete levels of brightness is far more complex.

Computers need a two digit, or *binary*, positional numbering system. Every fancy trick done by computers boils down to manipulations of 1's and 0's.

We don't need unique symbols to represent values 2 through 9 because they can be formed by combinations of 1's and 0's, just as decimal doesn't need more than ten symbols to represent values greater than 9.

Here's a positional chart for the first four places in the binary numbering system.

Binary Positional Chart

Fourth Digit	Third Digit	Second Digit	First Digit
2^3	2^2	2^1	2^0
8	4	2	1

```
1010 breaks into:

    1 * 8    +     0 * 4    +  1 * 2    +     0 * 1 = 10 dec.

1110 is:

    1 * 8    +     1 * 4    +  1 * 2    +     0 * 1 = 14 dec.
```

In programming small computers, the most common lengths of binary numbers are eight and 16 digits. Bits are numbered from right to left as shown in this diagram. This is the first of many cases we'll encounter in machine language where we start counting at zero, not one, so get used to it. Bit 0 is often called the least significant bit (LSB), and bit 7, the most significant bit (MSB).

It is also common to retain leading zeros when working with binary. For example, you might see 101 written as 00000101.

7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	1

Binary numbers can be added and subtracted with the same techniques we know for decimal.

```
                     1              11   <-- carrys
   1010            1010            1001
 + 0100          + 0010      +     0011
   1110            1100            1100
```

Carrys happen a lot in binary addition, and borrows are common in subtraction. Otherwise, nothing too taxing about binary arithmetic.

"Round numbers" in binary are the powers of two: 1, 2, 4, 8, 16, 32, 64,

128, 256, 512, 1024, 2048, 4096... These values will turn up throughout your machine language programming career, so get acquainted with them.

Here's a formula (the only one in this book) to calculate the largest number X you can store in n positions of base B numbers:

$$X = B^n - 1$$

The largest value you can represent in three digits of decimal is 999 ($10^3 - 1$). Base seven can represent numbers as big as 342 ($7^3 - 1$) in three places. Binary can only get as high as 7 ($2^3 - 1$). Representing even modest quantities in binary tends to be wasteful of paper. Counting to 10:

```
   0
  01
  10
  11
 100
 101
 110
 111
1000
1001
1010
```

To handle the range of numbers we encounter in day to day life takes a lot of binary digits.

```
365 = 101101101

      1*2^8 + 0*2^7 + 1*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0

531 = 1000010011

1*2^9 + 0*2^8 + 0*2^7 + 0*2^6 + 0*2^5 + 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0
```

Numbers like 1000010011 and 101101101 have a tendency to confuse people. It helps a little if we clump binary digits ("bits") into even groups. Four bits make a nibble. Eight bits make a *byte* (isn't that *precious*!) In nibble form, 531 is 0010 0001 0011. That's a little better, but not much.

Of course, just because computers need to run internally with binary numbers doesn't mean that humans who use computers have to deal with all 1's and 0's. Basic programmers can and do live in a fantasy

world where decimal is king. Unfortunately, machine language programmers must frequently deal with the computer in straight, undiluted binary.

One way around the computers-love-binary-but-people-love-decimal problem is to let the human think in decimal—and convert into and out of binary as necessary to communicate with the machine. Good idea, but converting between binary and decimal is, as mathematicians say, nontrivial. Going from binary to decimal is easiest, so let's start there.

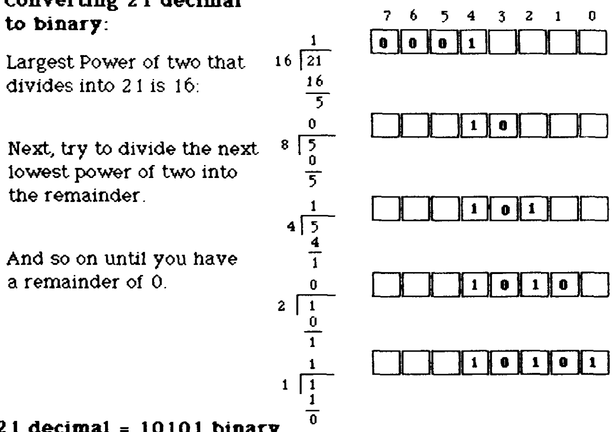
Take a binary number. Starting from the leftmost (most significant) digit, accumulate the decimal equivalent of each position.

```
Position:               5      4      3      2      1      0
Decimal Equivalent:    32     16      8      4      2      1
-------------------------------------------------------------
0000 1001  is                         8  +                 1 =  9

0001 1010  is                16  +    8  +          2        = 26

0010 0101  is          32  +                 4  +          1 = 37
```

If you can remember the powers of two and do addition, you can convert from binary to decimal.

Decimal to Binary

Going this direction is harder. The conversion resembles a simplified form of long division.



Don't practice this any longer that it takes to get the general idea. You can buy calculators that eat base conversions for breakfast.

Even with base conversion calculators, the chasm between binary-loving, paper-wasting computers and 10 fingered, tree-loving human beings is a mile wide. Numbers that are very "round" in binary, such as 1000 0000 0000, are very jagged in decimal (2,048). And vice versa: 2,000 decimal is 0111 1101 0000 binary. Luckily, there's a bridge, called hexadecimal, across the binary/decimal gap.

Suppose Phones Had Two Buttons

Suppose the phone company decided to release a new, improved telephone: "*DigiPhone, The Phone of Tomorrow*", with only two buttons, 1 and 0. They'd have a big advertising campaign to convince people that the DigiPhone would be faster, more modern, and generally better in every way than the old, decimal phones.

Everyone's telephone number is converted to binary: 844–7171 becomes 1000–0000 1110 0100 1100 0011. Area codes get expanded from three digits to 10, enough to cover all 1,000 possible area codes. The phone book doubles in size, but that's no problem—they make the type twice as small.

But the American people are not adjusting well to the new system. They say it's almost impossible to correctly dial, much less memorize, a phone number like:

(0010 1100 1001) 0101–0000 1000 1001 0011 1000

One mistake and you're calling a McDonald's in Kansas City instead of your grandmother in Rockford, Illinois.

The phone company has already built 286 million DigiPhones and they're not about to junk them. But they do offer a compromise. They take out full page ads in newspapers across the country:

> "Here's what we'll do, America. We'll go back to our old phone books and publish everyone's number in the old 10 button form. Numbers will be easy to remember, just like before. When you need to call someone, convert it to 2 button format and make the call."
>
> "Converting your old fashioned decimal telephone number into modern, digital form is a breeze. First, try to divide 8,388,608 into the number. If it fits, the first digit is one, if it doesn't, the first digit is zero. Next, divide 4,194,304 into the remainder. If it fits, the second digit is a one. Otherwise, it's a zero. Next,..."

People let the phone company know that a ten minute calculator session each time they made a call was a less than perfect solution. A company think tank huddled for a week and announced a second compromise.

The solution? A new phone book, with numbers listed in a new, fairly easy to remember format that converts easily, almost automatically, into binary. The great breakthrough? Something called *hexadecimal*. Easier for people to remember than binary. Not quite as easy as the decimal they've been using since the first grade, but much easier than binary. And easy to convert into and out of binary for dialing.

Whereas decimal has 10 symbols in its counting alphabet, and binary two, hexadecimal has 16. This is a problem because there aren't symbols laying around to represent the six values for 10–15. Although we could have invented new symbols, it was expedient to use something that people and other writing machines already knew how to write. It was decided that the first six letters of the alphabet would stand for the missing symbols: A = 10, B = 11, C = 12, D = 13, E = 14, F = 15. Music set a precedent when it stole letters to stand for Do-Re-Mi-Fa-So-La-Ti.

Armed with the notion that letters can sometimes be numbers, here's a positional chart for hex.

Hexadecimal Positional Chart

Fourth Digit	Third Digit	Second Digit	First Digit
16^3	16^2	16^1	16^0
4096	256	16	1

```
A3F breaks into:

                10 * 256  +  3 * 16  +     15 * 1  =  2,623

D006 is:

   13 * 4096  +   0 * 256  +  0 * 16  +      6 * 1  = 53,254
```

The utility of hex isn't the ease with which it converts into and out of decimal—it's how well it works with binary. Hex gives humans good, "ball park" feel for numbers (admittedly, not as comfortable as decimal), with straightforward, one-to-one conversions into binary. A03 may look strange to you now, but it's a heck of a lot easier to deal with than 1010 0000 0011. How easy is it to convert between hex and binary? Examine this chart that counts in all three bases.

Decimal	Binary	Hex	Decimal	Binary	Hex
0	0000 0000	00	16	0001 0000	10
1	0000 0001	01	17	0001 0001	11
2	0000 0010	02	18	0001 0010	12
3	0000 0011	03	19	0001 0011	13
4	0000 0100	04	20	0001 0100	14
5	0000 0101	05	21	0001 0101	15
6	0000 0110	06	22	0001 0110	16
7	0000 0111	07	23	0001 0111	17
8	0000 1000	08	24	0001 1000	18
9	0000 1001	09	25	0001 1001	19
10	0000 1010	0A	26	0001 1010	1A
11	0000 1011	0B	27	0001 1011	1B
12	0000 1100	0C	28	0001 1100	1C
13	0000 1101	0D	29	0001 1101	1D
14	0000 1110	0E	30	0001 1110	1E
15	0000 1111	0F	31	0001 1111	1F
			32	0010 0000	20

See the relationship between hex and binary? One hex digit can stand in for each binary nibble. Once you've memorized the hex equivalent of each of the 16 nibble patterns, conversion between hex and binary is a snap. A hex telephone number like 4–56CA0 becomes:

4 = 0100
5 = 0101
6 = 0110
C = 1100
A = 1010
0 = 0000

Put it all together and you've got the binary equivalent:

0100–0101 0110 1100 1010 0000

Converting from binary to hex is equally simple. Substitute the hex equivalent of each nibble, and you've got it.

```
0011 0111 0001 1000 1100 0010
  3    7    1    8    C    2    = 3718C2
```

There are two problems left in acclimating ourselves to this new numbering system: First, how do we tell whether a number like 345 is hex or decimal just by looking at it, and second, how on earth do we pronounce something like F3C0?

To clear up the former situation, 6502 programmers agreed to always precede hex numbers with a dollar sign ("$"). This convention will be followed in this book. It has nothing to do with Basic's use of "$" to indicate string variables. 345 is a decimal number. $345 is a hex number equal to 837 in decimal.

For most of you the long term problem will be how to internally verbalize hex numbers containing letters. No one conquers this entirely, but as a rule, call the thing by each character if it contains a "funny" number. $C13 is "cee-one-three". Also, try calling $F000, "ef-thousand" and $C00, "cee-hundred".

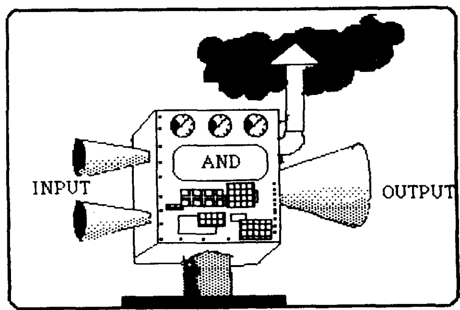
Logical Operators

Binary numbers have some properties that go beyond simply representing decimal values and wasting paper. Numbers as simple as 1 and 0 lend themselves to some special tricks involving what are called the logical (or *boolean*, after George Boole, 19th century English mathematician) operators. These operators are *and*, *or*, and *exclusive or*.

The logical operators are not unlike the four common arithmetic operators, plus, minus, multiply, and divide. The biggest difference is that they operate on binary numbers only one digit long. Sometimes the terms "true" and "false" are used in place of 1 and 0. Examples of logical operations are:

1 AND 1
0 OR 1
True AND True

Frequently, logical operations are shown schematically, as a "black box" with two inputs, a mysterious internal process, and one output.



The Rules

An AND operation yields a 1 if and only if both inputs are 1.

An OR operation yields a 1 if one or both inputs are 1.

An EOR operation yields a 1 if the inputs are different.

That's it for the rules. Not much to them. You've probably used logical operators in Basic programs, perhaps without knowing it. Basic's IF statement is based on logical operations.

IF (expression is true) THEN do this.
IF A > B THEN GOTO 1000

To handle this line, Basic first resolves the assertion portion of the command (A > B) to a simple logical value: either true or false. if A is less than or equal to B, false is inserted. If A is greater than B, a true is inserted. By definition, falses cause THEN statements to be bypassed, and trues cause them to be executed.

IF A OR B THEN GOTO 1000

will cause a branch to 1000 if either variable A or variable B is nonzero. (Basic considers anything nonzero to be a 1). Logical operators can be grouped and combined to express complex relationships.

IF A > B or (FLAG and G < 14) THEN GOTO 1000

This comes natural to most people, because we phrase such expressions everyday:

"If I can find it and you give me the money, I'll buy it."

"If it doesn't rain tomorrow or if you leave the car, I'll go downtown"

"If the copy machine is working, or Bill has the flyers printed and I can get them in time, you'll get your brochure."

Final Exam/Alternate Numbering Systems

Fill in the blanks.

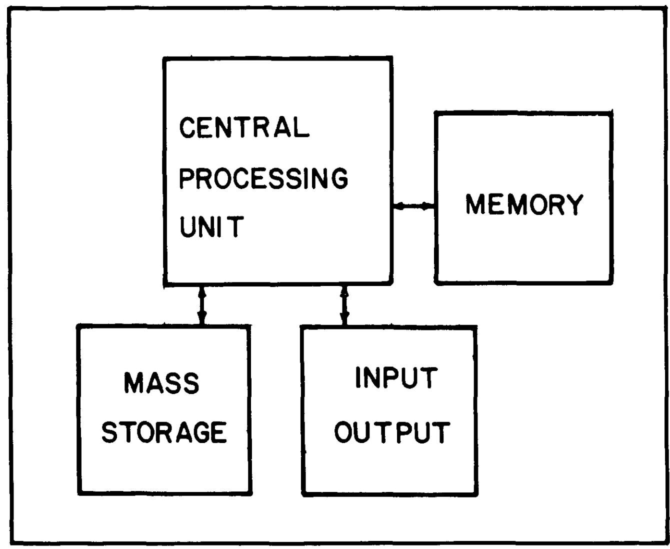
BINARY	HEX	BINARY	HEX
1001 1010	___	_________	$F0
1111 1011	___	_________	$02
0000 0001	___	_________	$CA
1111 0000	___	_________	$0C
1100 1101	___	_________	$DD
0101 1010	___	_________	$11
1011 1011	___	_________	$E6

Convert 10111 to decimal.
Convert 69 to binary.
Perform these logical operations.

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

3. Hardware

A Control Data Corporation Cyber 6600 computer is big enough to fill a medium sized house. A Commodore 64 doesn't weigh 10 pounds soaking wet (perish the thought). But these machines have a lot in common; in fact, at the block diagram level they are identical.



Central Processing Unit (CPU)

The absolute monarch of every computer is the Central Processing Unit. The CPU makes all the decisions and puts the other components through their paces. Although there are almost as many different central processing units as there are computers, they each share the same duties of control, decision, and calculation.

Memory

Memory is second fiddle to the CPU, but still an indispensable member of the team. The CPU goes to memory for the stream of numbers that govern its operation, a machine language program. The fundamental

operation of a computer is the CPU *reading numbers out of memory and writing numbers into memory.* Were it not for the need to occasionally communicate with human beings, CPU and memory could happily function without the other two components.

Mass Storage

Mass storage holds data that isn't needed immediately and there isn't room for in memory at the moment. Mass storage usually has desirable financial qualities compared to memory; it costs less per byte. Examples include disk drives and cassette tape.

Input/Output (I/O)

These are links that connect the binary, numerical world of a computer with the world of people, devices such as printers, keyboards, and joysticks.

The Commodore 64

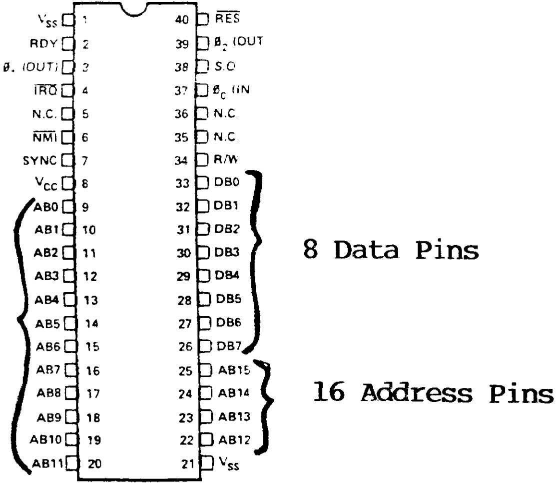
The Commodore 64 uses the 6510 microprocessor as CPU. A microprocessor is an integrated circuit (IC, or chip) that has an entire CPU squeezed onto it. The CPU of a large mainframe computer may consist of a thousand or more IC's. The 6510 is a top-of-the-line member of the 6500 series of processors. However, since from a software standpoint it behaves exactly like the more familiar 6502, we will refer to it in this book as a 6502. When someone writes machine language programs for the 6510, they are using 6502 machine language.

The 6502 was introduced in 1975 by a small California company named MOS Technology. The designers of the 6502 borrowed from and improved upon an earlier microprocessor, the Motorola 6800. It is a tribute to the quality of their design job that the 6502 is still a powerful, workable processor for small computers. MOS Technology was subsequently bought out by Commodore Business Machines. Every personal computer ever built by Commodore, beginning with the 4K PET in 1977, has been designed around the 6502 processor.

Commodore is far from the only company using the 6502 in their machines. Apple and Atari computers feature it. Many devices that are not full blown computers, such as display terminals and video games, have a 6502 calling the shots.

The 6502's greatest challenger for supremacy in the 8 bit microprocessor field is Zilog's Z-80. Although the 6502 and the Z-80 have more similarities than differences, the Z-80 is considered a *register-oriented* processor and the 6502 a *memory-oriented* processor. The Z-80 has more on-board storage, and the 6502 more flair in dealing with memory.

The 6502 is said to be an eight bit microprocessor because it deals with memory in eight bit chunks. This number stems ultimately from the fact that eight of the 40 pins on the 6502 handle the transfer of binary numbers into and out of the chip. Each leg transmits and receives the electronic equivalent of 1 and 0.



Eight bits is enough to represent numbers from 0 through 255. Although this sounds like a serious limitation, a little programming, teamed with the 6502's tremendous speed, allows the use of numbers as big as we want. Tremendous speed? In the the time it takes to press and release the return key, a 6502 can do 50,000 eight bit additions.

16 of the 6502's 40 pins are used to specify addresses to memory. This amounts to a 16 digit binary number, and means there are 65,536 (2^{16}) memory cells potentially addressable by a 6502. Memory can be thought of as a series of numbered cubbyholes, 65,536 of them, maybe

in a giant roll-top desk; each cubbyhole has a slip of paper that can hold a number from 0 – 255 (actually, eight tiny slips of paper just big enough for a one or a zero). Reading memory is the act of first specifying the cubbyhole and then reading back the number stored there. Writing to memory involves locating a specific cubbyhole, and scribbling a new number on the slip of paper, erasing whatever was there before.

The 6502 in a powered-up C–64 is continuously engaged in a fast dialog with its memory. If you were to put your ear against the keyboard, and were very quiet, you might hear something like this (then again, you might not):

6502:	Memory—Give me the contents of cell 45601.
MEMORY:	Okay. That number is... uh, 134.
6502:	Now I need the contents of 45602.
MEMORY:	That's going to be... 101.
6502:	Hmmmm... Very interesting. (He thinks for a microsecond or two). Okay, I need you to put 59 in location 101.
MEMORY:	You got it.

There are three basic types of memory cells that make up the memory mechanism a 6502 will find attached to its address and data pins.

RAM

RAM is the most useful type of memory cell, and not coincidentally, the most plentiful. RAM stands for Random Access Memory, meaning you can ask one microsecond for location 3 and the next microsecond for location 6,319. As opposed to cassette tape, a *sequential* storage medium. If you want the last byte on a tape you must go through the first 22,000 to get it. A memory cell implemented with RAM will obediently read and write data at the command of the CPU. When you turn off the power to a C–64's RAM, within a few milliseconds, the numbers stored there disappear. (Who among us hasn't gnashed his teeth because of this at least once.)

ROM

Like RAM, a memory cell implemented with ROM contains numbers that the 6502 can read (in any order; it's just as random access as RAM). The difference is that the numbers in a ROM cell are permanently en-

graved at the factory, and cannot be changed, no matter how many times the CPU tries to write to it. Thus the acronym: Read Only Memory. This is both a liability and a blessing. It's not very flexible (what if you want to do something with the computer that doesn't need these numbers?), but has the endearing quality of withstanding being turned off without losing the numbers stored there.

I/O Locations

I/O locations are the third type of memory cell. These are memory locations that are tied to elements of the computer outside the CPU-ROM/RAM clique. I/O locations let the 6502 communicate with the rest of the machine. Some I/O addresses allow external devices to leave messages for the 6502. In the rolltop desk analogy, these cubbyholes have a trap door in the back, and some third party is responsible for the numbers that appear there.

The 6502 looks at the slip of paper in cell number 56,320 marked "PORT A" when it needs to know what's happening with the first joystick. Address 56,320, $DC00, is inside a chip called the 6526 CIA. (complex interface adapter). The CIA acts as a window, or port, through which external activities can be observed and influenced.

Divide and Conquer

A useful way to organize 65,536 (*64K*, where $1\ K = 2^{10} = 1{,}024$) memory locations is to group them into 256 "pages" of 256 locations each. Think of memory as a book with 256 pages, and 256 words (bytes) on each page. Page 3 is locations 768–1023, or $300 - $3FF. The page concept is a natural for hex representation, as every address breaks neatly into a *page* and a *location* within the page. Memory cell $3411 is the $11th byte of page $34.

Just because the 6502 has the potential to access 65,536 memory locations doesn't mean that every 6502 in the world can count on having that many locations available to it. The design engineer who wants to use a 6502 as the brains of a microwave oven, may decide that he doesn't need more than 1,000 bytes of ROM and 100 bytes of RAM to build the world's smartest microwave oven.

The 6502 that finds itself installed in such a microwave still has the capacity to access 65,536 locations, but only a thousand are really there. If it tries to access one of the unimplemented addresses, it's like a robot

in a Toyota factory blindly trying to arc-weld a Corolla stalled 10 feet up the assembly line. It *thinks* it's reading an instruction at location $C000, but it's seeing random, arbitrary garbage.

So what locations have what on the C-64? The 6502 programmer has to know, lest he try to store his data in ROM. The memory map is a useful tool for seeing at a glance the basic layout of the 64K addressing range.

Commodore 64 Memory Map

```
Page Number

Decimal  Hex          Function
-----------------------------------------------------
  255    FF
   .             Kernal (ROM-8K)
   .
  224    E0
   .             I/O Addresses (4K)
   .
  208    D0
   .             Utility RAM (4K)
   .
  192    C0
   .
   .             Basic Interpreter (ROM-8K)
  160    A0
   .
   .
   .
   .
   .             Basic Program Area (RAM-39K)
   .
   .
   .
   .
   8     08
   .             Screen Memory (RAM-1K)
   4     04
   .             Basic & Kernal Work Area (RAM-1K)
   0     00
-----------------------------------------------------
```

I thought my 64 had 64K RAM

It does, even though adding up the blocks of RAM in this memory map will only total 45K. Commodore got sneaky and piled ROM and I/O addresses "on top" of certain RAM areas. Cell $A037 is stored in a ROM chip by default, although it is possible to turn off the ROM at this address so that the underlying RAM cells in the range $A000–$BFFF are turned on. The technique of turning off an area of memory while turning on another is called *bank selecting*. If we didn't bank select, when the CPU tried to read one of these shared addresses, there'd be a fight on the data bus as RAM tried to say one thing and ROM something else.

A Trip Through the Memory Map, From Bottom Up

The nature of the processor forces designers of 6502-based computers to populate the lowest memory locations with RAM, for important system functions. Soon we'll learn why.

Screen Memory

The C–64 uses so-called "memory mapped" video, a modern technique that is both flexible and inexpensive. The memory range $400–$7FF leads a double life. It is part of the 6502's memory map, and appears to the CPU to be garden variety RAM. It is simultaneously an important input for the C–64's video circuitry, that part of the machine that decides what dots to light up and what dots to leave dark on your screen.

Basic

Next comes 39K of RAM, normally used for Basic programs and data. This is the 39K the C–64 is referring to when you turn it on ("38911 BASIC BYTES FREE"). Basic itself (an 8K machine language program) comes next, stored in ROM. From $C000 through $CFFF are 4,096 "utility" RAM locations, often used to store machine language routines to be used with Basic programs.

At the Top

$D000-DFFF are I/O addresses, locations that allow the 6502 to communicate with the rest of the computer, and consequently with human beings. Included in this range are two CIA's with 16 ports each, a 6581

Sound Interface Device (SID), with numerous control locations for sound generation, and a 6566 Video Interface Controller (VIC) for managing the display. VIC is the heart of the C-64's video mechanism; he looks at display memory and figures out what to put on the screen. Acronymically speaking, C-64 I/O is simply two CIA's, a VIC and a SID.

The top 8K addresses are ROM, and contain a series of utility programs that perform various odd jobs such as reading the keyboard and positioning the cursor, called the Kernal.

Mass Storage

65,536 bytes is a lot of RAM, but sometimes not enough. The 1541 disk drive can store 170,000 eight bit numbers on one disk—and we can have as many individual disks as we can afford. Ironically, the 1541 is a computer in its own right; it has its own 6502, which runs a boring little program that has it listening all day for commands from the "boss" computer, and fulfilling requests to read or write information.

The connection between C-64 and disk drive is a serial (one bit at a time) interface, with an effective transfer rate of about 350 bytes per second. Once the main CPU gets the data from the disk drive into RAM, it can deal with these numbers in the normal, fast way. Like ROM, a disk retains its data when power is removed.

But How Does It Work?

The movie *TRON* notwithstanding, the world of the 6502 is as far removed from human experience as anything could possibly be, more like the whirling cams and levers of a bottle capping machine, than men in funny hats playing catch with luminous Frisbees. Even so, an analogy relating the 6502 to the actions of human beings is a good way to explain how machine language works.

Consider, if you will, the loading dock of Giant Metropolitan Software Publishing House, Inc. Delivery trucks move ponderously in and out of loading bays. Workers with dollies and fork lifts move refrigerator-sized cartons of blank disks coming in and completed programs and manuals going out.

The undisputed boss of the dock is the foreman, an imposing figure in sky blue jump suit and orange Astros cap, directing workmen to and fro, signing paperwork, glancing occasionally at a clipboard in his left hand.

He runs things tight, by The Book. The Book is a much worn spiral notebook of maybe 150 pages chained to his desk. The label on the torn cover, although now illegible from years of use, once said: "SHIPPING DOCK PROCEDURES MANUAL". Each page is numbered. Some pages have only two or three lines on them, others, 10 or 15. Page 12, for example, says, in careful lettering:

```
PROCEDURE 12  UPB  -- UPS BLUE SHIPMENT

STEPS:

1.  PACKING LIST FOLLOWS WORK ORDER
2.  AFFIX BLUE LABEL STICKER
3.  LEAVE AT UPS AREA
4.  DONE
```

Every morning the foreman finds a thick stack of work orders waiting in his in basket. Each has some inter-office mumbo jumbo on it, and, in the upper left hand corner, the all important shipping dock procedure number. Not all of the sheets in the stack are work orders; most need the sheet or two of paperwork with them to be complete.

The dock foreman's most important tool is his green work order clipboard. After his morning coffee he takes the first one from the stack and clips it to the clipboard. As long as that work order is on the clipboard, he devotes his energies totally to performing the operations required to fulfill it.

There's a bunch of writing on each work order, but he's only interested in the procedure number. Today's first one has a procedure number of 22. "TPC", he mumbles to himself as he flips to page 22 of the procedure manual. (He knows 22 by heart, but turns to the page anyway. He's that kind of man.)

```
PROCEDURE 22  TPC  -- TEXPAK C.O.D

STEPS:

1.  PACKING LIST FOLLOWS WORK ORDER
2.  CALL TEXPAK FOR PICKUP
3.  FILL OUT C.O.D. FORM
4.  MOVE PACKAGE TO SHIPPING AREA
5.  DONE
```

When he finishes the last step of TPC, if it takes five minutes, or 15, he comes back to his desk, unclips the old work order and puts it and the packing list that went with it face down in the OUT basket. Without a pause, he takes the next work order from the In box, tacks it to the clipboard, and goes to work on it. All day long: Get a work order. Look up procedure. Perform the work order. Get a work order. Look up procedure...

A 6502 runs the same way: Access a memory location. Decode the contents of that location. The instruction may require the next byte or two in memory for execution. Execute the command, and proceed to the next memory location for the next instruction.

The Fantastic Voyage

Remember the movie *Fantastic Voyage*? Where some intrepid scientist/military types are shrunk to the size of a microbe to assist in the removal of a tumor from a valuable (I *guess!*) scientist's brain?

Through the magic of the printed word, we're going to do the same thing. Only without Raquel Welch in the crew. Take that back—she can come too. We climb into our manta ray shaped submarine and buckle up. Brace yourself. Soldiers are blasting us with strange violet light. We're shrinking. We're getting smaller. . .smaller. . .smaller. . .

We're so tiny now that the period at the end of this sentence looks like the Astrodome. We lift off (this submarine can fly, too) and head straight for a nearby C-64. It looks as big as Mount St. Helens. We slip easily through a crack in the keyboard, into a bizarre, alien landscape of ribbon cables and clock crystals. We drift eerily for what seems hours. Suddenly, dead ahead is a huge black monolith. The objective of our mission: *the 6502 microprocessor*. . .

4.
Getting Started

It's time to get acquainted with program portion of The Visible Computer. If you have anything plugged into the expansion port, remove it. TVC will not run with any other software resident in the computer. Turn on the monitor (or TV), disk drive, and computer, in that order.

Insert the TVC disk, and run it by entering:

LOAD "TVC",8,1

If the Software Masters copyright message does not appear within 10 seconds, repeat these steps. If you have followed instructions exactly, and still can't get the program to run, contact your dealer or Software Masters for assistance.

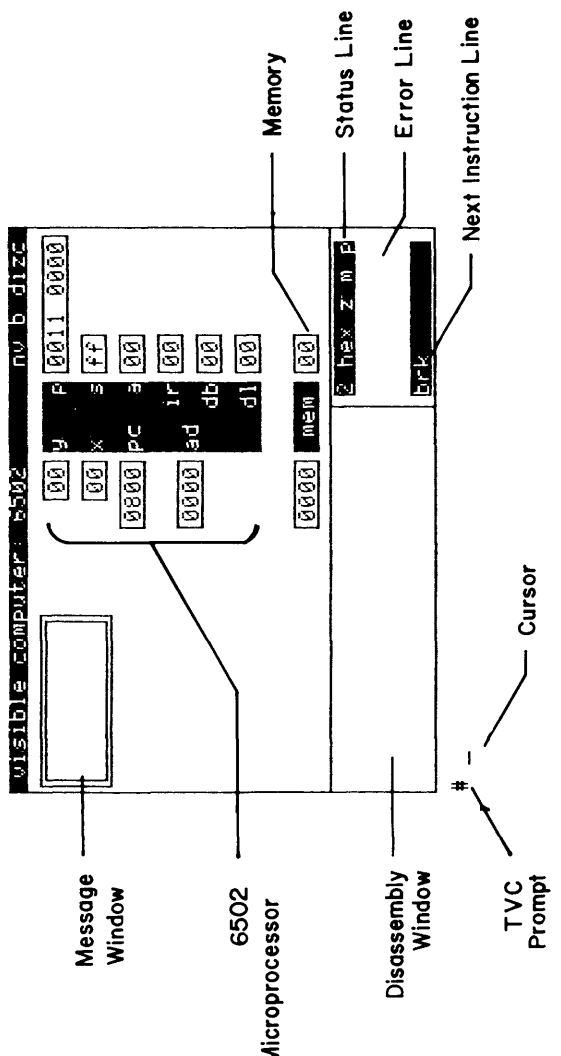
The copyright screen will remain for several seconds, after which TVC will begin to load into memory. In about 90 seconds, the main display will appear.

You're now looking at something that few have ever seen: the innards of a working microprocessor.

Each of the boxes holding a hex number is a *register*. A register is a place inside the processor where we can store binary numbers, a lot like a memory location. The 6502 can perform marvelous feats with a few simple operations on the contents of these 10 registers. In the next chapter we'll begin to see how this is done.

Speaking of memory, the 6502's contact with its 65,536 address locations is via the two registers labeled "mem", for memory. The 16 bit register is for the address; the eight bit register is for the data stored at that address. During program execution, this is where numbers appear that are being stored in and retrieved from memory.

The message window in the upper left hand corner is where TVC outlines the steps it follows in executing 6502 instructions. When TVC is not actively running a program (now, for instance), this window is blank.



Visible Computer Display

We'll put off talking about the disassembly area until we learn what disassembly is. The status box in the lower right part of the screen displays various tidbits germane to TVC's execution. At the very bottom is the command line, where you'll enter commands to control TVC. The "#" (number sign) is the monitor prompt. It serves as a reminder that entries should be statements recognizable by TVC.

The line next to the prompt is the cursor, and, like the flashing block cursor of Basic, shows you where you are on the screen when typing.

The Visible Computer consists of two major parts: the *monitor* and the *6502 simulator*. The monitor controls ("monitors") the simulator. The simulator is the part that actually executes 6502 programs. Throughout this manual we will use phrases like "in the monitor" and "returning to the monitor." You are "in the monitor" when the prompt is at the bottom of the display. You are "in the simulator" whenever the prompt is *not* at the bottom of the screen.

Monitor Commands

TVC has more than 20 commands in its vocabulary. You tell it something, and, if what you told it is something it understands how to do, it'll do it.

Entering Commands

To issue a command, type your request and press return. If a command consists of more than one part, use a space between the parts to separate them.

To fix a mistake, use the delete key to erase the error, and retype. (The C-64 screen editor does not function within TVC, and attempting to use these techniques may clobber the display.) If TVC cannot understand your instruction, it will tell you so with error messages.

Commands have the general form:

COMMAND [argument1] [argument2]

Argument is a 25 dollar computer word that means "modifier". Some TVC commands need no arguments; others need additional information to be complete, just as some Basic commands ("END", "NEW") stand alone, while others ("IF", "GOSUB") don't make sense unless you include more information.

Examples of one word monitor commands are ERASE and RESTORE.

The monitor command BASE (change a register's base to hex, binary, or decimal) needs two arguments; one to indicate the thing we're changing the base of, and another to specify the new base. BASE PC BIN changes the display mode of the program counter to binary.

You *must* separate a command and its arguments by one or more spaces. You *must not* use spaces *within* a command or argument. For example, if you entered the ERASE command as ER ASE, the command interpreter of TVC would understand it to mean: "Perform ER using argument ASE". Which, upon trying to find command ER, produces an error. If you can't get a command to work, check your syntax, and don't forget the spaces in commands that have arguments.

Now to get our feet wet with a couple of commands. First, we'll call up the calculator and see if two plus two equals four. I know that's a question of great concern to many of you.

Type CALC and press return. (Typing an instruction and pressing the return key is called "entering" in this manual.) If you make a mistake, back up with delete and fix it. If "Command" appears in the status window, TVC could not understand what you entered.

Eventually you should see the following on the command line:

<HEX><CALC> 00

The "HEX" means that the current calculator base is hex. All numbers produced by the calculator will be displayed in hex (*without* dollar signs), and all numbers you enter must contain only characters valid in hex. In other words, no hex numbers like G3#B, or decimal numbers like FC3. Do not include dollar signs; if the calculator's base is hex, the dollar sign is understood. The calculator base can be changed to binary by typing a Ctrl-B (Type Ctrl-B by holding down the CTRL key and pressing B), and a Ctrl-D for decimal. For now leave it in hex (Ctrl-H). Enter:

2 + 2

2 + 2 is replaced by 04. If you didn't get four for an answer, make sure you include the spaces between the two's and the plus sign. Now try these problems:

3 + 3
4 * 4
6 / 2
3EA * C (Try that on your Casio!)

To convert between bases, follow these steps. Converting 65,000 decimal to hex:

Ctrl-D
65000 <return>
Ctrl-H

Convert it to binary with Ctrl-B, and back to decimal with Ctrl-D. With the base decimal, try adding FF to 3. The BASE error that results is TVC telling you that you have entered characters not valid in the current base. FF is not a valid decimal number.

Answers are displayed with leading zeros, and for binary numbers, with spaces separating each nibble. This calculator has certain properties that make it undesirable for everyday checkbook balancing and miles per gallon calculations. It is an integer calculator; numbers with decimal points are not allowed as input. Divisions produce truncated (chopped off, as opposed to rounded off) results. (e.g., 6 / 2 = 3; 5 / 2 = 2; 9 / 10 = 0)

You may not enter negative numbers. Dashes entered anyplace except as the operator are treated as invalid characters. If you do a subtraction that produces a negative number, by subtracting a larger number from a smaller number, the answer will be displayed in *two's complement* form (later we'll learn what that is). Lastly, you may not enter, or produce via calculations, numbers greater than 65,535. These quirks are a result of the calculator's purpose in life, to help you write machine language programs.

When you've had all the fun you can stand changing numbers back and forth between bases, return to the monitor by pressing F1. Got the monitor prompt back? Next, try this short and sweet command:

ERASE

Wow. Spectacular. ERASE clears a space where you can experiment with the display. But since it's sad to see a lonesome little monitor prompt all by itself, bring the display back with the RESTORE com-

mand. If you like, you can issue these two commands over and over. For the more adventurous, let's move on.

Enter: WINDOW OPEN. Now we're erasing only a part of the display. When you know more about 6502 programming, you'll appreciate the choice of registers that remain onscreen when the window is open. Again, we don't want to leave our display looking so empty, so replace the part that got erased with WINDOW CLOSE.

What's So Great About Hex?

TVC defaults to a display mode of hex for all registers except P, but you don't have to leave it that way. Change the base of all the registers to binary with:

BASE ALL BIN

Change only the A register to decimal with BASE A DEC. Mix and match any combination of hex, binary and decimal. Get it looking like you want a 6502 to look. By the way, the names of memory's address and data registers are MEMA and MEMD, respectively.

This concludes our first session with TVC. We've learned what the monitor is, and have experimented with the commands CALC, ERASE, RESTORE, WINDOW, and BASE. In the next chapter we'll go in up to our knees and splash around a little.

5.
Working With Memory

TVC Memory Allocation

```
Page Number

  ---------------------------------
  FF
  ::  Kernal (8K)
  E0
  ::  I/O (4K)
  D0
  ::  Reserved for TVC (2K)
  C8
  ::  Screen Memory (1K)
  C4
  ::  Primary User Memory (1K)
  C0
  ::  Basic (8K)
  A0
  ::
  ::
  ::
  ::  Reserved for TVC (39K)
  ::
  ::
  ::
  ::
  ::
  ::
  02
  ::  Stack Page
  01
  ::  Page Zero
  00
  ---------------------------------
```

In this chapter we'll learn how TVC subdivides the C-64's memory map. Then we'll practice the monitor techniques of examining and changing the contents of memory.

Although you can read bytes from anywhere in memory, only the ranges $C000-$C7FF and $0000-$01FF can be written to. If you were

allowed to populate other areas with the numbers of your choice, you might hurt TVC; perhaps crash it, maybe just subtly alter a single function. TVC will appear to accept an order to place a value at $4003 (no error messages), but will not obey it. It handles ROM and I/O locations the same way. Later, when you've passed a few milestones in your studies, you'll learn a command that lets you to write addresses anywhere in the memory map. (Although you'll never have any luck writing to ROM. no matter how much you learn.)

The 1K primary user area ($C000–$C3FF) is plenty of room for most machine language programs.

A Window into Memory

Display the contents of 16 memory locations by entering:

WINDOW MEM

Unless you tell it otherwise with the LC or RC commands, TVC displays addresses $C000–$C007 and $0000–$0007 in the memory window. You can change the base of the memory window to decimal with BASE MEM DEC. BASE MEM HEX changes it back. If you want to change the value of a location, there are three options.

Direct Loads

The quickest way to write to a memory location is a direct load. Enter the address and the value you want stored there, separated by a space. Both address and value must be numbers valid in the current monitor base (the second entry on the TVC status line—hex, if you haven't changed it since startup). To put $CA in location $C006, enter:

C006 CA

See the contents of $C006 change? How nice. C004 3 writes a three into location $C004. To use decimal numbers, set the monitor base to decimal with BASE MON DEC. Now addresses and data must be given in decimal. 49158 4 places a four into $C006. We're going to be using hex almost exclusively, so change the monitor base back to hex (BASE MON HEX).

Remember memory mapping? How what's on the display is a function of a special area of memory called screen memory? We're going to ver-

ify that by writing the code for a lower case "g" into screen memory, and consequently, onto the display. TVC, for good reasons of its own, has moved screen memory from its normal place in the C-64 memory map ($400-$7FF), to $C400-$C7FF. It's organized the same way; each address is $C000 greater.

The screen is subdivided into twenty five rows of 40 character positions each, for a total of 1,000 characters. The shape that the VIC chip draws in the row 0, column 0 position (upper left hand corner) of your screen at any given moment is controlled by the number stored at location $C400. The bit pattern displayed just to the right of it, at row 0, column 1 is controlled by the value in location $C401. The last column position of the first row is at $C427. The first column of the second row is at $C428.

Let's put a "g" in the third column of the second line of the display. That's going to be memory location $C400 + (1*$28) + 3, or $C42B. The code for "g" happens to be $07.

Here goes:

```
C42B 7
```

Instantly the VIC chip responded to the new value and produced a different dot pattern in the corresponding part of the screen.

Direct loads are okay for a couple of quick writes, but if we want to write data in 50 consecutive locations, it's a lot of work specifying the address each time. A more efficient way to change several consecutive locations is the EDIT function. Invoke editing by entering:

```
EDIT C000
```

This causes editing to start at memory location $C000. To change location $C000, enter a number (naturally, an 8 bit number valid in the current monitor base). The number you enter replaces the previous value, in memory and onscreen. The address advances by one and the process repeats. There are a couple of tricks you can do with your first keystroke. Cursor left displays the contents of the previous location. Cursor right (or return) jumps to the next location, without changing the number stored at the first address.

Change the values stored at $C000 - $C007 to $11, $22, $33, $44, $55, $66, $77, and $88. Move forwards and backwards through this range

with the the cursor key. Do the values returned by EDIT agree with the memory window?

F1 cancels EDIT and returns you to the monitor.

If you write to a valid location not displayed in the window, the change is made in memory, but not onscreen. There are two commands which change the memory locations displayed, LC (change left column) and RC (change right column).

To list locations $FF00–$FF07, enter:

LC FF00

If you want, you can display the same locations in the right column. It's a free country.

Loading From Disk

The LOAD command loads memory with data stored on disk as either program or sequential files. The demonstration 6502 machine language programs we will be using shortly are loaded this way. You can also use LOAD to write zeros into your working area, to clean it up. This is done by LOADing a file on the TVC disk consisting of nothing but zeros, named, appropriately enough, ZEROS. Try it now.

LOAD ZEROS

This zeros addresses $C000–$C3FF, the TVC user area. The memory window should reflect the change. TVC cannot deal with file names that have spaces embedded within them, e.g., MY PROGRAM, because it uses spaces to separate arguments. None of the demonstration files on the TVC disk have embedded spaces, but keep this restriction in mind when loading your own programs later on.

As you might imagine, there is a counterpart to LOAD named SAVE. SAVE writes all or part of the user area to disk.

Loading Registers

Earlier we learned how to used the BASE command to change the display mode of a register. There's also a way to change a register's contents. Next to the monitor prompt, enter the name of the register and

the value you want to put there. As with all monitor commands, express the value in the current monitor base. To place $89F in PC, enter:

PC 89F

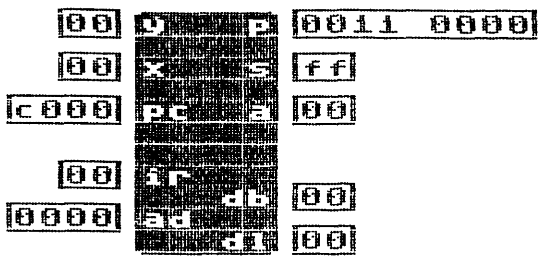
You may not write numbers larger than $FF into an eight bit register, or larger than $FFFF to a 16 bit register. Practice with it. Change the base of registers you've written numbers to. Do you get the same conversions you get on paper or with the calculator?

Appendix C is a reference on the TVC commands. Even though there are some we won't be using for some time, turn there now and quickly look through it.

6.
First Programs

Let's get a clean start for our first program. If you haven't just now loaded TVC, enter the RESTART command to put the 6502 simulator into its default state.

TVC is a program that makes your computer screen behave like a 6502 microprocessor. The simulator runs much more slowly than a real 6502, roughly a million times slower. Furthermore, it is transparent so we can see inside as it works. Behind the scenes, making the simulated 6502 tick, is your computer's real 6502.



A Tour of the 6502

The 6502 has eight 8 bit registers. A register, remember, is just a box where we can store binary numbers. Their abbreviated names: A, S, P, X, Y, DL, DB, and IR. There are two 16 bit registers, PC and AD. We'll discuss each register individually, as they have more personality than the typical memory location.

A The A register, or accumulator, although not especially large, is probably the most important register in a 6502. It gets a workout in almost every program.

S Directly above it is S, the "stack pointer" register. S is used for stack operations.

P The P register (Processor Status) holds the distinction of having probably the most unnatural abbreviation of all 6502 registers. It also is the only one that defaults to a binary dis-

play, because we are more interested in P's individual bits than their collective value.

X

Y

Next are the 6502's twins, the X and Y registers. They are virtually interchangeable, and get a lot of use, though not as much as A. They are often called *index* registers because of their use in something called indexed addressing.

PC

The big register beneath X is the program counter. It serves as a placemark to remind the 6502 where it is in memory and what instruction it should execute next. Fans of the program counter could make a good case for it being the most important register in the 6502. At the very least it's twice as big as the accumulator.

DL

DL, the data latch, is the 6502's bus station, the crossroads of data coming into and out of the processor to and from memory. No data comes into or leaves the 6502 without passing through this register.

DB

DB, the data buffer, is a place where we can temporarily shuffle a number off in the middle of an instruction until we're ready for it.

IR

IR is the instruction register. This is where a 6502 deposits the instruction currently being executed. It's the 6502's equivalent of the dock foreman's clipboard, a place where an instruction can be studied ("decoded") to figure out what it is and how to execute it.

AD

AD is the address latch, the register that holds the memory location to be accessed during reads and writes.

The 6502's 16 bit registers, AD and PC, have a dual personality; sometimes they behave like one big 16 bit register, other times like a pair of 8 bit registers. When used in this way, the high order halves are called PCH and ADH, and the low order halves, PCL and ADL.

A, S, P, X, Y, and PC are sacred abbreviations agreed on by all 6502 programmers. The other registers, DL, DB, IR, and AD, have more flexible names, as they were invented by the author of this manual. That's right. You could buy 11 books on 6502 machine language, and not one

would mention the DL, DB, IR, or AD registers. The reason is that a programmer doesn't have to worry about these registers to write 6502 programs. They're in every 6502, essential for running things, but since they perform temporary, scratch pad functions, the programmer need not concern himself with them. Since TVC simulates the inner workings of a 6502, we couldn't leave them out.

Now to put some of this knowledge into action. Let's load and execute the first of the demonstration programs, named, appropriately enough, PROG1. Load PROG1 (no space between the "G" and the "1") into memory, all two bytes of it, with the command:

LOAD PROG1

Unless you specify otherwise, LOAD loads data starting at address $C000, so PROG1 now resides in RAM beginning at $C000. PROG1 is a simple affair that will accomplish one small feat. It will cause a $33 (51 decimal, 0011 0011 binary) to appear in the accumulator. I know you could easily do that with the monitor command: A 33, but bear with me.

Let's look at the data that makes up PROG1. Put the window in memory mode with the WINDOW MEM command. PROG1 consists of the $A9 at $C000 and the $33 at $C001. Hmmmm. . . The program is going to put a $33 in the accumulator and one of the two bytes in the program is a $33. Could be a connection.

The zeros that follow mark PROG1's end. CLOSE the WINDOW. We want to see the whole processor/memory setup for our first program. Next, put TVC in its slowest, most helpful state with the command: STEP 3.

I know you're anxious to get started, but before we turn the simulator loose on PROG1, consider the current contents of the registers. Since we just booted TVC, the registers are in their default condition. Most, but not all, hold zeros. For now, don't worry about poorly abbreviated P and its binary contents, or the $FF in the stack pointer. I call your attention rather to the $C000 stored in the program counter.

$C000 is where in memory the 6502 that's about to come to life will find the instruction it will execute. It is no coincidence that LOAD placed PROG1 at $C000. If we were to make the program counter

something other than $C000, say $1AFF, the simulator would not execute PROG1, but rather whatever unknown data it found laying around at $1AFF.

To activate the simulator, press return without entering anything at the monitor prompt. Things happen fast now, so pay attention.

The Simulator

This is our first excursion into the 6502 simulator. It doesn't have nearly as many commands to worry about as the monitor. It executes programs while you watch. The message window displays "fetch". This means an instruction fetch, the first phase of instruction execution, is in progress. When the 6502 has fetched a byte and placed it in the instruction register, the fetch cycle ends, and the execution phase begins.

The second line of the message window contains a more cryptic message.

T: PC -- > AD

This translates into English as *Transfer: The contents of the Program Counter to the Address Latch*. This is a *microstep*, a building block of a machine language instruction. The nine TVC microsteps are listed in Appendix D. The transfer microstep, which blinks the source and then the destination register, is the most common, used by every instruction at least twice.

The transfer will occur as soon as you exit the pause you're in. Pressing "C" invokes the calculator, the only monitor function available from within the simulator. Exiting the calculator with F1 returns you to the pause.

Restart the simulator by pressing the spacebar. AD now contains $C000; note that PC is still $C000. A transfer doesn't affect the contents of the source register.

READ is the next microstep of the FETCH process. A read happens fast, so be ready. It consists of the following steps:

1. The contents of the address latch are transferred to memory's address bus.

2. Memory fetches the contents of that address and transfers it to memory's data bus.

3. That value is transmitted to the 6502's data latch.

Before we let this Read happen, a pop quiz: What value will be read from location $C000? Answer: $A9. It won't have changed from a minute ago when we looked at it from the monitor. Okay, press the spacebar.

The 6502 is still in the fetch cycle. It's taken a work order from the Inbox, but hasn't got it to the clipboard yet. Until it gets this byte to the instruction register (IR), the 6502 doesn't have any idea of what the instruction is, much less how to complete it. The next step, then, is to put the instruction in IR so we can get on with decoding and executing it. As soon as the $A9 is in IR, the fetch phase ends, and the *execute* phase begins. "Fetch" is replaced in the message window by "LDA IMMED", the 6502 saying to itself "I need to load my accumulator with the next byte in memory." A Load Accumulator, Immediate, also known as instruction number $A9.

And it knows what to do next. First: Increment the program counter. Now it contains $C001. Transfer it to the address bus. Do you feel a READ coming on? Yes. Read the contents of location $C001 into the data latch. Copy the number you found there, a $33 (big surprise), into the accumulator.

We're almost done. Something called "CONDition FLAGS" happens that blinks the P register. We'll talk about this phenomenon later. The last step is to increment the program counter. We do this not to assist in the execution of *this* instruction, but to prepare for the *next* one. When you're in the monitor, the program counter always points to the next instruction, not the end of the one just completed.

A real 6502 doesn't have the luxury of sitting around doing nothing while a monitor takes over for half an hour. It has to execute one instruction after another, bing-bing-bing, hundreds of thousands of times a second, without so much as a break to pat itself on the back. After every instruction, the value in the program counter is the address of the first byte of the *next* instruction.

That last increment of the program counter completed the instruction, and deposited us in the monitor. The last act of the simulator is to list

the instruction just performed in the disassembly window. For now, consider the disassembly window a mysterious trail of the last five instructions the simulator has executed.

What would happen if we were to enter the simulator now? Press return and find out. It'll fetch the $00 that's in $C002, put it in IR and digest it. $00 is a 6502 instruction called BRK (software break), that puts you right back in the monitor without doing anything. It's a signal to the simulator that the program is over and we want to get back to the monitor. Later we will learn more about this unique instruction.

Congratulations! You have just watched your first program. If you were able to follow along, you have learned about 90% of the fundamental basis of machine language. Don't worry about memorizing the exact sequence of microsteps. The important thing is that having the 6502 execute the sequence of bytes: $A9 $33 caused it to load the accumulator with $33.

Before you go on to the next session and more complex programs, be sure you understand how this one works. You can have an instant replay of PROG1 by setting the program counter back to $C000 with PC C000. While you're at it, why don't you change memory location $C001 from $33, to say, your age—then PROG1 can serve the useful purpose of telling the accumulator how old you are.

Moving Right Along

So far we know exactly two of the fifty-six 6502 instructions. LDA, also known as $A9: "load the accumulator with the byte following this one", and BRK, $00, "Break out of the program and return to the monitor". "LDA" and "BRK" are not haphazardly chosen abbreviations; they are official 6502 *mnemonics* (neh-*mon*-ics). A mnemonic is a memory aid, based on the theory that it's easier for human beings to associate "LDA" with the act of loading the accumulator than $A9. The 6502 has no idea, of course, what LDA means; if you want a 6502 to load its accumulator you have to give it the *opcode* $A9. Each 6502 instruction has a three letter mnemonic. Some of the abbreviations are better than others, but all are easier to remember than a number.

Remember me saying that the accumulator is the most important register on the 6502? That makes LDA-$A9 a good instruction to know. Loading is all well and good, but what about *storing* a value in the ac-

cumulator somewhere in memory? Is there a way to do that? You bet. LOAD PROG2.

PROG2 introduces the flip side of LDA, STA (Store Accumulator; opcode $85). This instruction causes the contents of the accumulator to be placed in the memory location of our choice. PROG2 will first LDA with $66, and then STA it at memory location $43 (a page zero address). PROG2 is longer than PROG1, a whopping two instructions, four bytes. Take a look at it with either WINDOW MEM or EDIT. It begins, as did PROG1, at $C000. Notice the $43 at $C003. A coincidence? You know better.

With the program counter set back to $C000, and with WINDOW CLOSE and STEP 3 in effect, step your way through this two instruction program. PROG2 will be only half done when you first return to the monitor, since we return to the monitor after completing each full instruction. Press return to execute the second instruction. Pay special attention to STA-$85. STA is a tad more involved than LDA-$A9.

When the 6502 sees an $85 in the clipboard register, it knows it must get one more byte out of memory, just as it did with LDA. But what it does with the second byte (a $43) is different.

First it transfers it to the address bus. Since AD is 16 bits wide, and we're loading it with an eight bit number, the most significant byte becomes zero. We have now formed the zero page address $0043. Putting a number on the processor's address bus is always a precursor to a read or write of memory. Next, we transfer the accumulator to the crossroads register, DL. The stage is now set for the WRITE microstep.

A write consists of the following steps:

1. The contents of the address latch are transferred to memory's address bus.
2. The contents of the data latch are sent to memory's data bus.
3. Memory inserts the value into the selected location.

After the write, STA is complete except for a final increment of the program counter to make it point to the next instruction. No flag conditioning this time.

When you get back to the monitor, check the contents of memory loca-

tion $0043 with either WINDOW MEM (and an LC 40) or EDIT and verify that it really got the value PROG2 put there. Run this program several times. Use different values for $C001 and $C003. Do *not* change the opcode values, the $A9 in $C000 or the $85 in $C002. Change them and you change the instruction from LDA to who knows what.

That's three instructions down, 53 to go. But we're about to learn 10 more with astounding ease.

Loading and Storing Other Registers

The accumulator is top dog on the 6502, but occasionally we need to load and store some of the other registers. There are instructions for just that: LDY and STY for the Y register. LDX and STX for X.

MNEMONIC	OPCODE	OPERATION
LDX	$A2	Load X register
LDY	$A0	Load Y register
STX	$86	Store X register
STY	$84	Store Y register

PROG3 demonstrates all the instructions we've learned. LOAD it, set PC to $C000, and step through it. Each of the new instructions functions exactly like its accumulator counterpart. Now we're really starting to accomplish things; three consecutive memory locations loaded with $FF. Great.

All of the instructions so far have been two-byters: An opcode byte to give the 6502 its orders, and a second byte to use in completing the order. The 6502 also has one byte instructions, instructions so self explanatory they don't need a second byte to finish the job. Such instructions are said to be "implicit", or *implied.*

The six Transfers are representative of the Implied group of 6502 instructions. They're for moving data between registers.

MNEMONIC	OPCODE	OPERATION
TAX	$AA	Transfer A to X
TAY	$A8	Transfer A to Y
TXA	$8A	Transfer X to A
TYA	$98	Transfer Y to A
TXS	$9A	Transfer X to stack pointer
TSX	$BA	Transfer stack pointer to X

Don't confuse the 6502 transfer instructions with the "T:" microstep. A "T:" is only one phase of a 6502 transfer instruction like TYA, or for that matter, any 6502 instruction.

Two of the transfer series are demonstrated in PROG4. For now, take the others on faith. LOAD and STEP 3 your way through it. Notice that the 6502 knows it need not fetch any additional bytes out of memory after the instruction fetch to complete a transfer instruction. It knows what to transfer where by looking at the opcode byte.

Ultimately, PROG4 accomplishes the same function as PROG3 (the not-so-earth-shaking feat of writing $FF into locations $40, $41 and $42), but does it faster. It takes less time to execute a one byte transfer instruction than a two byte load instruction. It's also two bytes shorter.

Now we're making some progress; 13 instructions down, 43 to go.

7. Processor Status Register

The P (processor status) register is something of an oddity in the 6502 family. Not only does it have a confusing abbreviation, it is also the only register where we are more interested in contents on a bit rather than byte level. In other words, if both P and A happen to contain 0011 0011, we will usually interpret A as the number $33, and P as containing ones in positions 0, 1, 4, and 5, and zeros in positions 2, 3, 6, and 7. These bits are so important they have their own names. P defaults to binary display so that each bit falls under its abbreviation.

Speaking of defaults, why are bits 4 and 5 set? Because that's what you usually find in this register inside a real 6502 running in a C-64. The full names of these rugged individualists:

Position		Name
7	N	Negative flag
6	V	oVerflow flag
5		
4	B	Break flag
3	D	Decimal Mode flag
2	I	Interrupt Disable flag
1	Z	Zero flag
0	C	Carry flag

Two things: A "flag" is a bit of unusual importance. Bit 5 of the processor status register is not used. It's *there*, obviously, but we have no control over it, nor will we ever be interested in its value.

Instructions that Affect the P Register

There are implied (one byte) instructions to set and clear many, though not all, of the P register flags. "Set" and "clear" are handy verbs describing the act of forcing a bit to become either a one or a zero, respectively. "Reset" is used interchangeably with "clear" in this manual.

MNEMONIC	OPCODE	OPERATION
CLC	$18	Clear carry flag
CLD	$D8	Clear decimal mode indicator
CLI	$58	Clear interrupt disable indicator
CLV	$B8	Clear overflow flag
SEC	$38	Set carry flag
SED	$F8	Set decimal mode indicator
SEI	$78	Set interrupt disable indicator

This list of commands seems incomplete; there are no instructions for setting or clearing the negative and zero bits, and none for setting overflow. There don't need to be, as we shall see.

The Zero Flag: 6502 Historian

The Z flag contains a single binary fact about previously executed instructions. It is "conditioned" (set or cleared) by the 6502 every time it executes a load or transfer instruction. If you load a zero into the accumulator, Z will be set. This is backwards from common sense, so I repeat: If you load X, Y, or A with a zero ($00; 0000 0000), the Z bit will be set. It will stay set until such time as another load or transfer comes along that loads a nonzero value into a register. Once cleared, it will stay that way until the next zero load comes along to set it.

The Negative Flag

The N flag is also conditioned with every load and transfer instruction. If you load or transfer a number that has bit 7, the most significant bit, set, N will be set. Any 8 bit number greater than $7F has this bit set (check it out!). Conversely, loading or transferring values with this bit clear, will clear the N flag. N gets its name from the fact that frequently bit 7 is used to indicate negative numbers. We will describe the signed number situation in more detail later on, but quickly, the convention is: If bit 7 is set, the number is negative. If it is reset, the number is positive. If you are not using signed numbers, the behavior of the N flag can be disregarded.

This conditioning effect, in conjunction with instructions to be presented in the next chapter, allow the programmer to test conditions that existed on previous load and transfer instructions. The technique is to examine the state of the Z or N bits, and decide what to do next on the basis of that finding. This is related to Basic's IF / THEN statement.

Basic
IF A = 0 THEN GOTO 1000
A = A + 1
etc.

Machine Language
TXA
[If Accumulator = 0, GOTO XXXX]

In the next chapter we'll learn an instruction to fill in the brackets.

PROG5 demonstrates both the implied clear/set instructions and the conditioning effect of loads and transfers. LOAD PROG5, but before you run it, we're going to explore a feature of TVC for anticipating what a program will do without actually running it.

Disassembly: The L(ist) Command

With PROG5 LOADed, enter: C000 L. As with all monitor commands, separate the L from the C000 with a space. What appears in the disassembly window is a sneak preview of the first five instructions in PROG5. Unlike the instructions put there by the simulator after executing an instruction, the address is not in inverse video.

```
C000: 38      SEC
C001: F8      SED
C002: 78      SEI
C003: 18      CLC
C004: D8      CLD
```

Disassembly is an awkward word for the extremely useful process of presenting a machine language program in a form more palatable than plain hex. The hex is there too, address and contents—but the humanized version of the instruction is what we're after.

A disassembled instruction usually contains two parts: mnemonic and *operand* (implied instructions, e.g., DEX, are operand free). In "LDA # $33", LDA is the mnemonic, and #$33 the operand. Both assist, sometimes subtly, the programmer in determining what the instruction does.

You can disassemble anywhere in the memory map, but don't expect reasonable results unless the numbers stored there are actually machine language programs. Enter: 500 L. Address $500 is part of TVC itself. TVC is largely a Basic program, and Basic programs do not directly make sense to either a disassembler or a 6502. Another way to go wrong is to start disassembly in the wrong place. Even though PROG5 sitting there at $C000 is a valid machine language program, if you start listing it at $C00A, you'll get question marks because the first byte (the data byte of the LDX instruction that starts at $C009) is an undefined opcode.

A tipoff that you're disassembling something other than machine language is the appearance of "???" mnemonics. Of the 256 possible eight bit values that could have been 6502 opcodes, the designers of the 6502 gave only 151 defined meanings. Give the disassembler one of these unimplemented opcodes, such as $02, or $FF, and it says: Huh? Probe around inside ranges known to contain machine language, such as the Kernal ($E000–$FFFF). Although even the Kernal area is not solid machine language, you'll see many marvelous mnemonics and operands, including many we haven't learned yet.

The "next instruction line" of the status area, if you haven't already guessed, holds the disassembled form of the instruction that is either about to be executed (if you are in the monitor) or is currently being executed (if you are in the simulator). Minus the address (which is defined to be the program counter, anyway), and the hex values themselves.

The next instruction display changes whenever the program counter, or memory pointed at by the program counter, is changed. Change the program counter to $C100. The instruction line should have changed. Now write $18 to $C100. Again it should have changed. We're about to run PROG5, so put PC back to $C000.

Disassembling a machine language program is not the same as executing it, any more than listing a Basic program is the same as running one. Although now PROG5 will be anticlimactic, having already seen what instructions are in it, work your way through it with the simulator. Those of you with printers can have a little extra fun by activating the output-simulator-disassembly-to-printer feature of TVC with the command:

PRINTER ON

TVC will now output to the printer one line of disassembly after every instruction executed by the simulator. For further information on the printer command, consult Appendix C.

PROG5's Set/Clear instructions are straightforward enough, but pay special attention to the COND FLAGS microstep of the loads and stores that follow. Each load conditions the N and Z flags. If we load a register with a zero, the 6502 will set Z. If it was already set, it'll stay set. N is altered at the same moment. It will be set whenever a load occurs that sets bit 7 of the register that is loaded, and reset when bit 7 is reset. For now, just observe the conditioning process and don't worry about why it goes to this trouble.

Tinker around with the data portion of the load instructions. What do the Z and N flags do with a load of $FF? Or $31? Find out, and meet me at the start of the next chapter.

8.
Branches: Decision Making

If you're like me, the first Basic program you ever saw didn't do much for you. It probably went something like this:

```
100 INPUT "WHAT IS YOUR NAME ";A$
110 PRINT "THAT'S A NICE NAME, ";A$
120 END
```

Unless you were exceptionally creative with your input (THAT'S A NICE NAME, GRAND CAYMAN ISLAND), it wore thin quickly. But my first encounter with testing and looping was almost a religious experience.

```
100 N = 0
110 PRINT N , N * N
120 N = N + 1
130 IF N <= 10 THEN 110
140 END
```

Somehow the concept of testing, and if necessary repeating, a series of instructions was fascinating: "Wow, I could change the 10 in line 130 to 1000. . . or 1000000 . . . Or change line 110 to print the cube root too!"

Put simply, decision making and looping is what programming computers is all about. This is as true for machine language as it is for Basic. To execute our first decision-and-loop 6502 program we'll need some new instructions: The Decrement/Increment series, and a Branch or two.

There are 4 implied instructions to increment (increase by one) and decrement (decrease by one) the contents of the X and Y registers. They are: DEX, DEY, INX, and INY.

MNEMONIC	OPCODE	OPERATION
DEX	$CA	Decrement X register
DEY	$88	Decrement Y register
INX	$E8	Increment X register
INY	$C8	Increment Y register

These instructions have a "wraparound" effect. If you decrement a register that contains $00, it goes to $FF. If you increment a register that contains $FF, it goes to $00. There is also a way to increment and decrement memory locations. Strangely enough, there isn't an inc/dec pair for the accumulator, although there is a way to accomplish the same thing.

Like loads and transfers, these instructions condition the N and Z flags. If we execute DEX at a moment when the X register contains $01, we get $00 in X, a set Z flag, and a clear N flag. This makes the inc/dec instructions useful in counting loops. Load the X (or Y) register with the number of times you want the loop to occur. Next, do the operation(s) you intend to repeat. Now decrement X to reflect that you've been through the loop one time. Last comes something that can both test the Z bit, to see if X has been reduced to zero yet, and depending on the result of the test, cause us to jump back and repeat the process again.

These conditions are met by the BNE instruction. Pronounced "Branch if Not Equal", with an opcode of $D0, this instruction is the equivalent of the Basic statement:

IF A <> 0 THEN GOTO 1000

BNE is one member of the branch family of 6502 instructions. There are seven others, two for each of the four testable flags of the P register (C, N, Z, and V). One that tests for the desired bit set, another for the same bit clear.

MNEMONIC	OPCODE	OPERATION
BCC	$90	Branch on carry clear
BCS	$B0	Branch on carry set
BEQ	$F0	Branch on result = 0 (Z Set)
BNE	$D0	Branch on result ≠ 0 (Z Clear)
BMI	$30	Branch on result minus (N Set)
BPL	$10	Branch on result plus (N Clear)
BVC	$50	Branch on overflow clear
BVS	$70	Branch on overflow set

Branches are said to use *relative* addressing because of the way they are executed. A branch instruction is two bytes long; an opcode byte (which tells the 6502 what bit to test, and for what value), and a second, *offset* byte to tell it where to go if the test passes. This "telling it

where to go'' is tricky, and has to do with why their addressing form is called relative.

If the condition specified by the test is true, then the second byte is used to calculate a new value for the program counter. The program counter, remember, is the placemark in memory that keeps the 6502 executing instructions in sequence. If the test fails (as it would for a BNE when the Z bit is set), the program counter advances normally by one and things proceed as if there had been no branch instruction at all.

If the test succeeds, the program counter is modified by having the second byte *added to it*. For example, if PC contained $C005 (having just read from memory the second byte, say a $10, of a BNE instruction), and the 6502 determines that the test has passed, it forms the new PC by adding $10 to the $C005 already there. The next instruction to be executed would be the one at $C015 ($C005 + $10).

Does this mean that branches can only happen in the forward direction? No, negative branches are possible, although understanding how a negative branch is calculated is a little more difficult. If the data byte of the branch instruction is $80 or greater (Hint: bit 7, the sign bit, set), the 6502 knows to do a *subtraction* on the program counter rather than an addition. We will leave the details of this subtraction until a later section. (Sneak preview: $FF = -1, $FE = -2, $FD = -3...) Branches can go either way, depending on the data byte, by making the program counter either larger or smaller. We may branch about 128 bytes in either direction.

Branching is demonstrated in PROG6. LOAD and disassemble (L) it. It begins by loading X with $04; we are evidently intending to do something four times. Next are two set/clear instructions, there only to give the program some busy work to do in the loop. Next comes the new instruction DEX. DEX conditions the Z flag; if it didn't, this program wouldn't work. The branch instruction BNE consists of an opcode byte ($D0) at $C005 and an offset ($FB) at $C006. $FB, being greater than $80 has bit 7 set, and therefore is a negative branch; the program counter will be reduced some amount if the BNE test passes.

```
C000: A2 04    LDX  #$04
C002: 38       SEC
C003: 18       CLC
C004: CA       DEX
C005: D0 FB    BNE  $C002
```

The TVC disassembler goes out of its way to help you understand where the branch will end up if the test passes. BNE $C002 means "Branch if not equal to location $C002". This is friendlier than saying BNE $FB, leaving you to figure out where the branch will go.

The first time we encounter the DEX instruction, X will be reduced to $03. This is nonzero, so Z will be cleared and the branch test will succeed, causing the loop to repeat. Finally, after four repetitions, the test fails and BRK ends the program.

Since this program is significantly longer in execution time (though not in length) than previous programs, now is a good time to learn how to control the speed of the simulator. We've been using Step 3 exclusively. What do the other step values do?

Step 2 executes an entire instruction without pausing at each microstep. You can force the simulator to pause by pressing the spacebar. When the instruction is over, you are returned to the monitor.

Step 1 is similar to Step 2, except that you don't enter the monitor between instructions, but instead plunge ahead with the next instruction. F1 forces the simulator to enter the monitor at the completion of the current instruction.

Step 0 is TVC's flat out, high speed mode. It saves time by skipping the process of writing to the screen during execution. The only things updated are the disassembly window and the next instruction area. The registers will not reflect their true values until you return to the monitor. "Flat out" and "high speed" are relative terms. "Flat out", TVC operates at something on the order of one millionth as fast as a real 6502.

If you are in step modes 1, 2, or 3, you can slow down or speed up the action by typing one of the number keys (1–9), while the simulator is running. 1 is fastest, 9 slowest. Now press return, and happy looping.

9. Addressing Modes

We've moved so quickly that we've glossed over some very good questions you might have had. One being: "If there are only 56 instructions, why are there 151 opcodes?" The answer is tied up in something called *addressing modes.*

The 6502 is good at addressing modes; in fact, it makes some of its contemporaries (like the Z-80) look positively anemic in this regard. In a nutshell, addressing modes determine not what instruction to perform, but where to get the data the instruction will use. So far the demonstration programs have worked with a small subset of the many addressing modes available on the 6502. All loads have used *immediate* addressing, the form that tells the 6502 to load a register with the next byte following in memory. All stores have used *zero page* form, which specifies a memory location on page zero.

What if we wanted to load the accumulator, not with a number that we knew ahead of time when the program was written, but with the contents of a memory location outside the program. The Basic statement:

LET A = 14

is the equivalent of the way we've loaded the accumulator so far. More common in Basic is the statement:

LET A = B

Accomplishing this in 6502 machine language requires a LDA of a different color. There is another opcode that decodes as LDA, but not the LDA-$A9 that makes the load occur from the next byte. It's LDA-$A5, and it makes the load occur from *the memory location specified in the next byte*. This is a slippery idea, I'll admit, but crucial to your future happiness as a world famous machine language programmer.

PROG7, another two-byte special, will clear up the mystery. Load and list it. Notice that the disassembly is not quite identical to that for PROG1.

```
PROG1

C000:A9 33    LDA #$33

PROG7

C000:A5 33    LDA $33
```

Opcodes $A5 and $A9 cause the disassembler to produce the same mnemonic, LDA, but different operands. The "#" is your clue to understanding what kind of LDA you've got. By 6502 convention, a number sign in the operand means that the value to load is "immediate", contained in the byte occurring next in memory. The absence of the number sign in the second instruction tells us that the load will occur from the memory location specified in the operand, in this case from location $0033.

We just learned a new opcode, $A5, but not a new instruction. $A5 is LDA using the *zero page* addressing mode. $A9 is LDA using *immediate* addressing. Now execute PROG7. Pay close attention to how it gets $0033 into AD. Similar to the STA $33 instruction of PROG2.

What, there's more? Now a third way to LDA. Some of you have been asking: "What if I want to load the accumulator with a value stored somewhere in memory, but not a location down in page zero? Say an address like $A09 or $BFFF?"

Very good question. And yes, there is a way to do it. You may specify any of the 65,536 locations using *absolute addressing*. An instruction using absolute addressing requires three bytes: an opcode byte, and two bytes that specify the memory location the operation is to use.

Load PROG8 and list it. PROG8 will load the accumulator from $B1C, when we let it, which we will in just a second. First, a close examination of the disassembly.

```
C000:AD 1C 0B LDA $0B1C
```

Notice that the *least significant byte of the address comes first*. 6502 convention is to store two byte values in sequential memory locations with the least significant byte stored first (lowest address). There's no special reason for this; they just adopted a convention and stuck with it. Again we find the disassembler working hard to make life easier for us. It rearranges the operand into normal left-to-right form. It's a lot easier to grasp the meaning of "LDA $0B1C" than "AD 1C 0B".

Now execute PROG8. As you might expect, it takes longer to run than the other forms of LDA we've used. The data buffer is used to temporarily store the first byte of the address until we're ready for it. However, except for the extra memory fetch and transfer to the address bus, it runs exactly like the other two, finishing up with a flag conditioning and a final increment of the program counter.

So there you have it. One instruction, LDA, and three different opcodes ($A9 for immediate; $A5 for zero page; $AD for absolute). Can we use absolute addressing to access zero page locations? Yes, you may. There is no rule against the instruction:

```
C000:AD 12 00 LDA $0012.
```

Then why is there a zero page addressing mode at all? Because only two bytes are needed instead of three. Absolute addressing takes more storage and more time to execute. For efficiency, 6502 programmers place their most frequently accessed variables in page zero. As a result, the zero page is prime real estate in the 6502 memory map. Although in theory you can use page zero for program storage, this is rarely done; it would be like using a square block in downtown Chicago to grow tomatoes.

There are only 256 locations, and everybody wants to use them. If you're writing a machine language program that will be called from Basic, you'll have to be careful to use zero page locations that Basic and the Kernal routines *don't* use. To determine what locations are safe, consult the table on pages 310–316 of the *Commodore 64 Programmer's Reference Guide*. Four known safe addresses are $FB-FE.

If you don't have a copy of the *Programmer's Reference Guide*, get one. This 500 page book can be found at most bookstores and is full of facts you'll need when writing 6502 programs for the C–64.

The load and store instructions of the index registers have these addressing modes also. This table summarizes the opcodes for all three addressing modes for LDA, STA, LDX, STX, LDY, and STY.

INSTRUCTION	ADDRESSING MODE			OPERATION
	ABS	IMM	ZP	
LDA	$AD	$A9	$A5	Load accumulator
STA	$8D		$85	Store accumulator
LDX	$AE	$A2	$A6	Load X register
STX	$8E		$86	Store X register
LDY	$AC	$A0	$A4	Load Y register
STY	$8C		$84	Store Y register

There aren't any immediate addressing stores, because a store immediate doesn't make sense. That's like saying LET14 = A in Basic.

A final ominous word before we move on to more jumping around fun in the next chapter. I said earlier that the 6502 is a champion at addressing modes. You don't get to be a champion having just three modes for a popular instruction like LDA. You get to be a champion by having *eight*.

10. Subroutines: The Stack

Next on the agenda are three instructions that, like a successful branch, alter program flow by changing the program counter. Unlike the branches, the 6502 has no choice in the matter.

The new instructions are: JuMP (JMP, $4C), Jump to SubRoutine (JSR, $20) and ReTurn from Subroutine (RTS, $60). All three have direct counterparts in Basic.

MNEMONIC	OPCODE	OPERATION
JMP	$4C	Jump to new address (Basic GOTO)
JSR	$20	Jump to subroutine (Basic GOSUB)
RTS	$60	Return from subroutine (Basic RETURN)

JMP is a three byte, absolute instruction that puts the address of our choice in the program counter, thus shuffling us off to wherever in memory we've got instructions that need executing. As with all absolute instructions, the destination address is stored in memory with the least significant byte first. One use for JMP is to extend the range of a branch. A branch on its own is limited to about 128 bytes in either direction. If you use a branch in combination with a JMP, you can go as far as you want.

Instead of:

```
C010: BEQ $F000 (can't branch that far)
C012: ETC...
```

Use:

```
C010: BNE $C015
C012: JMP $F000
C015: ETC...
```

Load PROG9 and list it. PROG9 is full of jumps—six of them, to be exact. But the disassembly lists just the first one. If you want the disassembler to show you what's out there waiting at $C100 after the first jump, you have to ask for it specifically.

Now execute it. What you have at the end of PROG9 is an *infinite loop*. Like a cat chasing its tail, this program will never go anywhere. Although not a problem when we're executing programs with a simu-

lator that lets us quit with a press of F1, it can be a serious problem under real 6502 execution.

A different sort of jump is controlled by the JSR/RTS pair. They're used like the GOSUB/RETURN combination of Basic. In fact, most every programming language has some way to implement this concept.

Executing a 3 byte (absolute) JSR instruction will, just like a JMP instruction, divert program flow to the address contained in the operand portion of the instruction. But with an important difference: before it goes to the new address, the 6502 *saves where it is now*, by storing the current contents of the program counter in memory. This enables the 6502 to find its way back when it finishes the subroutine.

How JSR and RTS Work

Even though it is not strictly necessary to understand the underlying mechanics of the JSR/RTS pair to use them, I'm not going to let you off that easy. That's okay for Basic programmers, to accept a gift without worrying about where it came from. Machine language programmers look every gift horse square in the mouth to see the pitfalls lurking there.

JSR and RTS use something called the *stack* to accomplish the feat of returning after a subroutine has been completed. The 6502 stack is two things, working together: the stack pointer register (S), and $100 bytes of memory ranging from $100 - $1FF, the stack page. Although there is nothing to stop the machine language programmer from using the stack page of memory for general purpose program and data storage, it is *strongly* recommended that you reserve this area for the stack. With freedom comes responsibility.

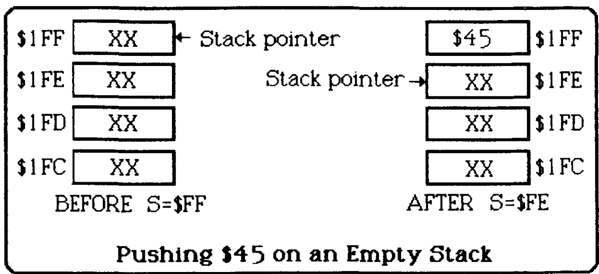
The Classic Cafeteria Tray Analogy

The stack can be visualized as a stack of trays in a spring loaded container at the beginning of a cafeteria line. The tray at the top, ready to be pulled off next is the one most recently entered. The one at the bottom may have been there since Mother's Day. This is called a LIFO data structure, for Last In, First Out. As opposed the serving line, which is a FIFO (First In, First Out) data structure, also known as a *queue*.

If we put two green trays on a stack of red ones, we know that the next two trays pulled off will be green. To implement the stack for useful

purposes of storage, we need only two operations: Push (put a tray on the stack) and Pull (take a tray off the stack). We don't care if there are 50 trays or 15 when when we issue a Pull command, only that we get the one most recently put there. If I push a $45 ($45 written on a tray with a magic marker) onto the stack, and then an $FF, when I turn around and execute a pull, I'll get the $FF tray back first.

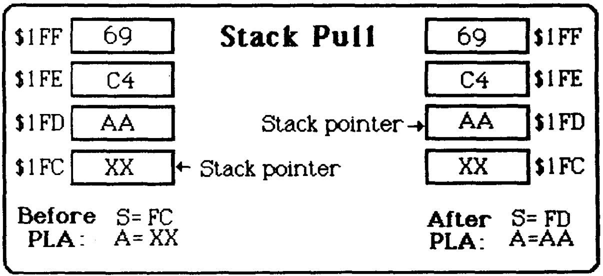
How is a one byte register and $100 memory locations like a cafeteria? The trays are one byte numbers that the 6502 pushes and pulls. The holder is the stack page—but instead of moving all 256 bytes down one every time we push a value on the stack, the only thing that moves is the contents of the stack pointer. The stack pointer always points to the most recent entry in the stack minus one. If S contains $FF, and we execute a push, the value we push winds up stored at $1FF, and S is decremented to $FE. The first position in the stack is $1FF, and subsequent entries (i.e., more recent ones) use successively lower memory locations.



Pushing $45 on an Empty Stack

Microsteps of a Push

1. Transfer stack pointer to ADL. ADH = 1 (for stack operations, ADH is "hardwired" to 1 to force address references to be in the stack page)
2. Transfer register to be stored to data latch
3. Write
4. Decrement stack pointer



Microsteps of a Pull

1. Increment stack pointer
2. Transfer stack pointer to ADL. ADH = 1
3. Read
4. Transfer data latch to selected register

By convention, the stack pointer always points to the first vacant space in the stack. A Pull therefore increments the stack pointer *before* the read; a Push decrements the stack pointer *after* the write. Now that you're thoroughly confused, watch PROG10's JSR-RTS pairs put the stack through its paces.

Load and list the first few instructions. As with programs that contain JMPs, the disassembly shows the first five instructions in sequence, not those at the destination of a JSR. If you want to see that code, you'll have to ask for it.

This program "calls" (to use a popular synonym for gosub) a routine at $C200 to load the X and Y registers with $FF's, and a second routine at $C100 that stores X and Y in a pair of consecutive zero page addresses.

The things to watch: *JSR*'s put data on the stack (what data? The two halves of the program counter, PCH and PCL). *RTS*'s pull data off the stack and into the program counter. For this program, do a WINDOW MEM and use RC 1F8 to display locations $1F8–$1FF. That's where the action will be.

Note that PCH is pushed *first* during JSR, and so must be pulled *last* during RTS. The address that goes into memory is the address of the last byte of the JSR instruction. RTS takes care of a final increment of

the program counter to fully restore it to where we want to be, pointing to the instruction after the JSR. Also notice that pulling a byte from the stack does not erase it; it is not changed until something else is pushed there.

Nesting

The subroutine at $C100 demonstrates how one subroutine can call another. The RTS at $C182 makes us return to the point of the most recent subroutine call, the one at $C100. The 6502 and its 256 byte stack allow you to go 128 subroutines deep, and still find your way back to the calling program.

The Stack for its Own Sake

There are four other instructions that use the stack. They are implied, one byte commands to push and pull the accumulator and P register.

MNEMONIC	OPCODE	OPERATION
PHA	$48	Push accumulator on stack
PLA	$68	Pull accumulator from stack
PHP	$08	Push processor status register
PLP	$28	Pull processor status register

You will probably not have occasion to use PHP or PLP for awhile, even though this is the only way to load or store the processor status register. Usually P just sits there.

PHA and PLA, however, get lots of use as a means of temporarily storing a number without tying up a register or memory location. Suppose the accumulator contains the result of an important operation, but before we can use that result, we need the accumulator for another calculation. We have two options: save the intermediate value in an unused register or memory location, or, push it on the stack. In many cases the latter course is best. When we are ready for the intermediate value, we pull it back into the accumulator.

There are two things to watch out for when you use the stack for data storage: First, there are a limited number of bytes in the stack and you will overwrite data with the 257th push (wraparound effect). If you are sharing the stack with Basic and the kernel (such as when a machine language program is called from Basic), you have even fewer stack bytes available.

Second, if you are currently "within" a subroutine (i.e., a JSR has been executed without a corresponding RTS), you must be careful not to tamper with the stack so that the RTS will not work. This can happen two ways: Pushing a number and not pulling it before the RTS, or pulling a number without a preceding push. Both cause RTS to use two bytes that point somewhere, but *not* to the end of the JSR that called this routine.

PROG11 demonstrates care and feeding of the stack. The first subroutine (at $C100) is a painfully slow delay loop. How can we get out? (We're willing to accept on faith that eventually X will be reduced to zero, and RTS executed.) By getting out, I mean getting back to the main loop that called this subroutine. Pretend you don't remember that we started at $C000.

There are a couple of ways to do this. We could haul off and use the monitor to load X with 1 (doesn't take long to decrease a 1 to zero), and let the RTS occur normally. Or, we could peek into the stack page, figure out what bytes are the return address of the subroutine, and load the program counter (plus one, of course) with those numbers.

The easiest way is the monitor's POP command. Executing a POP places the top two bytes of the stack (plus one) in the program counter, and increments the stack pointer by two. POP is the monitor's equivalent of RTS, and is useful in situations where you weren't watching closely and got into a subroutine without knowing how you came to be there. POP the address of the calling program to find out.

The subroutine at $C200 demonstrates how *not* to use PHA and PLA. By the time we get to the RTS that should return us the main program, the data at the top of the stack is part return address, part left-over pushed accumulator contents. *Ouch.*

Jump, Indirect

Both JMP and JSR are three byte instructions using absolute addressing. JMP has a second addressing mode called indirect, opcode $6C. In mnemonic form:

```
JMP ($2000)
```

Like JMP, absolute, JMP, indirect is a three byte instruction that diverts program flow, without saving a return address; the mechanism for determining the address jumped to is different, however.

JMP ($2000) tells the 6502 to jump to the address stored in memory locations $2000 and $2001. Not to jump to $2000 and start executing code, but to look there for the values to place in the program counter. If $2000 contains $F0 and $2001, $FD, then the program counter will end this instruction containing $FDF0. This is conceptually one level deeper than a normal JMP and you are entitled to feel a bit queasy at this moment. If you think of JMP as a load instruction for the program counter (which it is; we just don't call it that), then JMP absolute is a load immediate. JMP indirect is a load absolute. Since the program counter is 16 bits wide, two loads must be made from sequential locations. With a little imagination, the operand's use of parentheses implies how the indirect jump works.

The 6502 is a great microprocessor, but the folks at MOS Technology made one little goof (hardware can have bugs, too) in the JMP indirect instruction. Under certain conditions, it doesn't work. Luckily, people discovered this problem years ago, so you don't have to; just remember that it's there.

The bug affects indirect jumps that cross page boundaries. JMP ($20FF) will fetch the bytes from $20FF and *$2000* to form the new program counter, instead of from $20FF and *$2100*. This quirk has been faithfully copied in TVC.

PROG12 contains an indirect JMP. The first time through the instruction: JMP ($C200), we end up at $C010. Later, the same instruction puts us somewhere else.

Now, a Message From Our Sponsor

Why should machine language programmers organize their programs in subroutines? For the same two reasons that a smart Basic programmer does. First, for efficiency, so that separate parts of a program may share a section of code without each having to duplicate it. Second, for clarity of structure.

If you are to become a successful machine language programmer, you will need to make things as easy on yourself as possible, by writing programs that are clear and easy to follow. The "rat's nest" technique of jumps to jumps to jumps will have you spending more time figuring out what you did yesterday than on today's work. A good structure for machine language and Basic programs is to use subroutines liberally, sometimes even if they are called only once.

The Ideal Basic Program

```
100 GOSUB 1000
110 GOSUB 2000
120 GOSUB 3000
130 GOSUB 4000
140 GOTO 110
```

The Ideal Machine Language Program

```
      JSR $1000
LOOP: JSR $2000
      JSR $3000
      JSR $4000
      JMP LOOP
```

To climb down from my soapbox, let me say that even in well structured programs, you will make enough mistakes to satisfy your inborn programmer's desire for debugging sessions.

11.
Instructions That Work: ADC/SBC

So far we haven't learned any instructions that really sink their teeth into a programming problem. We've loaded and stored and jumped over, under, around, and through, but haven't accomplished much in the process. A 6502 with only the instructions we've learned so far would be like a car with a great stereo, and plush seats, but no engine. This chapter introduces a pair of high octane computational instructions, ADC (add with carry) and SBC (subtract with borrow). These instructions may use any of the three all-purpose addressing modes we've used so far.

INSTRUCTION	ADDRESSING MODE ABS	IMM	ZP	OPERATION
ADC	$6D	$69	$65	Add with carry
SBC	$ED	$E9	$E5	Subtract with borrow

We've made reference to the accumulator's importance without saying why it's such a popular place; now we'll see. The accumulator is where numbers have to be to have SBC and ADC operations performed on them. You can't use any other register.

To add $23 to $14, load the accumulator with $23 and ADC #$14 to it. The answer, $37, replaces the $23 that was in the accumulator. Results *accumulate* there. The accumulator is always involved in half of a computation and holds the result.

The operation of the ADC instruction is as simple as adding two eight bit numbers, something that humans learn to tackle in the second grade. The only thing remotely tricky has to do with why it's called "ADC", add with carry, and not just "ADD". The word "carry" means exactly the same process that humans use when they add numbers on paper.

```
    1         11
   34         66
+  19      +  44
   --        ---
   53        110
```

The 6502 needs the carry flag to keep track of whether an addition has produced a result greater than can be held in the accumulator. The accumulator can't grow, so C is drafted to be its ninth bit. Is nine bits enough to represent the largest possible result of eight bit addition? Check it out.

```
   $FF
 + $FF
  $1FE   (1 1111 1110)
```

Apparently so. Anytime an ADC produces a value greater than 255, the carry flag is set.

```
   $7F                          $31
 + $82                     +    $16
   $01   + a carry              $47   no carry
```

ADC also conditions the Z and N flags, according to the same rules we have already learned for these flags. If an ADC causes a zero to be in the accumulator, the Z bit will be set. If it causes bit 7 of the accumulator to be set, then the N flag will be set. Otherwise, N and Z will be reset.

Not only does an ADC condition carry going out, it also includes carry in the addition; if carry happens to be set going into an ADC, the result will be *one greater* than otherwise. This is a slight annoyance when we need to quickly add a couple of eight bit numbers, as we must execute a CLC before ADC to insure that we get the right answer, but is a blessing for more complex calculations, as we shall see.

PROG13 demonstrates ADC in action, using immediate addressing. Bring in PROG13 and execute it. Play around with different values for the data bytes until you are comfortable with your understanding of how ADC computes a new value for A, based on the operands and the carry bit going in, and second, its conditioning of the Z, N, and C flags going out.

Multiprecision Arithmetic

Despite the potential confusion in having to consider the state of the

carry bit on every addition, the C flag is the key to performing calculations on numbers greater than 255.

Even though the accumulator is limited to eight bits, it is possible to add and subtract numbers much larger than 255 using multiprecision arithmetic. This means using two or more bytes in memory to represent values. How big a number can you store in two bytes?

$$2^{16} - 1 = 65{,}535$$

In three?

$$2^{24} - 1 = 16{,}777{,}215$$

We quickly come up to a range of useful magnitudes. PROG14 is a two byte addition, using the zero page forms of ADC, LDA, and STA. Before we run it, we'll need to EDIT some numbers into page zero for it to use. Do this addition:

```
   $13FC  (A)
 + $4597  (B)
   $????  (C)
```

You might want to first run this problem through the calculator to see if the program produces the same result (it better!). We're going to use zero page memory locations $F0 – $F5 to store operands A and B, and the answer, C. Use $F0 and $F1 for A, $F2 and $F3 for B. Initialize $F4 and $F5 with zeros. Use EDIT mode to write the data into memory. As always, LSB in the lowest location. $F0 should get $FC, $F1 should contain $13, and so on.

Run the program a few times with different data. What happens if your addition produces a value greater than we can store in 16 bits? Is the carry flag still enough to handle the result?

Subtraction

The 6502 also has an instruction for subtracting one byte numbers, SBC, Subtract with Borrow. It functions more or less like ADC with a confusing twist. Like ADC, it uses the accumulator for the first operand and a memory location for the second, with the accumulator getting the result. Subtracting 2 from $14:

```
LDA #$14
SBC #$02
```

The confusing part concerns the borrow flag; namely, *there is no borrow flag*. (B is the break flag, and has nothing whatever to do with subtraction.) Borrow is defined to be the opposite of carry. If C is set, borrow is reset; if C is reset, borrow is set. Confusing? You bet it is.

Take the subtraction:

```
  7
- 2
---
```

To perform this problem on the 6502, place 7 in the accumulator, and execute SBC #$02. As with ADC, the answer includes the carry flag in some way. If C is set when this instruction is executed, you'll get 5 in the accumulator for an answer, because *a set carry bit means no borrow*. If C was clear, then the answer will be 4, because a clear C bit means a borrow occurred previously.

Like ADC, SBC conditions the carry flag going out, too. Whenever a bigger number is subtracted from a smaller one, a borrow is generated (*carry is cleared*).

```
   $14             $14            $14
 - $22           - $12          - $14
 - $0E             $02            $00

  Borrow         No Borrow      No Borrow
(Carry clear)   (Carry set)    (Carry set)
```

PROG15 contains some exercises that demonstrate SBC and its backwards use of the carry bit. Load and list it.

```
PROG15

SEC                (Clear borrow, by setting carry)
LDA #$07
SBC #$02
CLC                (Set borrow, by clearing carry)
LDA #$07
SBC #$02
LDA #$14
SBC #$22
```

Experiment with different values until you understand how carry affects subtractions going in, and how subtractions condition carry going out.

Multiprecision Subtraction

Situations arise that require multiprecision subtraction. PROG16 demonstrates a 2 byte subtraction. LOAD and list it. PROG16 will subtract the two byte number stored at $F2, $F3 from the two byte number stored at $F0,$F1, and put the answer in $F4,$F5. Use EDIT to set up this problem:

```
  $73A1
- $46B1
```

Now execute it. The carry bit winds up set at the end of this program, meaning no borrow resulted from the overall subtraction of these two numbers. And this is what you'd expect, since $46B1 is smaller than $73A1. Tinker with the values until you are able to predict each time the behavior of the imaginary borrow flag going into and coming out of SBC instructions.

Multiplication and Division

Regrettably, the 6502 has no built-in multiply and divide instructions. Some of the newer microprocessors (8086, 68000, Z-8000) do. But with a little programming we can use multiple applications of addition and subtraction to get the same result.

To multiply n times m, add m to itself n times. To divide n by m, count how many times m can be subtracted from n. This sounds involved, and for a human it's not recommended, but a speedy rascal like the 6502 can do this a hundred times in the beat of a hummingbird's wing.

For example, 12 X 4 is equivalent to:

$$12 + 12 + 12 + 12$$

or,

$$4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4$$

PROG17 is an eight bit multiply. The values stored in $F0 and $F1 are multiplied together, with the result going to $F2 and $F3. Verify for yourself that two bytes are sufficient storage to cover the greatest possible 8 bit multiply. Before you execute it, give it some numbers to use. To save time, keep $F1 fairly small, say less than $10.

PROG18 is an eight bit division. The number in $F0 is divided by the number in $F1. $F2 gets the quotient and $F3 the remainder. These two programs only scratch the surface of the subject of machine language multiplication and division algorithms, i.e., there are *better* ways to do it.

12.
Beyond Adding and Subtracting

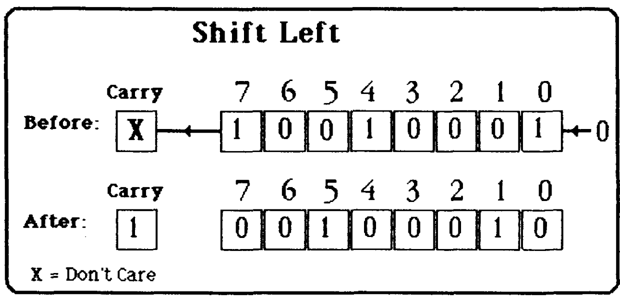
Thus far we've encountered two groups of 6502 instructions that actually get their hands dirty and perform calculations: The ADC/SBC pair, and the increment/decrement series. This chapter introduces two more groups of instructions to tackle problems with: The *logical* and *shift* instructions.

These commands differ from the ones seen previously in that they use the contents of registers (usually the accumulator) on a bit basis rather than on a cumulative basis. When we added $14 to $78 in the last chapter, we were happy to consider the $8C that turned up in the accumulator as just that: the quantity $8C. For the logical and shift instructions, however, we are usually more interested in a number's individual bits than their collective value.

Shift and Rotate

The 6502 has commands for sliding the bits in the accumulator one position to the left or right. As did ADC and SBC, these instructions treat carry as the ninth bit of the accumulator.

An ASL (Arithmetic Shift Left) shifts all the bits in a memory location or the accumulator one position to the left. All the bits slide over one position to the left, bit 7 goes into C (whatever was in C is lost), and a zero replaces whatever moved out of bit 0.

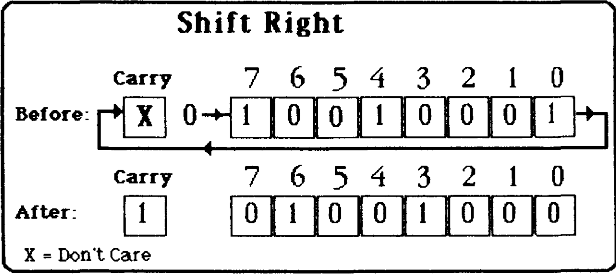


Cute, but what good is it? First, it gives us a way to test any bit in the accumulator and branch accordingly. Suppose we've done an operation and we need to sample the contents of bit 5 and branch depending on what we find there. There is no *BA5*, "Branch on Accumulator Bit 5 Set", so we proceed as follows: Three consecutive ASL instructions to slide bit 5 into carry, then BCS to test and branch.

A shift left has the surprising effect of multiplying by two. Try it.

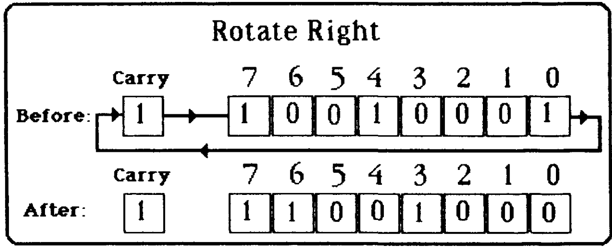
```
$20 (0010 0000)  X  2  = $40  (0100 0000)
$37 (0011 0111)  X  2  = $6E  (0110 1110)
$64 (0110 0100)  X  2  = $C8  (1100 1000)
```

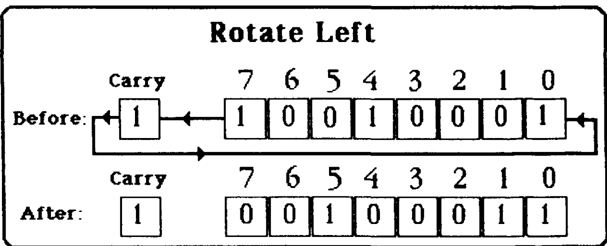
You multiply by four with two ASL's, by eight with three, and so on.



LSR (Logical Shift Right) is like ASL only we move right instead of left. Bit zero goes to the carry bit and a zero is shifted into bit 7. This *divides* the accumulator by two. Again, don't take my word for this. Experiment with the TVC calculator. The value left in the accumulator is the quotient; the values shifted out of bit 0 are the remainder.

The rotate instructions are only slightly different. A rotate doesn't shift in a zero, it *rolls* in the contents of the carry flag.





Rotations do not produce multiplication and division by multiples of two, unless you clear the carry bit ahead of time.

(Historical aside: ROR, the Hawaii of 6502 instructions, was the last instruction to be added to the 6502 instruction set. In fact, the earliest 6502's did not have ROR at all.)

None of the other registers may be shifted or rolled; however, you may shift and roll memory locations. Both absolute and zero page modes are available for shifts of memory.

INSTRUCTION	ADDRESSING MODE			OPERATION
	ABS	ACC	ZP	
ASL	$0E	$0A	$06	Arithmetic shift left
LSR	$4E	$4A	$46	Logical shift right
ROL	$2E	$2A	$26	Rotate left
ROR	$6E	$6A	$66	Rotate right

Shifts and Rolls of the accumulator are one byte, implied instructions, which for some reason are not grouped with the other implied instructions, but rather are the only members of so called "accumulator" addressing.

PROG20 is a multiprecision shift. The two byte value at $C100 and $C101 (low order byte first, of course) is multiplied by four by the application of two ASL/ROL pairs. Shifting the low order byte puts its old bit 7 in carry; we get that value into bit zero of the high order byte with a roll of the high order byte.

The Logical Instructions

The standard assortment of logical operators are available to the 6502 programmer.

AND Logical And

ORA Logical Or (Inclusive Or)

EOR Logical Exclusive Or

Like SBC and ADC, these instructions operate on the accumulator. In addition, they condition the Z and N flags according to the same rules.

Like the shifts, the logical instructions are cases where the trees are more important than the forest. What occurs in the instruction AND # $33 is eight simultaneous logical ANDs of each bit of the accumulator and the corresponding bit of the selected memory location. For example:

```
      0011 0011  ($33)           1100 0000  ($C0)
AND   0100 0110  ($46)     AND   0100 1111  ($4F)
  =   0000 0010  ($02)       =   0100 0000  ($40)
```

One use for AND is to force selected bits of the accumulator to zero. To force bits 6 and 7 of the accumulator to zero, without affecting the other bits, AND the accumulator with $3F. To force every bit position but #2 to zero, and leave #2 in whatever state it was in before, AND the accumulator with $04. Verify on paper that this works.

ORA is useful for setting selected bits. To set bits 4 through 7 of the accumulator, without affecting the values in positions 0 through 3, use ORA #$F0. To set only bits 3 and 7, use ORA $88.

EOR can be used to complement a number (reverse the polarity of each bit). EOR #$FF will flip every bit in the accumulator; ones become zeros and zeros ones. Two applications of EOR #$FF leave the accumulator unchanged.

```
    0011 0111  $37
EOR 1111 1111  $FF
    ---- ----
    1100 1000  $C8

    1100 1000  $C8
EOR 1111 1111  $FF
    ---- ----
    0011 0111  $37
```

Compare

A powerful tool in test-and-loop situations is CMP, Compare Memory with Accumulator. There's also a CPX and a CPY for the index registers. A compare subtracts the selected memory location from the accumulator (or X or Y) and sets the N, Z, and C flags accordingly, but does not affect the value in the accumulator. So what good is a subtraction that doesn't affect the accumulator? Plenty.

INSTRUCTION	ADDRESSING MODE			OPERATION
	ABS	IMM	ZP	
CMP	$CD	$C9	$C5	Compare memory with accumulator
CPX	$EC	$E0	$E4	Compare memory with X register
CPY	$CC	$C0	$C4	Compare memory with Y register

Following is a program to demonstrate the compare instruction, but first a digression on the subject of joysticks.

Joystick Fundamentals

Stripped of bright plastic and fancy pistol grips, a joystick reduces to five switches, each of which can be either open or closed. There's one switch for the fire button and one for each direction. If you push the stick diagonally, two direction switches close at the same time.

These switches are mapped into the first five positions of I/O address $DC00 for for port A and $DC01 for port B.

Position	Switch
4	Fire
3	Right
2	Left
1	Down
0	Up

Set bits in $DC00 or $DC01 reflect open (not pressed) switches. That is, if the fire button is pressed, bit 4 will be a 0. FCHECKPROG is a subroutine that determines whether or not the fire button of joystick B is pressed. If the fire button is pressed, the subroutine returns with $FF in the accumulator. If the fire button is not pressed, the accumulator returns as zero.

```
START: JSR   FCHECK
       BRK

FCHECK:LDA   $DC01
       AND   #$1F       mask out top three bits
       CMP   #$10       is fire pressed?
       BCC   NOFIRE      If bit 4 is set, A is $10 or above.  If
                        Bit 4 is reset, A is less than $10, a
                        borrow occurs (Carry is cleared)
       LDA   #$FF
       RTS
NOFIRE LDA   #$00
       RTS
```

Load and execute FCHECKPROG. As you run it, be aware that TVC checks the joystick port at the moment of the read microstep of that location, so do your pressing then. If you don't have a joystick plugged into port B, $DC01 will behave as though there's a joystick plugged in with ~~all the switches continuously pressed.~~
NO SWITCHES DEPRESSED

Special Case: The Bit Instruction

The last 6502 logical instruction is BIT, a hybrid of AND and CMP. BIT (Bit Test) performs an AND operation between the accumulator and memory location—but, like CMP, conditions flags without altering the accumulator. As a bonus, BIT also transfers bits 6 and 7 of the memory location under test to the V and N flags, respectively. It is useful in checking I/O addresses that contain status information, particularly if bit 6 or 7 is the one that we're watching.

The logical instruction are supported by the three addressing modes we have encountered so far.

INSTRUCTION	ADDRESSING MODE			OPERATION
	ABS	IMM	ZP	
AND	$2D	$29	$25	And memory with accumulator
EOR	$4D	$49	$45	Eor memory with accumulator
ORA	$0D	$09	$06	Or memory with accumulator
BIT	$2C		$24	Test memory with accumulator

PROG21 demonstrates AND and ORA setting and clearing bits in the accumulator. The subroutine at $C200 uses AND as a logical operator. If memory locations $C100 and $C101 both contain $FF, return with the accumulator equal to $FF. Otherwise, return with $00 in the accumulator.

13.
Indexing: Special Uses for X and Y

We mentioned in passing a while back that X and Y could be used as index registers. The time has come to find out what an index register is, and learn some new addressing modes in the process. So far we've encountered five addressing modes. Two of the five, Relative and Implied, are special cases. Relative addressing is for branches only. Implied instructions (TAX, CLC) have no other form.

The other three addressing modes, Immediate, Zero Page, and Absolute, are more general, allowing the same instruction to be used in different situations. We have a choice in how we may load the accumulator; with a number contained in the instruction itself (immediate addressing), or with the contents of an address specified in the instruction (absolute and zero page addressing).

To this list of general purpose addressing modes we now add four indexed addressing modes: Absolute, X; Absolute, Y; Zero Page, X; and Zero Page, Y. Operands to indicate these new addressing modes are as follows:

```
LDA $4000,X
LDA $4000,Y
LDA $00,X
LDX $00,Y
```

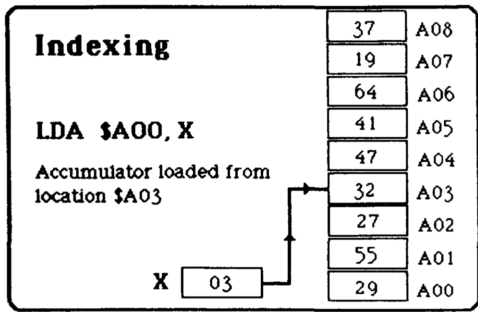
Indexing can be explained by presenting a problem that can't be easily handled by the addressing techniques we already know. Suppose we need to move a a cluster of $10 bytes residing in addresses $C100 through $C10F, to make room for something else. With the addressing modes we've learned up to now, we can accomplish this "block move" with the following program:

```
LDA $C100
STA $C200
LDA $C101
STA $C201
LDA $C102
STA $C202
LDA $C103
STA $C203
etc...
```

To move all 16 bytes we'd need 32 instructions at three bytes apiece. Not very efficient to use 96 bytes of program to make room for 16 bytes of data. And what if we had to move 100 bytes? Or 8000? Wouldn't it be nice if there was a way to handle this situation with some incrementing and looping? Enter indexed addressing, in which the X and Y registers are used as *offsets* from a *base* address.

Bases? Offsets? Let me show you what I mean.

So far we only know one way to load the accumulator from $0903, LDA absolute. But what if we use a new addressing mode for LDA that provides a two byte *base address* of $0900, and tells the 6502 to modify that base address with the current contents of the X register. If we execute the instruction LDA $0900,X (hex form $BD $00 $09) at a moment when the X register contains three, the accumulator is loaded from $0903. If we then increment X and execute the same instruction, the accumulator will load from $0904.



Indexed addressing makes block moves a breeze. PROG22 demonstrates a more elegant solution to the move problem.

```
$C000 LDX #$00
$C002 LDA $C100,X
      STA $C200,X
      INX
      CPX #$10
      BNE $C002
      BRK
```

From 32 instructions, 96 bytes, to 6 instructions, 13 bytes. Quite a savings. And we can move as many as 256 bytes without the program growing one whit. As you step through this program, the thing to watch is the new microstep "CALC ADDRS" (calculate address), in which the address bus is modified by the X register. Otherwise, in conditioning of flags, and ultimate result, LDA absolute, X, is exactly like LDA absolute.

Arrays

Another important application of indexing is in arrays. Absolute, X addressing allows us to easily form up-to-256 element arrays, with program controlled access of each element. For example, suppose we wrote a payroll system entirely in machine language. Forget that we said in Chapter 1 how dumb that would be.

For accounting purposes, Binary Inc.'s payroll system must know the age of every employee in the company. A natural way to store these ages is in an array, with one entry for each employee.

```
Array Position      Contents
(Employee No.)       (Age)
      0               $40         Mr. Boole, President
      1               $16         Mr. Watson, Vice President
      2               $20         Mr. Smith, Accounting

  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    255               $10         Brian, mail clerk
```

Assume the table is stored at \$C100–\$C1FF. When the program needs to know how old a given employee is, it uses his employee number to index the correct element in the array. As long as employee numbers are less than 256, and there aren't any employees older than 255, this can be done as simply as:

```
LDX  (employee number)
LDA  $C100,X
```

Suppose Binary is required by Federal Corporate Witholding Guideline T-19342.12 to submit quarterly reports listing the total age of every employee of the company. Not the average age (that's form A-19342.12), but the sum of each employee's age. COUNTPROG does just that. Load it now and examine this listing.

```
COUNTPROG

START: LDX   #$00
       STX   $FB
       STX   $FC
TOP:   LDA   $C100,X
       CLC
       ADC   $FB
       STA   $FB
       BCC   SKIP
       INC   $FC
SKIP:  INX
       BNE   TOP
       BRK
```

COUNTPROG steps through the $100 element age table at $C100, from the bottom up, accumulating a total in zero page locations $FB,$FC, where $FB is the least significant byte. The X register serves as both loop counter and index register.

First, X and the zero page addresses that make up the age accumulating locations (call them "Total") are initialized to zero. We then enter a loop that loads the accumulator with the age of employee #X, and adds that eight bit value to the 16 bit value in $FB–$FC.

The add step rates more detail. First we clear carry—if we didn't, we'd add an extra year to the total on some adds. Next, add the accumulator (now holding employee X's age) to Total's LSB, and immediately write the result back to the LSB. If this operation produces a carry, for instance, as it would in the case of Total = $3F2 and Age = $21, the MSB is incremented by one. In this example, after the ADD step, the accumulator holds 13, and carry is set. The BCC test fails and we fall through to INC $FC, where Total's MSB becomes $04. Verify with the calculator that ($3F2 + $21) is $413. If the addition hadn't produced a carry, we would have skipped the increment of Total's MSB.

256 repetitions of the loop will occur before the BNE test fails, causing the program to encounter the ending BRK. Now run COUNTPROG. It's going to take a while, so go fix yourself a sandwich. When it finishes, divide Total by 256 with TVC's calculator to get the average age of a Binary Inc. employee.

The Y register can be used interchangeably with X in most indexing operations. This table summarizes opcode values for the load and store instructions for these new addressing modes.

INSTRUCTION	ADDRESSING MODE	
	ABS,X	ABS,Y
LDA	$BD	$B9
STA	$9D	$99
LDX		$BE
STX		
LDY	$BC	
STY		

Notice that for the first time an opcode table has gaping holes. There isn't an opcode for STX ABS,Y. Nor is there one for LDY ABS,Y. Not all addressing modes are available for all instructions. This is partially due to a logical conflict: Does it make sense to load the very register you've used to locate the memory location you're loading it with? But it stems mainly from the physical limitations of integrated circuit technology, circa 1975. Much as we'd like to have them, there wasn't room on the chip to provide every addressing mode for every instruction.

The most important instructions were given the most addressing modes: ADC, SBC, EOR, AND, ORA, CMP, LDA, and STA. Consult appendix F for the addressing modes available for each instruction.

Indexing on Page Zero

That's two new addressing modes, Absolute, Y and Absolute, X. Indispensable, but like all three byte instructions, real memory hogs. There are also two byte, space saving, Zero Page,X and Zero Page,Y addressing modes.

INSTRUCTION	ADDRESSING MODE	
	ZP, X	ZP, Y
LDA	$B5	$B9
STA	$95	$99
LDX		$B6
STX		$96
LDY	$B4	
STY	$94	

PROG23 demonstrates zero page indexed addressing. Watch for wraparound. If adding X to AD in the CALC ADDRSS microstep produces a value greater than $00FF, ADL wraps around so that it always contains a zero page address. LDA $80,X, if executed at a moment when X contains $90, will load the accumulator from $10.

14. The Kernal: Canned Subroutines

The C-64 has eight thousand bytes worth of built-in subroutines at the top of the memory map called the Kernal. These programs perform utility chores such as moving data to and from disk, and reading the keyboard. The Kernal has more than 40 functions you can tap to control the Commodore 64. If your program needs to do any of these tasks, this is the method of choice.

One such Kernal subroutine is named GETIN. Here's what the *Programmer's Reference Guide* has to say about this function:

Function name: GETIN
Purpose: Get a character.
Call Address: $FFE4
Communication registers: A
Registers Affected: A, X, Y

Description:

...This subroutine removes one character from the keyboard queue and returns it as an ASCII value in the accumulator. If the queue is empty, the value returned in the accumulator will be zero. Characters are put into the queue automatically by an interrupt driven keyboard scan routine which calls the SCNKEY routine. The keyboard buffer can hold up to ten characters. After the buffer is filled, additional characters are ignored until at least one character has been removed from the queue.

How to use:

1. Call this routine using a JSR instruction.
2. Check for a zero in the accumulator (empty buffer)
3. Process keystroke.

Interrupt driven? SCNKEY routine? Queue? What is all this stuff?

At this point, we don't care, and we don't need to. The beauty of using a Kernal routine is that you don't *have* to know how it works. You only have to know how to use it.

GETIN is a simple tool. When your programs needs to know what the human is doing at the keyboard, it calls $FFE4, and when that subroutine returns, the accumulator holds the ASCII value of the key pressed. Even though GETIN is a complicated guy, with internal subroutine calls of its own, and strange interactions with various and sundry elements of keyboard hardware—we don't have to know about this stuff to use it.

ASCII

Commodore uses a modified form of the ASCII (*as*-key, American Standard Code for Information Interchange) character set used by most computers and peripherals, so it is possible, if not always easy, to hook a Commodore 64 up to other manufacturer's equipment (like an Epson printer), and have them agree on the number that represents a comma and so on. Appendix C of the *Programmer's Reference Guide* lists the ASCII value returned by GETIN for each key.

There are three good reasons for letting Commodore's routines do the work for you. First, your program requires less memory because you don't have to include code to perform these functions. Second, your program takes less time to write and debug because you don't have to consider the peculiarities of disk or keyboard hardware to get the job done. GETIN is already written and debugged.

Third, should Commodore, in its wisdom, someday invent a new computer, with a new keyboard, requiring a different method of reading keystrokes, your programs will still run—because you used GETIN to get characters from the keyboard, rather than reading the keyboard directly. GETIN changed, one of Commodore's programmers had to deal with the new keyboard—but your program doesn't. It only cares that a JSR $FFE4 returns a keystroke in the accumulator.

Commodore has committed to supporting the Kernal routines in future machines. As long as GETIN still has the same "calling conventions", (same JSR address, returning a value in the same way), your 64 programs will run unmodified on the new machine. This technique of hiding the details of a problem, of making a program's components as in-

dependent of each other as possible, is central to the concept of structured programming.

Another Kernal function tells time. One of the C-64's talents is keeping track of how many sixtieths of a second (or *jiffies*, where 60 jiffies = 1 second) have gone by since it was turned on. Although it is simple enough to look at the memory locations where this 24 bit counter is stored ($A0–$A2, according to the *Programmer's Reference Guide*), doing so puts your program at the mercy of Commodore's whims. The *C#128* might store the time at $B0–$B2. Better to call the Kernal function RDTIM.

Function Name: RDTIM
Purpose: Read system clock.
Call Address: $FFDE
Communication Registers: A, X, Y
Registers Affected: A, X, Y
Description:

This routine is used to read the system clock. The clock's resolution is one 60th of a second. Three bytes are returned by the routine. The Y register contains the most significant byte, the X register contains the next most significant byte, and the accumulator contains the least significant byte. (Note: This order is stated backwards, A-X-Y, in the *Programmer's Reference Guide* discussion of RDTIM.)

Let's step through RDTIM with the simulator. No need to load a PROG; we're going to write this one on the spot. Using the EDIT function, place these values at $C000 through $C003:

20 DE FF 00

We just wrote the instruction JSR $FFDE, a call to RDTIM, followed by an ending break. Set the program counter to $C000, if it isn't already, and step through it. Don't worry at this point why RDTIM is so concerned with the interrupt disable flag, and why it insists on writing back the values it read a moment ago. RDTIM is complete when you hit the BRK at $C003.

It probably returned a counter value of 00 00 00 jiffies. Hmmmmm. This implies that approximately 0.00 seconds have elapsed since we turned on the computer. Since TVC requires more than a minute to

load, this calls for some investigation.

If you browse through page zero with EDIT, you'll notice that this area is *all* zeros. This seems contrary to earlier statements about the crowded conditions in low memory, resulting from Basic's and the Kernal's storage of important data there. Surely this important data isn't all zeros.

We confess. TVC has a "fake" zero page (*gasp!*). All this time you've had a wonderfully uncluttered page zero to work with. 256 pristine bytes of zeros, free for the asking. This is possible because TVC, clever simulator that it is, fakes us out when we ask it to read from or write to an address in the range $0000 - $00FF; it stores this range elsewhere.

The C-64's jiffy counting mechanism uses addresses in the real, bona fide zero page, which is not available to the beginning TVC user. Access to the real zero page requires a TVC master.

Master Mode

Thus far we've been in non-master mode exclusively. This is a good place for beginners to be; non-master mode makes it just about impossible for you to hang up the computer, short of prying the 6502 out of its socket with a fingernail file. Writes to locations inside the TVC program and other dangerous areas are not allowed, and I/O references that can do messy things are locked out.

To enter master mode, simply ask: MASTER ON.

The M (for "master") and Z (for "zero page share") flags on the status line illuminate. You are now a Visible Computer Master. (Feels great, I know.) You're also a giant step closer to the real world where a single wrong move can confuse your computer so thoroughly that you'll have to turn it off and on again to set things straight.

Many, indeed, most of the locations in page zero are critical to the health of Basic and the Kernal. Do an LC 88. That $60 at location $8A, for example, is critical to your machine's ability to input characters from the keyboard. Change it and see what happens.

We'll wait for a couple of minutes while you get TVC running again. Tap, tap, tap... Fidget. Twiddle thumbs... While we're waiting, let me reassure you that running a program, no matter how bugged, can't physically damage your computer. The only thing a bugged program can hurt is your ego.

Okay, back? Be sure to make yourself a master again.

Location $8A is not unique; there are many addresses that have to have the correct value all the time. That great engine that is C-64 Basic needs its zero page locations intact so that it can go on correctly executing the program named "The Visible Computer". The Kernal needs zero page locations to deal with matters such as writing text to the display. If just one of them is changed, the whole house of cards that is the Commodore 64 operating system can come crashing down.

Anyway, the point is, as a machine language programmer, you have a lot of control, and the price of that control is increased responsibility. You've become an equal partner with Basic and the Kernal, and need to be a team player. Again, four safe zero page locations (unused by both the Kernal and Basic) are $FB–$FE.

Rewrite the JSR $FFDE that was at $C000 before we pulled the plug, and run through RDTIM again. This time it should return a nonzero value. To calculate how many jiffies have accumulated since you last applied power, convert this six digit hex number into decimal.

For example, suppose RDTIM returned Y=$03, X=$31, and A=$F0. $0331F0 = (3*65536)+(3*4096)+(1*256)+(15*16) = 209,392 jiffies. That's (209,392 / 60), or 3,490 seconds, a little less than an hour.

Now display the timer locations with LC A0. Note that they aren't constantly changing. TVC doesn't consider the possibility of something other than itself writing to memory. Force it to recheck these locations with another LC A0 and you'll see updated values. Soon we'll learn about the mysterious mechanism that does jiffy counting.

15. Indexing, Part II

Indexing is a powerful technique that allows a program to access different addresses with the same instruction. This section introduces two more indexed addressing modes: Indirect, Indexed and Indexed, Indirect.

Let's review the concept of indirect addressing. Back in Chapter 10 we used indirect jumps (remember feeling queasy? That was JMP indirect). An indirect addressing mode doesn't specify the address to perform an instruction with—it specifies the address that *stores* the address with which to perform the instruction.

A garden variety JMP $B136 puts $B136 in the program counter and that's that. JMP ($B136) instructs the 6502 to fetch the contents of locations $B136 and $B137 and use those contents to form the new program counter. This enables us to change where the jump points under program control.

The 6502 has two addressing modes that use the indirect concept in conjunction with the index registers. Two forms are available: one that uses the X register only, called indexed, indirect, and one that uses the Y register exclusively, called indirect, indexed. (Yes, the names are confusing.)

Indirect, Indexed

Suppose you faced a situation that required a block move of greater than 256 bytes. You could tackle this problem with two consecutive applications of normal absolute, indexed addressing as shown below (Moves $200 bytes from $3000 to $4000).

```
        LDX #$00
LOOP1:  LDA $3000,X
        STA $4000,X
        INX
        BNE LOOP1
LOOP2:  LDA $3100,X
        STA $4100,X
        INX
        BNE LOOP2
        BRK
```

While this program would work, it is lacking in elegance. Two loops instead of one. If we needed to move four pages of memory we'd need four loops. Enter Indirect, Indexed addressing. In mnemonic form:

LDA ($45),Y

The operand's arrangement of the parentheses is a clue to how indirect, indexed addressing works. Since the Y is outside the parentheses, it's trying to tell us that the indirect portion of the instruction will be carried out first, and the indexing applied second.

For example, executing LDA ($45),Y: Memory location $0045 is read and the value stored in the data buffer. Next, location $0046 is read. Suppose we read a $00 from $0045, and a $20 from $46. We have now "indirectly" formed the address $2000, (as always, LSB first). Finally, apply indexing. If Y was equal to 6 when we executed this instruction, we will load the accumulator from location $2006. If we were to increment location $46 (making it $21), executing LDA ($45),Y again would fetch the byte stored at $2106.

Even though it takes several fetches of memory to execute (IND),Y instructions, and consequently more time than other addressing modes, they are extremely efficient for code length. (IND),Y instructions require only two bytes; one to specify the instruction and addressing mode; the second, the first of the consecutive zero page addresses that will form the base address.

Despite their two byte length, they can specify a location anywhere in memory. Indirect, indexed addressing is a big reason for the space crunch in page zero—every program needs a couple of zero page pointers. (Pairs of zero page locations used in this way are frequently called pointers because their contents "point" in memory to where an operation should occur.)

Only the Y register can be used this way. There is no LDA ($45),X instruction. There's a demonstration program named CLEARPROG for (IND), Y addressing. Load and list it.

```
C000:   LDY  #$00
        LDA  #$C4       ; make pointer pair $FB, $FC point to
        STY  $FB        ; first screen memory address ($C400)
        STA  $FC
        LDX  #$04       ; 4 pages of memory to write to
        LDA  #$20       ; $20 is control code for blank space
LOOP:   STA  ($FB),Y    ; write space code to screen address
        INY             ; finished a page yet?
        BNE  LOOP       ; branch if not...
        INC  $FC        ; now pointer pair points to next page
        DEX             ;
        BNE  LOOP       ; done after four pages
        RTS

C080    JSR  $C000
        BRK
```

CLEARPROG writes a $20, the control code for space, to all $400 screen memory addresses, effectively erasing the display. Actually only the first 1,000 addresses are used; writing to the unused 24 locations at the end is unnecessary but harmless.

We're using the popular free zero page pair $FB–$FC to point successively to every location in screen memory. First we make $FB–$FC point to $C400, and initialize X with 4 and A with $20. X is decremented after each page of writes. After four pages, the program terminates.

Don't execute more than a few cycles of this program. If we let it run all the way through to the BRK at $C083, eventually the screen would be cleared, but you would be bored into a coma. It's time to learn an important new command.

The GO Command

GO causes a program in memory to be executed not by the simulator, but *by the 6502 itself*. If you are in master mode, and if the next instruction is a JSR, then TVC passes execution of that subroutine directly to the 6502. When and if the 6502 encounters an RTS at the end of the subroutine, TVC regains control and redisplays the X, Y, P, A, and PC registers with the values they acquired in the subroutine.

There are a million and one ways (conservative estimate) that a machine language program can go wrong, and almost all of them will cause you to lose control of (''lock up''; ''hang''; ''kill''; ''crash'';

"blow away"; "bomb") the computer. You may spray a deadly hail of bytes into the TVC program, or your sick program may send the 6502 on a wild goose chase (e.g.: C000: 4C 00 08 JMP C000), in which case pressing the RUN/STOP and RESTORE keys simultaneously may regain control. (The RESTORE key is connected to the 6502 in a more intimate way that the rest of the keys, having an effect on it more like the power switch than a mere keypress.)

In severe cases of crashed computer syndrome, your only recourse is to turn the machine off and back on and start from scratch. Suffice it to say, bugged machine language programs are not especially forgiving. Luckily, CLEARPROG is pretested and guaranteed to work.

Get the program counter pointing to the JSR instruction provided at $C080. GO won't work if you're not on a JSR. Now brace yourself and GO. Doesn't take long, does it? RESTORE the display. Change the $20 at $C00B to a different value and you change what the screen becomes. $07 produces all "g"'s. $BF, inverse question marks. Have fun while you can, because we're about to spin your head completely around.

Indexed, Indirect

If you *liked* indirect indexed, you'll *love* indexed, indirect. Whereas indirect indexed addressing is only available with the Y register, indexed indirect is only available with the X register. Confusing? You know it. The mnemonic form is:

LDA ($45,X)

Again, an examination of the operand and some educated guessing furnish clues to how this addressing form works. Indexed indirect uses X to index a particular pointer pair out of several, which then points to an address in memory. For example, suppose addresses $10 - $17 held these values:

```
$10= $00        $14= $00
$11= $C0        $15= $C2
$12= $10        $16= $00
$13= $C1        $17= $C3
```

Eight locations make up four pointer pairs. The first points to $C000, the second to $C110, the third to $C200, and the fourth to $C300. In-

dexed indirect addressing uses the X register to select one pointer pair of many. LDA ($10,X) will load the accumulator from $C000 if X is 0, from $C110 if X is 2, from $C200 if X is 4, and from $C300 if X is 6. Pointer pairs don't have to be aligned on even addresses. If X is 3, LDA ($10,X) loads from $00C1.

Indexed, indirect addressing is usually used to select under program control which of several tables will be used in an operation. In practice it doesn't get as much use as (IND),Y, but the day will come when you'll be glad it's there.

More Binary, Inc. Payroll Problems

Through an interoffice memo, you've just learned of Binary, Inc.'s new retirement plan. All employees under the age of 48 are now required to double their monthly contribution to the retirement fund. Coincidentally, the youngest member of the board of directors is 48. Binary's overworked programming team is given the task of modifying the deduction calculation portion of the payroll system.

Here's the job: Using the employee age table as input, build a "retirement plan" table, with the following characteristics: If an employee is to be on the new retirement plan (he's 47 or younger), put his age in the table. If an employee stays on the old plan, his spot in the new table gets a zero. While you're at it, total how many people will be staying on the old plan. The age table, as always, is at $C100. The retirement table should be built at $C200.

```
BTABLEPROG

START:   LDA  #$C1      ; set up two zero page pointer pairs
         LDY  #$C2      ; $FB,$FC points to $C100 (age table)
         LDX  #$00      ; $FD,$FE points to $C200 (new table)
         STX  $FB
         STX  $FD
         STA  $FC
         STY  $FE       ; Y counts occurrences of over 47 folks
         LDY  #$00      ; initialize it to zero
LOOP:    LDA  ($FB,X)   ; read from age table (X = 0)
         CMP  #$30      ; $30 = 48 decimal
         BCC  YOUNG     ; clear carry means set borrow (age<48)
         LDA  #$00
         INY            ; increment counter for people over 47
YOUNG:   LDX  #$02
         STA  ($FB,X)   ; write to new table (X = 2)
         LDX  #$00      ; clear X
         INC  $FB       ; and increment pointers to prepare
         INC  $FD       ; for next pass through loop
         BNE  LOOP
         BRK
```

The active ingredient of BTABLEPROG (build table program) is the CMP #$30 step. The value in the accumulator fetched from the age table has 48 subtracted from it (on a trial basis only—the accumulator isn't affected, just the flags). If that value was 47 or less, a borrow occurs, and carry is cleared to reflect that fact. The BCC skips over a nulling of the accumulator, to the STA ($FB,X) that writes to the equivalent spot of the retirement table.

Load BTABLEPROG. If you have doubts about any part of it, especially the comparison step, run a few cycles with the simulator. Then zap it with GO. This program executes more or less instantaneously under 6502 execution. Before you can GO it, you'll need to write a JSR $C000 instruction somewhere, say $C080. To write this instruction and a following BRK, EDIT the values: 20, 00, C0, and 00 at $C080. Now GO it. How many people are 48 or older at Binary, Inc.? (Answer: 12—11 board members and the chairman's brother-in-law.)

Only the heavyweights of the 6502 instruction set have these powerful index/indirect addressing modes available to them: ADC, AND, EOR, SBC, ORA, CMP, STA, and of course, LDA. For opcode values consult appendix F.

16.
Some Fine Points

You may have noticed that this manual is filled with phrases like "this is a powerful group of instructions", or "this instruction gets a lot of use". This chapter concerns a couple that aren't so powerful, or don't get a lot of use, or both.

NOP (No OPeration), opcode $EA, implied addressing, doesn't do a thing. *Nada*. When the 6502 executes a NOP, the only effect is that the program counter will end up one bigger, and a little time will have been wasted. What good is an instruction that does nothing? It has two uses: as a short delay in a carefully timed counting loop, and most importantly, as a means of plugging blank spaces in memory, usually as a debugging technique.

If you were debugging this program:

```
0800:20 00 10   JSR $1000
0803:20 00 20   JSR $2000
0806:20 00 30   JSR $3000
```

and determined that the second subroutine had a problem, you could quickly check out the third subroutine by writing over the middle JSR instruction with three NOP instructions.

```
0800:20 00 10   JSR $1000
0803:EA         NOP
0804:EA         NOP
0805:EA         NOP
0806:20 00 30   JSR $3000
```

When we execute this program now, after the subroutine at $1000 returns we fall through to the subroutine at $3000. Appropriately enough, there is no demonstration program for NOP.

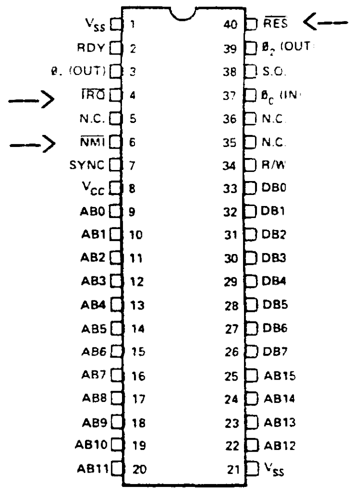
NOP's potential as an innocuous time waster brings up the subject of instruction execution times. Normally, we are only concerned that a program run fast, or at least fast enough. Sometimes we have to know exactly much time an instruction requires. The basic unit of time for the 6502 is the *instruction cycle*. In the C-64, one instruction cycle

takes one microsecond (.000001 second). All instructions need two or more instruction cycles. In general, the less reading and writing of memory an instruction requires, the faster it runs. DEX and SEC are fast, requiring only 2 cycles. LDA $45 takes 3 cycles. LDA ($01),X requires six cycles.

RTI

RTI (return from interrupt) is an instruction you may never use, although one is executed 60 times a second every moment a C–64 is turned on. But first, the sixty-four-dollar question: What's an interrupt?

Three of the 6502's 40 pins allow circuitry external to the 6502 to alter its normal fetch/execute/fetch/execute pattern. Like the bits in the status register, these pins are important enough to have their own names. They are: Reset, Non-Maskable Interrupt (NMI), and Interrupt Request (IRQ). All three cause the 6502 to stop what it's doing and go do something else. Some are more courteous to the program that's being interrupted than others, however.



Reset

There's only one way to generate a signal on the reset pin of the 6502 in a C–64: turn the machine on. A few milliseconds after you flip the

power switch, a circuit applies a pulse to the reset line. The 6502 drops what it was doing (probably nothing coherent, anyway) and performs an indirect jump to $FFFC (i.e., to the address stored in locations $FFFC and $FFFD). The reset program at the destination of the jump takes care of a complicated series of startup tasks that need doing before the machine can say, in light blue letters on a dark blue background:

```
            **** COMMODORE 64 BASIC V2 ****

     64K RAM SYSTEM  38911 BASIC BYTES FREE

     READY.
```

The reset handling program executed at power-up time, and the memory locations that point to it ($FFFC, $FFFD), had better be in ROM, rather than RAM. Why?

IRQ (Interrupt Request)

60 times a second the 6502 installed in a Commodore 64 receives a pulse on its IRQ pin. Like a reset, this causes the 6502 to drop what it's doing and do something else, but to *first save where it is now so that it can get back later*. This is accomplished by saving the program counter and P register on the stack. Once saved, the program counter is loaded with the address stored in locations $FFFE and $FFFF, the highest two locations in the memory map. The interrupt routine performs its function, and returns to the interrupted program with an RTI (ReTurn from Interrupt). RTI is like an RTS that pulls the status register first, and program counter second.

The Kernal's interrupt handling program takes care of three basic jobs: display control, keyboard scanning, and jiffy counting.

Such interrupt-activated programs are often called "background" tasks. Sixty times a second the 6502 in a C–64 stops working on the main, "foreground" task (TVC, at this moment) and fiddles around with jiffy counting and keyboard scanning. However, since the processor can pick up smoothly where it left off, and since the interrupt program consumes relatively little time, TVC appears to have the processor all to itself.

Interrupt programs must be careful to preserve registers that are altered during the course of the interrupt service routine. You can imag-

ine the fun of debugging a foreground program in which 60 times a second the X register is mysteriously zeroed. The only registers saved automatically during an interrupt are PC and P; if the interrupt program wants to use any of the others, it must save them in a safe place beforehand, and restore them just before returning to the main program.

Two bits of the P register are directly related to interrupt requests. The I flag, interrupt disable, is used to "mask out" interrupts. If I is set, the 6502 ignores signals on the IRQ pin. That's how IRQ gets its name; it requests, rather that demands, attention. You may want to disable interrupts, for example, when you're in the midst of handling an interrupt already.

BRK: The Whole Truth

Have you wondered why the P register has a B flag that doesn't have anything to do with borrow? Or why BRK is sometimes called a *software interrupt*?

BRK makes the 6502 behave as though an interrupt request had occurred on the IRQ pin. An indirect JMP is made to the same program, pointed to by $FFFE and $FFFF. The only difference is that the B flag of the status register is set by a BRK and cleared by an IRQ. This gives the interrupt handling program a way to determine if the break was due to software or hardware. If it finds the break flag set, it knows the interrupt was due to BRK. Otherwise, it has a bonafide hardware interrupt on its hands.

BRK is for debugging 6502 programs. By setting BRK instructions at key points in your program, you can usually find out what's working and what isn't. TVC's simulator doesn't execute a BRK the way a 6502 does; it treats BRK as a signal to stop execution. By they way, BRK cannot be masked by setting I, either in real life or with the simulator.

If the 6502 encounters a BRK while executing a subroutine via TVC's GO command, TVC regains control and updates the A, S, X, Y, P, and PC registers with the values they held at the moment of the BRK. By convention, the program counter is the address of the BRK instruction plus two.

Watch an Interrupt

Let's watch the simulator field an interrupt request.

```
INTPROG

$C000: CLI
       ST[illegible]
       JMP     $C000

C300:  STA     $FB
       PLA
       PHA
       ASL
       ASL
       ASL
       BMI     BREAK
       INC     $FE
BREAK: LDA     $FB
       RTI
```

Load INTPROG and begin executing it in step mode 1. At some point, request an interrupt by pressing 'I'. If the I flag is set, the simulator will ignore the request. Otherwise, when the current instruction finishes, TVC's 6502 simulator will respond to the command.

You'll see the program counter and status register pushed on the stack. The status register value pushed on the stack will have a clear B flag; the handling program will check this stack image of P to determine if this is a BRK or IRQ generated interrupt. Next, the interrupt disable flag is set. Unless we specifically clear it, interrupts will be locked out until the interrupt handling program finishes. The program counter then loads from locations $FFFE and $FFFF, in effect jumping to the handling program at $C300, which was loaded into memory along with INTPROG.

The handling program first checks to see if this is a hardware or software interrupt request. It saves A before doing anything else; a paramount duty of interrupt programs is preserving the foreground program's registers. Then we pull the byte at the top of the stack (the interrupted program's status register) into A, and immediately push it back. This gets the byte that used to be the status register into A where it can be examined. If we hadn't pushed the accumulator after the pull, the stack would have been messed up. RTI would have used the MSB of the return address as the value to put back in P, and who knows what would be pulled into the program counter.

Three ASL's of the accumulator slide bit 4 into bit 7. After considerable work, we've made the negative flag equal to the B flag at the moment of the interrupt. If set, we have a software break, and return to the calling program immediately. If reset, we're in the midst of an IRQ interrupt, and increment a one byte counter in page zero before returning. The return to the foreground program is smooth; everything is as it was before, as though the interrupt never happened. From the standpoint of the foreground program, the only evidence of the interrupt's occurrence is the ghostly increment of $FE.

Non-Maskable Interrupt

NMI is similar to IRQ, only it has a different vector ($FFFA) and it may not be ignored. The 6502 always responds to an NMI regardless of the I flag. In many designs (but not the C-64), NMI is connected to a power supply sensor. When the sensor gives warning that the incoming AC line has dropped below some minimum value, the time remaining to the system is short, maybe only a hundreth of a second. We can't afford to be polite and wait for another interrupt to finish. The NMI vector points to a program that may activate a backup power source, or simply close vulnerable files and wait quietly for the end.

In the C-64 the RESTORE key is tied directly to the 6502's NMI pin. Each time you press RESTORE, a non-maskable interrupt is generated. The Kernal's NMI handling program looks to see if the RUN/STOP key is down also, and if it is, control is transferred to a routine which warm starts the program in control, usually Basic. If RUN/STOP is not down, the Kernal quickly returns control to the needlessly interrupted program.

Because NMI always gets the computer's attention, the RUN/STOP RESTORE combination is able to break into infinite loops and similar problem conditions and restore control.

Signed Numbers

All the programs we've seen so far have assumed that the numbers being added, subtracted, decremented, etc. were always positive. Many times machine language programs face the same problem as the overdrawn checkbook, how to handle numbers less than zero. Or, put another way, what shows up in the accumulator when we subtract 6 from 3? You won't see any minus signs anywhere, that's for sure.

Any guesses as to how to represent negative numbers? (It has something to do with bit 7, hint, hint.) As a suggestion, how about using the lower 7 bits as the absolute value of a byte, and the 7th bit as a sign flag. Thus:

```
0011 1111 = + $3F
1011 1111 = - $3F

       and

0100 1010 = + $4A
1100 1010 = - $4A
```

That wasn't so bad, was it? Almost the way people do it—if there's no minus sign, numbers are positive; here, if we have a clear sign bit, we mean positive. A nice, sensible solution.

Using the most significant bit as a plus/minus indicator limits the range of values that can be represented with a single byte to -127 through +127. (Again the formula: $2^7 - 1 = 127$.) Two byte numbers can use the 7th bit of the most significant byte for the sign, with the remaining 15 bits storing the absolute value. This yields the range -32,767 through 32,767 (ring a bell somewhere about the storage limitations of Basic integer variables?).

But hold on. Even though this scheme has a certain pleasing logic, it has a nontrivial problem: *It doesn't work.* Adding 3 + (-6) should produce -3. Does it?

```
  0000 0011   (3)
+ 1000 0110  (-6)
  ---------
  1000 1001  (-9)
```

No. Any way you slice it, -9 is not -3. How about 26 + (-14)?

```
  0001 1010   (1A)
+ 1000 1110  (-0E)
  ---------
  1010 1000  (-28)
```

Not even close. And there's another problem. We have two bit patterns that mean zero: "Positive zero", 0000 0000, and "negative zero", 1000 0000. Ouch.

Logical, maybe—correct, uh-uh. Rather than subject you to a whole series of potential solutions that don't work, let us proceed immediately

to a way to represent negative numbers that *does* work, two's complement.

As with nonfunctional method #1, bit 7 still indicates whether a number is negative or positive. It's the other 7 digits that are handled differently. A two's complement is formed by complementing (reversing) each bit and adding one to the result. We represent –$19 with the *two's complement* form of positive $19.

```
 $19 = 0001 1001
-$19 = 1110 0110 + 1 = 1110 0111

 $64 = 0110 1000
-$64 = 1001 0111 + 1 = 1001 1000
```

While somewhat less logical than the first method, two's complement representation possesses the desirable property of actually working when we put it into action adding numbers. It also solves the problem of two zeros. There's just one, 0000 0000.

Performing the addition 3 + (-6):

First express -6 into two's complement form:

-6 = two's complement of 6 = two's complement of 0000 0110 =

1111 1001 + 1 = 1111 1010.

Now do the addition:

```
  0000 0011   (3)
+ 1111 1010  (-6)
  ---------
  1111 1101   (?)
```

Since the result has bit 7 set, we know the answer is negative, and by performing a two's complement to switch it to positive, we can see if we got the right answer.

Two's complement of 1111 1101 = 0000 0010 + 1 = 0000 0011 = 3.

It worked. We got -3 for an answer. Since the function of this book is to get you started in machine language, not to win you the George Boole Chair of Binary Studies at Stanford, there will be no rigorous proof at-

tempted here of why this method works. (Audible sigh of disappointment.)

To practice, use PRACTICEPROG to add one byte negative numbers to positive numbers. Represent negative numbers with two's complement form; if you're lazy (and/or smart), you'll use the calculator for this. Subtract the number you want in two's complement form from zero (e.g., to obtain the two's complement form of $3411, perform the subtraction: $0 – $3411).

A final note: For many, signed arithmetic proves to be one of the most elusive aspects of machine language. If this presentation left you more confused than enlightened, take comfort in the fact that most machine language programs don't need signed numbers. And when the day comes, six months or five years from now when you'll need to know it, I think you'll find you can pick it up.

Binary Coded Decimal

The Decimal flag (D) of the P register hasn't seen much action. In fact, except for a couple of sets and clears back in Chapter 7, we've ignored it entirely.

The D flag controls how the SBC and ADC instructions work. If reset, as it has been so far in all the demonstration programs (or should have been) SBC and ADC perform standard binary arithmetic. If D is set, the 6502 adds and subtracts using *binary coded decimal* (BCD) numbers.

BCD is a numbering system in a limbo midway between binary and decimal. Because you will almost certainly have no immediate use for additions and subtractions of binary coded decimal numbers, (unless you're planning to write a floating point package and are worried about rounding errors), no further mention of it will be made here. Just remember to keep this flag clear or all your adds and subtracts will be wrong. One of the first instructions executed when a C-64 comes to life during a power-on reset is CLD (Clear Decimal Mode). Leave it that way.

17. Putting It All Together

So far we've been using 6502 programs without much consideration to how they were produced in the first place. We said LOAD and there they were. Since the purpose of this manual is to get you *writing* machine language programs, it's about time we wrote a couple, taking ideas all the way to working 6502 programs.

Bubble Sort

The first program is a demonstration of a sorting technique called bubble sort. There are more sophisticated sorting techniques around, in fact, there aren't many *less* sophisticated, but when you're using machine language, and moderate amounts of data, there's no reason to get fancy.

A bubble sort works by systematically "bubbling" the lightest (smallest) numbers in an unsorted list to the top. We start at the bottom and compare each number with the value above it. If they're not in the right order already, they are swapped and we move up to the next pair. When we've worked our way to the top of the list, the smallest number is guaranteed to be at the top. Next, we repeat the entire process, except that we don't check the topmost number; we know it's the smallest already. After pass 2, the top two elements in the list are correct.

After $n - 1$ passes through an n element list, we are done. Let's work through a sample bubble sort of a four element list. We'll need to make three passes (4 - 1) through it, the first making three comparisons, the second two comparisons, the last, only one.

The arrow points to the lowest member of the pair under test.

```
Pass 1              34
                    19
                    77
              -->   22
```

Starting at the bottom: Compare 22 to 77. Since 22 is less than 77, bubble it up a notch by swapping it with 77. Advance the pointer. Now the list looks like this:

```
Pass 1           34
                 19
           -->   22
                 77
```

Compare 22 to 19. We don't need a swap this time. Move the pointer.

```
Pass 1           34
           -->   19
                 22
                 77
```

Compare 19 to 34. Swap. This completes one pass through the list. The smallest value in the list is now at the top. Move the pointer down to the bottom and repeat the process, only this time, stop one comparison sooner, since the top value is already correct.

```
Pass 2           19
                 34
                 22
           -->   77
```

Compare 77 to 22. No swap necessary, advance the pointer.

```
Pass 2           19
                 34
           -->   22
                 77
```

Compare 22 to 34. Swap. End of Pass 2 (top 2 values now correct).

```
Pass 3           19
                 22
                 34
           -->   77
```

The last pass requires only one comparison, 77 to 34. No swap needed. The list is now sorted.

That's the sorting method, or algorithm, known as bubble sort. With a minor change to the comparison step we could just as easily sort the list in reverse order.

Basic Bubble Sort

Back to the machine language payroll system. To comply with government form 3212.1-C, Binary, Inc. must submit a sorted list of the ages of its 256 employees. A Basic attack on the problem might be:

```
FOR COUNT = 0 TO 254
  (WORKING FROM BOTTOM TO TOP, TEST EVERY ADJACENT PAIR
   AND SWAP IF NECESSARY)
NEXT COUNT
```

Fleshing out the interior of the loop:

```
FOR POINTER = 0 TO (254-COUNT)
     IF ARRAY(POINTER) < ARRAY(POINTER+1) THEN SWAP
NEXT POINTER
```

One of bubble sort's few merits is that we don't have to check the top of the list each time. That's why the inner loop goes to (254-COUNT).

The swap of an out-of-order pair becomes:

```
TMP = ARRAY(POINTER)
ARRAY(POINTER) = ARRAY(POINTER+1)
ARRAY(POINTER+1) = TMP
```

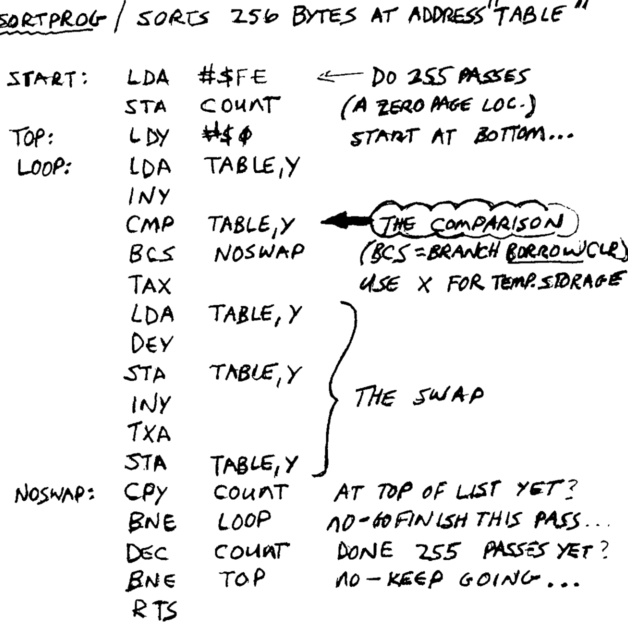
Putting it all together, here's a program that could do the job.

```
FOR COUNT = 0 TO 254
     FOR POINTER = 0 TO (254 - COUNT)
          IF ARRAY(POINTER) < ARRAY(POINTER+1) THEN GOSUB SWAP
     NEXT POINTER
NEXT COUNT

SWAP: TMP = ARRAY(POINTER)
      ARRAY(POINTER) = ARRAY(POINTER+1)
      ARRAY(POINTER+1) = TMP
      RETURN
```

This Basic rendition of bubble sort is surprisingly simple and appallingly slow. Even for short arrays there's a lot of comparing and swapping to do. In the neighborhood of 32,000 comparisons for a list of 256 numbers, and roughly half that many swaps, depending on how well the list is sorted already.

Next Step: Assembly Language

Now that we've massaged the bubble sort algorithm into program form, let's translate it into the mnemonics and operands of the 6502 instruction set. We're going to use labels in places because we don't want to tie ourselves down to real addresses yet.



This form of the program is called *assembly language*. It's not machine language yet—the 6502 can't cope with "STA COUNT", anymore than it can understand the whispered command, "Store your accumulator in address $FB, please". Before SORTPROG can be run, we must "assemble" it into the 1's and 0's of 6502 machine language.

```
CØØØ: A9  FE
CØØ2: 85  FB
CØØ4: AØ  ØØ ↙ØØ ↙C1
CØØ6: B9  ~~X~~  ~~ØØ~~
CØØ9: C8
CØØA: D9  ØØ  C1
CØØD: BØ  [ØD]
CØØF: AA
CØ1Ø: B9  ØØ  C1
CØ13: ~~AA~~ (88)
CØ14: 99  ØØ  C1
CØ17: C8
CØ18: 8A
CØ19: 99  ØØ  C1
CØ1C: C4  FB
CØ1E: DØ  [E6]
CØ2Ø: C6  FB
CØ22: DØ  EØ
CØ24: 6Ø
```

Assembling even a short program into object code is a tedious job. After a half hour of flipping pages in reference manuals, calculating relative branch values, and replacing labels with addresses, pretty soon learning machine language doesn't seem like such a good idea after all.

Once we've translated the source program into a flock of bytes, (call it the *object* program), we must edit these numbers into the computer. And hope we get every byte right, and don't change *STA*'s into *LDA*'s along the way.

A test to see how good a job we've done of assembling and entering is to disassemble memory where we placed the object program, and see if the result resembles the original source program. It probably won't and we'll need to make a patch or two. Even once we get it entered right, the program still won't work if the original source program had logical errors. If we do much rearranging at all of the source program, we'll have to re-assemble from scratch.

There's a better way.

The phase of the machine language programming process least suited to the talents of humans (and best suited to those of a computer) is the assembly itself. Wouldn't it be nice if we had a program that could assemble source programs automatically?

Happily, such programs, called *assemblers*, exist. An assembler converts 6502 assembly language (source) programs into 6502 machine language (object) programs. One fly in the ointment is that you won't have much luck getting an assembler to make sense of a pen and paper source program. You'll need a special program called an *editor*, a programmer's word processor, to produce the source program. Many assemblers include an editor as part of the package. Using an editor is ultimately faster than writing on paper, although it takes some getting used to.

```
 1          ORG    $C000
 2
 3  TABLE   EQU    $C100
 4  COUNT   EQU    $FB
 5
 6  START   LDA    #$FE       ; PREPARE FOR 255 PASSES
 7          STA    COUNT
 8  TOP     LDY    #$00       ; START AT BOTTOM OF LIST
 9  LOOP    LDA    TABLE,Y    ; GET ARRAY(Y)
10          INY
11          CMP    TABLE,Y    ; COMPARE W/ ARRAY(Y+1)
12          BCS    NOSWAP     ; BRANCH IF NO BORROW
13          TAX               ; ELSE SWAP
14          LDA    TABLE,Y
15          DEY
16          STA    TABLE,Y
17          INY
18          TXA
19          STA    TABLE,Y    ; SWAP COMPLETE
20  NOSWAP  CPY    COUNT      ; AT TOP YET?
21          BNE    LOOP       ; NO, COMPARE NEXT PAIR
22          DEC    COUNT      ; DONE 255 PASSES YET?
23          BNE    TOP        ; NO...
24          RTS
```

This editor-produced source program for SORTPROG looks remarkably like the hand written version, with a few exceptions. Every line is numbered. The editor that produced it uses line numbers as a means of editing, the same way Basic does.

Semicolons are this assembler's equivalent of Basic's REM. All lines beginning with an semicolon are comments intended to enlighten the person reading the source program. The assembler ignores them.

Lines 3 and 4 are *equates*. ".EQ" is a *pseudo op* that makes the assembler associate the name "COUNTR" with the number $FA. In writing the source program, you can use the name instead of the number. Notice that the assembler doesn't generate any bytes for .EQ statements. Pseudo ops are mnemonics for controlling the assembler, not the 6502. The ORG pseudo op in line 1 tells the assembler to generate code starting with address $C000.

Once the source program is ready, in a separate step the assembler translates it into object code. A short program like SORTPROG assembles in seconds, with guaranteed accuracy. Is that better than half an hour, and making mistakes to boot? (Rhetorical question.) Once assembled, we can save the object program to disk, or run it, or whatever.

```
                   1           ORG     $C000
                   2
                   3  TABLE    EQU     $C100
                   4  COUNT    EQU     $FB
                   5
C000: A9 FE        6           LDA     #$FE        ; PREPARE FOR 255 PASSES
C002: 85 FB        7           STA     COUNT
C004: A0 00        8  TOP      LDY     #$00        ; START AT BOTTOM OF LIST
C006: B9 00 C1     9  LOOP     LDA     TABLE,Y     ; GET ARRAY(Y)
C009: C8          10           INY
C00A: D9 00 C1    11           CMP     TABLE,Y     ; COMPARE W/ ARRAY(Y+1)
C00D: B0 0D       12           BCS     NOSWAP      ; BRANCH IF NO BORROW
C00F: AA          13           TAX                 ; ELSE SWAP
C010: B9 00 C1    14           LDA     TABLE,Y
C013: 88          15           DEY
C014: 99 00 C1    16           STA     TABLE,Y
C017: C8          17           INY
C018: 8A          18           TXA
C019: 99 00 C1    19           STA     TABLE,Y     ; SWAP COMPLETE
C01C: C4 FB       20  NOSWAP   CPY     COUNT       ; AT TOP YET?
C01E: D0 E6       21           BNE     LOOP        ; NO, COMPARE NEXT PAIR
C020: C6 FB       22           DEC     COUNT       ; DONE 255 PASSES YET?
C022: D0 E0       23           BNE     TOP         ; NO...
C024: 60          24           RTS
```

That's how SORTPROG came into existence. Now, how does it work?

Line by line:

Lines 6–7: Initialize counter to $FE. Counts passes through the list. We're done when this is reduced to zero. We count down, instead of up, to simplify the step that keeps us from checking topmost elements already known to be in the right order.

Line 8: Starting point of outer loop. Puts us at the bottom of the list for the start of each pass. The Y register is the pointer.

Line 9: Starting point of the inner loop, where we work our way up, pair by pair, until we reach the top.

Lines 9–19: The actual comparing and swapping. Test each consecutive pair. If the byte with the lower address is smaller than the addressed byte, swap them.

Line 20: Have we gone all the way through the list yet? Remember, we don't need to go any higher than we've already done passes.

Line 22: An entire pass has been completed. If this was the 255th pass, we're done. Otherwise, make another pass.

Load SORTPROG. It loads complete with the unsorted employee age array at $C100 - $C1FF. Unless you've really got a handle on every step, use the simulator for a couple of comparisons. Understanding the comparison step requires understanding the borrow flag. Take a whole day if you must to get it down pat, but do it, once and for all. When you get around to GOing it, you'll find that SORTPROG handles these 32,000 comparisons and 16,000 swaps in about one second.

How old is the youngest employee of Binary, Inc.? The oldest?

Extra Credit

Change SORTPROG's five absolute, Y instructions from STA/LDA $C100,Y to STA/LDA $C400,Y. Now the page of memory in SORTPROG's sights is screen memory, the first six and a half lines of the TVC display...

Now for Something Completely Different

What kind of program should we write next? Something with a little more *suss*... No, already done that. No, too complicated. How about... no, too easy. I've got it: Play music with the keyboard. Catchy name: "ASCII Organ". While we're at it, we'll flash colored bars on the screen along with the music, *Close Encounters*-fashion.

Good idea, but somewhat open-ended. Let's firm up ASCII Organ's design, specifying in English as precisely as possible what the program will and will not do.

ASCII Organ turns the keyboard into a 12 note organ, assigning notes to keys on the top row, from the "1" key on the left (lowest note) to the minus key on the right (highest note). Each key will be a half tone higher in pitch than the key to its left. All programs need an escape route, and ASCII Organ's will be the F1 key.

The screen will display a bar of color for each note, low notes at the bottom, high notes at the top. Since we have 12 notes, and 25 lines on the screen, we'll use two lines for each color. Note 1 will appear on lines 23 and 22 as color 1, white. Note 2 uses lines 21 and 20, and is red, and so on, up to note 12, lines 1 and 0, color "gray 2". These are the standard colors values listed in appendix B.

As important as saying what the program will do is saying what it won't do. ASCII organ won't play more than one note at a time. It won't control duration. Each note will last the same length of time, 1/2 second, regardless of how long a key is pressed. It also won't control volume. Hey, I said it had a catchy name, not that it would put Hammond out of business.

Before we can describe a machine language program to do these things, sit still for a quick lecture on C-64 sound.

Making Sound with the Commodore 64

The C-64 has a chip named SID (Sound Interface Device) devoted to generating sound. Much as VIC, the video controller, figures out what to do with the display by looking at screen memory, SID makes sounds according to the values in I/O addresses $D400 - $D41C. Although entire books have been written on the subject of making sounds with the C-64, an undemanding musician like ASCII Organ barely needs to lift a finger to make music. In fact, you can generate tones with EDIT, just by writing into a few SID control locations.

Four writes is all it takes. Place $61 at $D400, $08 at $D401, $F0 at $D406, and $21 at $D404. Make sure you're a master before writing to these addresses, or the values won't get through TVC's protective check.

$D400 is a 16 bit control register that tells SID the pitch to play; placing $0861 (4145 decimal) there makes SID generate a 128 hertz tone. To calculate what pitch a given value in $D400-$D401 will produce, multiply by .0596. $D404 controls the timbre of the tone. $D406 controls sustain. SID uses $D404's bit position zero as an on-off switch. The instant we write to $D404, the selected note begins to play.

Turn SID off by writing $00 to $D404.

A peculiarity of these SID control locations is that they always read as zero no matter what values you've placed there. To appreciate what a great tone producer SID is, consider what an Apple II program has do to generate the simplest of sounds. That computer doesn't have tone generation hardware, only a speaker tied to an I/O address. When you read that location, the speaker, almost imperceptibly, says: "click". That's it—no beep, no tone, just "click".

To create a sustained tone, one calls a machine language program that repeatedly clicks the speaker hundreds or thousands of times a second. The pitch of the emitted tone depends on how long you wait between clicks. If you wait 2,000 microseconds (1/500 of a second) between clicks, you get a note of 500 hertz. The only way to waste exactly 2,000 microseconds between clicks is to run a delay loop, the time wasting properties of which are known exactly. There's no clock ticking away to help you.

To make matters worse, you must also time how long you want the 500 hertz note to last. An Apple program playing a simple tone is two carefully timed loops, one running inside the other—an inner, pitch loop, and an outer, duration loop.

```
LOOP: CLICK SPEAKER
      GOSUB DELAY
      INCREMENT COUNTER; IF NOT DONE, GOTO LOOP
```

And waveform control? Multiple voices? Volume control? Forget it. So appreciate how much work SID does for you. If he didn't do it, guess who'd have to.

Now that we know a nickel's worth about making sounds on the C-64, the next step to making ASCII Organ a reality is to concoct a series of steps for making it happen. Again, let's use Basic (in a very syntax-relaxed form) to express the problem.

```
        GOSUB   INITIALIZE SCREEN
        GOSUB   INITIALIZE SID
START:  GOSUB   GETKEY
           IF KEY = F1 THEN END
           IF NO KEY PRESSED GOTO START
        GOSUB   CHKKEY
           IF KEY UNDEFINED, GOTO START
        GOSUB   BEEP/DRAW BAR
        GOTO    START
```

We're using a method called top-down programming. We start at the highest, overall level of the problem, and put off details into subroutines. You don't have to know how to make a tone or check a fire button to write the main level of ASCII Organ; you make the assumption that such a function can be programmed, give it a name, and specify its calling conventions.

Once the top level is solid, you move down to firm up the next level, and so on, until pretty soon, you're down at the bedrock of the problem. For example, subroutine BEEP/DRAW BAR has sub-levels of its own.

```
SUBROUTINE BEEP

LOAD SID FREQUENCY REGISTER
START PLAYING NOTE
GOSUB DRAW BAR
TURN OFF NOTE
RETURN
```

By putting DRAW BAR into yet a third level, we keep BEEP clean and simple.

```
SUBROUTINE DRAW BAR

DRAW BAR
WASTE TIME FOR 1/2 SECOND
ERASE BAR
RETURN
```

Load ORGANPROG. Before we get into a detailed discussion of it, GO the program and play around for a bit. (We put a JSR $C000 at $C200 for your GOing convenience.) For instructions, consult the definition. The first person to send in a tape of *Moonlight Sonata* played on the ASCII Organ wins a special No Prize and a hearty "Well done".

Now, How Does It Work?

Refer to the assembly language listing at the end of this chapter as we work through ASCII Organ. The simulator can assist your efforts at understanding the program, but keep a couple of things in mind.

Don't try to simulate every step. Not only would this take days, some routines (GETIN, for one), won't work at all under the simulator. Instead, step through the main level of the program, using GO to execute selected subroutines all at once.

Second, the display used by ASCII Organ is the same one used by TVC (only one per computer). As a result, TVC will interfere with ASCII Organ's efforts, and vice versa. Remember the RESTORE instruction.

Routine by Routine

Subroutine INITSCRN is surprisingly complicated. Why isn't it a simple rehash of that old favorite CLEARPROG, you ask? I'll tell you, after a digression on character colors.

Color Memory

When a Commodore 64 is first turned on, characters are presented as light blue letters on a dark blue background. TVC displays white characters against a black background. Why? How? (It has nothing to do with TVC's relocation of screen memory from $0400 to $C400.)

The answer concerns a hidden range of memory that serves as input for VIC, the display controller chip, just as screen memory does. There are 1,024 bytes of RAM from $D800–$DBFF (labeled as I/O in TVC's memory maps), organized just like screen memory, called color RAM. The least significant nibble of each location in this range tells VIC what color to display characters in.

The low order four bits of location $D800 determine the color of the character in row 0, column 0. Location $D801 is for row 0, column 1, and so on. The Kernal wrote $0E, the code for light blue, into color RAM at power-up time. TVC changed these 1,000 locations to 3, producing cyan characters (see appendix B). If you change $D801 to 2, you'll see the character in that cell (the *v* in *Visible Computer*) change to red.

The background color for the whole screen is controlled by a single VIC control address: $D021. EDIT a 3 there. The display blanks out. Why? TVC is still sitting in EDIT, waiting for you to do something about address $D022—but VIC is now displaying cyan characters against a cyan background. EDIT $D021 back to 0 (cursor left, then 0, then return).

What's all of this got to do with ASCII Organ? To display a solid line of red on the display, two things have to happen. First, every location in color memory corresponding to where on the screen the red bar will be must have a least significant nibble of 2. That takes care of the red part of the problem. Next, how do we make it a solid bar, and not just a line of red commas or question marks?

We'll use character number $A0, the inverse blank space. Since $20 is the space character, nothing appears on the screen when this character is output. If we invert it, by setting the high bit ($20 is 0010 0000; $A0 is 1010 0000), we get a character that is solid color. Parts of the TVC display are built out of inverse spaces, such the message window. If 40 inverse space characters are placed in a row that color memory has determined will be red, presto: a solid red bar.

Back to Our Story...

INITSCRN clears the screen through a double whammy of writes to both screen and color memory. First it sets the background color to black (even though it probably was already). Next it points zero page pair $FB,$FC to the first location in screen memory ($C400), and $FD,$FE to the first spot in color memory ($D800). Then it enters a

loop, similar to CLEARPROG's, that cranks through 1,024 memory locations. Each color memory address gets a zero, character color black. Since the screen background color was set to black earlier, this has the effect of erasing the screen.

Meanwhile, every position in screen memory is getting an inverse space. As soon as color memory is made something other than black, solid blocks of color will appear. When four pages of screen and color memory writes are finished, INITSCRN returns.

INITSID

This second initializing subroutine makes sure SID is ready to go when the main loop needs it. First we write zeros into SID's $1C control addresses, to put SID into a suitably turned off, known state. $D418, SID's volume address, is set to maximum, and we return. Where possible, the assembler pitches in to make the programmer's work easier. In line 91, the operand SID+4 is converted into address $D404 at assemble time.

GETIN

is the Kernal function we learned about in Chapter 14. It returns with the ASCII value of the depressed key in the accumulator. If 0 returns in the accumulator, no key was pressed.

CHKKEY

CHKKEY helps out the already complicated BEEP/DRAW subroutine by reducing the confusion of possible keystrokes returned by GETIN. When CHKKEY finishes, location $FC, also known as NOTE, holds a value from 0 through 12. If 0, the key pressed is not defined to mean anything in ASCII Organ (e.g., return, or "T"). If $FC is nonzero, it contains the selected note.

These checks are a surprising amount of work. First we test for the zero, plus, and minus keys, as their value as notes (10–12) bears no resemblance to their ASCII value as keys ($30, $2B, and $2D). The number keys, 1 - 9, on the other hand, have ASCII values nicely ordered from $31 to $39, and can be tested as a group.

First we test for the minus key, ASCII code $2D. If that's what really is in the accumulator, CHKKEY stores a $0C (note 12) in $FC, and

returns. If it wasn't the minus key, we test individually for the plus and zero keys; they become notes 11 and 10, respectively.

If we still haven't found a match, there are two possibilities remaining. It's either one of the continuous number keys, or an undefined key. The program finds out by assuming that it is a number key.

The number keys have ASCII values from $31 to $39. We subtract $30 from the value in the accumulator. If there was in fact a number key stored there, the accumulator would now be in the range 1–9 (we tested specifically for value $30, "0", earlier). This assumption is tested by comparing the accumulator with #$0A; if this compare generates a borrow, we have a number key. The BCS test would then fail (set borrow = clear carry), and we store the key value (already in the range 1–9) into NOTE.

If the BCS passed, the key is undefined, and we leave CHKKEY without having changed NOTE from the zero we wrote there in the second instruction of the subroutine. CHKKEY has now set the table for BEEP, by turning a bewildering array of possible keystroke values into a tidy package at $FC.

BEEP

The glamour girl of ASCII organ, subroutine BEEP plays the tone corresponding to the value in NOTE, and calls subroutine DRAW to put the colored bar on the screen. Making VIC play a specific pitch requires placing the frequency values for that pitch into the 16 bit pitch control register at $D400,$D401. Beep uses two short arrays to store these frequency values, PITCHL for the LSB, PITCHH for the MSB, and uses X as an index.

The tables store pitch values from C through B.

	Note	Control Value			Pitch (Hz)
		MSB	LSB	Decimal	
1	C	08	61	2145	128
2	C#	08	E1	2273	135
3	D	09	68	2408	143
4	D#	09	F7	2551	152
5	E	0A	8F	2703	161
6	F	0A	30	2864	167
7	F#	0B	DA	3034	181
8	G	0B	8F	3215	192
9	G#	0C	4E	3406	203
10	A	0D	18	3608	215
11	A#	0E	EF	3823	228
12	B	0F	D2	4050	241

We used the HEX pseudo op to generate these tables at the end of the program. The assembler turns the line:

202 PITCHL HEX 00,61,E1,68,F7,8F,30,DA

from text into bytes at assembly time, just as it does 6502 instructions. It also automatically makes the connection between PITCHL and the address of the table, so the source program can reference the symbolic name of the table, not a number. The first byte in each table is zero, since the note values used to index this array range from 1–12, not 0–11. The ORG at line 196 makes the tables start on a nice, even page boundary, and isn't really necessary.

For example, suppose the plus key was pressed. That's note number 11, so we'd read the 11th element of each table: $EF from PITCHL, $0E from PITCHH. That's pitch value $0EEF.

After the pitch has been set, we turn the tone on with a write to $D404. BEEP next calls DBAR to draw a bar in the appropriate spot in the appropriate color. Note that the only thing BEEP does after DBAR returns is shut off the tone by placing a zero in $D404. Our guiding design says notes last one-half second, and since note duration control clearly doesn't happen in BEEP, this timing must happen somewhere else. Specifically, in DBAR.

DBAR

DBAR is three levels down in our methodical division of the problem. First it sets the jiffy clock to zero, using Kernal function SETTIM. SETTIM is RDTIM in reverse. The values in Y, X, and A are copied into the clock's addresses. Next, DBAR uses NOTE to index into yet another pair of arrays. The values retrieved from tables BSL (Bar Start Low) and BSH are the LSB and MSB, respectively, of the first address in color memory that needs changing.

Again, assume that the plus key has been pressed. We have a note value of 11, and a 228 hertz (3823 * .0596) tone playing. The 11th entry in array BSL is $50; the 11th entry of BSH is $D8. $D850 is the first color memory location we'll modify to draw the bar for this note. For note 11 we want a bar drawn on lines 2 and 3 of the display (remember, high notes at the top). $D850 is the 1st column of row 2; the *first* of the 80 locations in color memory that need modification.

$D850 goes into pointer pair $FD,$FE, the note number in A, and we delegate the job of actually writing to color memory to subroutine BAR. Before we look at BAR, look at what DBAR is up to after BAR returns. First it reads the jiffy clock with RDTIM, and checks A (LSB of time counter) to see if 30 jiffies have gone by yet. If they haven't, we repeat the test. During ASCII Organ execution, 95% of the processor's time is spent here in repeated RDTIM calls.

When time is up, BAR is called again, this time with a bar color of zero (black), to erase the colored bar drawn in the first call. Actually, we're being sneaky in how we call BAR this second time. By jumping to it, rather than JSRing, we cause the RTS at the end of BAR to return us all the way to BEEP.

BAR

We're now down four subroutines deep. (The main program called BEEP which called DBAR which called BAR.) Here's where ASCII Organ actually puts color on the screen, alternately displaying and erasing the inverse spaces stored in screen memory. BAR writes the note/color value in the accumulator to $50 locations in color memory, starting with the address loaded in pointer pair $FD, $FE. $50 is 80 decimal, representing two consecutive rows of screen locations.

Top down programming keeps every module of ASCII Organ more or less equal in complexity. A big part of the skill of programming is knowing how to carve apparently monolithic problems into manageable slices. Your ability to do this will improve with practice.

ASCII ORGAN LISTING

```
                     1    **********************************
                     2    *                                *
                     3    *    A S C I I   O R G A N       *
                     4    *                                *
                     5    *                                *
                     6    *    BY:  JIM BLACKSHEAR         *
                     7    *              AND               *
                     8    *         BRIAN BOULDIN          *
                     9    *                                *
                    10    **********************************
                    11
                    12
                    13
                    14             ORG     $C000
                    15
                    16    NOTE     EQU     $FC
                    17    BGCOLOR  EQU     $D021
                    18    SID      EQU     $D400
                    19    VOLUME   EQU     $D418
                    20    SETTIM   EQU     $FFDB
                    21    RDTIM    EQU     $FFDE
                    22    GETIN    EQU     $FFE4
                    23
                    24
                    25
                    26    **********************************
                    27    *                                *
                    28    *         MAIN PROGRAM           *
                    29    *                                *
                    30    **********************************
                    31
                    32
C000: 20 1F C0      33             JSR     INITSCRN ; INIT SCREEN & COLOR MEM
C003: 20 47 C0      34             JSR     INITSID  ; INIT SID
C006: 20 E4 FF      35    TOP      JSR     GETIN    ; GET KEYSTROKE
C009: C9 00         36             CMP     #$00     ; 0 = NO KEY PRESSED
C00B: F0 F9         37             BEQ     TOP
C00D: C9 85         38             CMP     #$85     ; QUIT IF F1 PRESSED
C00F: F0 0D         39             BEQ     QUIT
C011: 20 61 C0      40             JSR     CHKKEY   ; TURN KEYSTROKE INTO 0-12
C014: A5 FC         41             LDA     NOTE     ; 0 = UNDEFINED KEY
C016: F0 EE         42             BEQ     TOP
C018: 20 8A C0      43             JSR     BEEP     ; PLAY TONE, DRAW BAR
C01B: 4C 06 C0      44             JMP     TOP      ; AD INFINITUM...
C01E: 60            45    QUIT     RTS
                    46
                    47
                    48    **********************************
                    49    *                                *
                    50    *         INIT SCREEN            *
                    51    *                                *
                    52    **********************************
                    53
                    54
C01F: A9 00         55    INITSCRN LDA     #$00      ;  SET BACKGROUND COLOR
C021: 8D 21 D0      56             STA     BGCOLOR   ;  TO BLACK
C024: A9 C4         57             LDA     #$C4      ;  SET UP TWO POINTER PAIRS
```

```
C026: A2 D8     58            LDX   #$D8
C028: A0 00     59            LDY   #$00      ; ($FB,$FC) --> $C400 (SCREEN)
C02A: 84 FB     60            STY   $FB
C02C: 85 FC     61            STA   $FC       ; ($FD,$FE) --> $D800 (COLOR)
C02E: 84 FD     62            STY   $FD
C030: 86 FE     63            STX   $FE
C032: A2 04     64            LDX   #$04      ;  DO FOUR PAGES OF WRITES
C034: A9 00     65   LOOP1    LDA   #$00      ;  STORE BLACK TO
C036: 91 FD     66            STA   ($FD),Y   ;  COLOR MEMORY,
C038: A9 A0     67            LDA   #$A0      ;  INVERSE SPACES
C03A: 91 FB     68            STA   ($FB),Y   ;  TO SCREEN MEMORY
C03C: C8        69            INY             ;  DO A FULL PAGE
C03D: D0 F5     70            BNE   LOOP1
C03F: E6 FC     71            INC   $FC       ;  ADJUST MSB OF POINTERS
C041: E6 FE     72            INC   $FE
C043: CA        73            DEX
C044: D0 EE     74            BNE   LOOP1     ;  DONE AFTER FOUR PAGES
C046: 60        75            RTS
                76
                77
                78   **********************************
                79   *                                *
                80   *          INIT SID              *
                81   *                                *
                82   **********************************
                83
                84
C047: A9 00     85   INITSID  LDA   #$00
C049: A2 1C     86            LDX   #$1C      ;  ZERO ALL $1C
C04B: 9D 00 D4  87   LOOP2    STA   SID,X     ;  SID CONTROL ADDRESSES
C04E: CA        88            DEX
C04F: D0 FA     89            BNE   LOOP2
C051: A9 20     90            LDA   #$20      ;  WRITE $20 TO
C053: 8D 04 D4  91            STA   SID+4     ;  TIMBRE CONTROL LOCATION
C056: A9 F0     92            LDA   #$F0      ;  WRITE $F0 TO SUSTAIN
C058: 8D 06 D4  93            STA   SID+6     ;  CONTROL LOCATION
C05B: A9 0F     94            LDA   #$0F      ;  SET VOLUME TO MAX.
C05D: 8D 18 D4  95            STA   VOLUME
C060: 60        96            RTS
                97
                98
                99   **********************************
                100  *                                *
                101  *          CHECK KEY             *
                102  *                                *
                103  **********************************
                104
                105
C061: A2 00     106 CHKKEY    LDX   #$00      ; INITIALIZE NOTE
C063: 86 FC     107           STX   NOTE
C065: C9 2D     108           CMP   #$2D      ; IS IT MINUS KEY?
C067: D0 05     109           BNE   PLUS      ; NOPE...
C069: A9 0C     110           LDA   #$0C      ; IF IT WAS, STORE 12
C06B: 85 FC     111           STA   NOTE      ; IN NOTE  AND RETURN
C06D: 60        112           RTS
C06E: C9 2B     113  PLUS     CMP   #$2B      ; PLUS KEY?
C070: D0 05     114           BNE   ZERO      ; NAW...
C072: A9 0B     115           LDA   #$0B      ; BUT IF IT WAS, PLUS
C074: 85 FC     116           STA   NOTE      ; IS NOTE 11
C076: 60        117           RTS
C077: C9 30     118  ZERO     CMP   #$30      ; HOW ABOUT 0 KEY?
C079: D0 05     119           BNE   CNVT      ; BRANCH IF NOT
C07B: A9 0A     120           LDA   #$0A      ; 0 KEY IS NOTE 10
C07D: 85 FC     121           STA   NOTE
C07F: 60        122           RTS
C080: 38        123  CNVT     SEC             ; CLEAR BORROW
C081: E9 30     124           SBC   #$30      ;
C083: C9 0A     125           CMP   #$0A      ; IF A < 10, A=NUMBER KEY
C085: B0 02     126           BCS   EXIT      ; ELSE, KEYSTROKE UNDEFINED
C087: 85 FC     127           STA   NOTE
C089: 60        128  EXIT     RTS
                129
```

```
                130
                131  **********************************
                132  *                                *
                133  *             BEEP               *
                134  *                                *
                135  **********************************
                136
                137
C08A: A6 FC     138  BEEP     LDX   NOTE       ; GET NOTE PRESSED
C08C: BD 00 C1  139           LDA   PITCHL,X ; READ LSB OF PITCH
C08F: BC 10 C1  140           LDY   PITCHH,X ; AND MSB FROM TABLE...
C092: 8D 00 D4  141           STA   SID        ; STORE IN BOTH
C095: 8C 01 D4  142           STY   SID+1      ; HALVES OF PITCH REG.
C098: A0 21     143           LDY   #$21       ; TURN ON GATE
C09A: 8C 04 D4  144           STY   SID+4      ; TONE STARTS NOW
C09D: 20 A6 C0  145           JSR   DBAR       ; DRAW THE BAR
C0A0: A0 00     146           LDY   #$00       ; RETURNS IN 30 JIFFIES
C0A2: 8C 04 D4  147           STY   SID+4      ; SHUT OFF TONE
C0A5: 60        148           RTS
                149
                150
                151  **********************************
                152  *                                *
                153  *          DRAW BAR              *
                154  *                                *
                155  **********************************
                156
                157
C0A6: A9 00     158  DBAR     LDA   #$00       ; RESET JIFFY CLOCK
C0A8: AA        159           TAX
C0A9: A8        160           TAY
C0AA: 20 DB FF  161           JSR   SETTIM
C0AD: A6 FC     162           LDX   NOTE       ; GET NOTE
C0AF: BD 20 C1  163           LDA   BSL,X      ; READ DATA FOR COLOR
C0B2: BC 30 C1  164           LDY   BSH,X      ; MEM POINTER PAIR $FD,$FE
C0B5: 85 FD     165           STA   $FD        ; FROM TABLE
C0B7: 84 FE     166           STY   $FE
C0B9: 8A        167           TXA              ; A TELLS BAR WHAT COLOR TO
C0BA: 20 C9 C0  168           JSR   BAR        ; DRAW; $FD,$FE TELLS WHERE
C0BD: 20 DE FF  169 WAIT      JSR   RDTIM      ; CHECK THE CLOCK
C0C0: C9 1E     170           CMP   #$1E       ; RETURN WHEN LSB
C0C2: D0 F9     171           BNE   WAIT       ; IS 30 DECIMAL...
C0C4: A9 00     172           LDA   #$00       ; DRAW BAR IN BLACK
C0C6: 4C C9 C0  173           JMP   BAR        ; BAR'S RTS RETURNS TO MAIN. P.
                174
                175
                176  **********************************
                177  *                                *
                178  *          BAR                   *
                179  *                                *
                180  **********************************
                181
                182
C0C9: A0 00     183  BAR      LDY   #$00
C0CB: 91 FD     184  BAR1     STA   ($FD),Y  ; WRITE COLOR VALUE TO
C0CD: C8        185           INY              ; COLOR MEM
C0CE: C0 50     186           CPY   #$50       ; DO 50 LOCATIONS
C0D0: D0 F9     187           BNE   BAR1
C0D2: 60        188           RTS
                189
                190  **********************************
                191  *                                *
                192  *          DATA TABLES           *
                193  *                                *
                194  **********************************
                195
                196           ORG   $C100
                197
                198   ****************
                199   *  PITCH LSB   *
                200   ****************
```

```
                 201
C100: 00 61 E1   202  PITCHL   HEX   00,61,E1,68,F7,8F,30,DA
      68 F7 8F
      30 DA
C108: 8F 4E 18   203           HEX   8F,4E,18,EF,D2,00,00,00
      EF D2 00
      00 00
                 204  *****************
                 205  *  PITCH MSB    *
                 206  *****************
                 207
C110: 00 08 08   208  PITCHH   HEX   00,08,08,09,09,0A,0B,0B
      09 09 0A
      0B 0B
C118: 0C 0D 0E   209           HEX   0C,OD,0E,0E,0F,00,00,00
      0E 0F 00
      00 00
                 210  ****************************
                 211  * BAR STARTING ADDRESS LSB *
                 212  ****************************
                 213
C120: 00 70 20   214  BSL      HEX   00,70,20,D0,80,30,E0,90
      D0 80 30
      E0 90
C128: 40 F0 A0   215           HEX   40,F0,A0,50,00,00,00,00
      50 00 00
      00 00
                 216  ****************************
                 217  * BAR STARTING ADDRESS MSB *
                 218  ****************************
                 219
C130: 00 DB DB   220  BSH      HEX  00,DB,DB,DA,DA,DA,D9,D9
      DA DA DA
      D9 D9
C138: D9 D8 D8   221           HEX  D9,D8,D8,D8,D8
      D8 D8
```

18.
Where Do I Go From Here?

Buy an assembler. Quick. Now that you know what one can do, never again waste time looking up opcodes or calculating relative branches. Two assemblers available for the Commodore 64 are:

Merlin 64

Roger Wagner Publishing, 10761 Woodside Avenue, Suite E, Santee, CA 92701.

Commodore 64 Macro Assembler Development System

Commodore Business Machines, Professional Computer Division, 487 Devon Park Drive, Wayne, PA 19087.

Whichever assembler you decide on, don't expect to be doing great things with it the first day you peel off the wrapper. With all this power come a host of new things to learn. You'll have to adjust to the editing commands, the keystrokes for deleting, adding, and rearranging lines in your source program. You may have to unlearn some commands you learned to run a word processing program.

You'll have to learn your assembler's pseudo ops, those special mnemonics that are not 6502 instructions, but rather commands for the assembly process to follow, such as determining where in memory a source program will be assembled to run. Expect to work as hard learning to effectively use an assembler as you did getting this far in machine language.

I Used My Assembler to Write a Program But it Doesn't Work and I Don't Know Why. Programs *never* work the first time, especially machine language programs. A machine language program that takes a wrong turn can go a lot of places and do a lot of bad things in a hurry. To debug it you can either use The Visible Computer, or a machine language monitor such as Micromon 64.

Bill Yee's Micromon–64 is an excellent debugging environment. Compact, complete, and free (if you buy a copy of *Compute's First Book of Commodore 64*). It has commands to disassemble, set breakpoints, compare memory ranges, fill memory with constant values, search for particular combinations of values, single step through programs, and

more. There's a simplified, nonsymbolic assembler built in. If you're writing machine language routines called from Basic, Micromon lets you quickly move between Basic and the debugger. Micromon is $1000 bytes long, and usually runs out of utility RAM at $C000, although it can be relocated elsewhere.

Basic and Machine Language: Sharing the Work

We said way back in Chapter 1 that the smart programmer doesn't use machine language unless he has to. Even then, he first tries to get the job done with a hybrid program, with Basic providing the main framework and machine language for the part that needs speed. To accomplish sharing requires an understanding of what Basic is and isn't.

What is Basic?

Commodore 64 Basic is an 8K 6502 machine language program that normally resides in ROM from $A000 – $BFFF. Basic hogs, and rightfully so, most of the addresses in page zero for its internal variables. If it didn't, Basic programs would run even slower. Pages 310–316 of the *Programmer's Reference Guide* give a detailed breakdown on what locations are used for what.

Basic creates an environment that lets a programmer work with concepts instead of registers and addresses. Where you can say ***X = Y + Z*** without considering where to store variable X. Where you can copy a formula like ***R = SIN (2*THETA)*** almost straight out of a math book, without worrying how sines are calculated. Basic goes a long way toward bridging the enormous gap between the English language and standard algebraic representation, and the 8 bit, 56 instruction world of the 6502.

If you learned one thing in this book, it's that a 6502 can no more understand:

```
1000 INPUT "ENTER YOUR NAME ";A$
```

than it can play chess or calculate income taxes. 6502 *programs* can be written to do these things, but the 6502 just rolls, shifts, jumps, adds and subtracts.

A Basic program is an elaborate data table constructed and maintained by the machine language routines residing in $A000 – $BFFF that we

refer to collectively as Basic. The data table begins at $801 and works up. Actually, it works up from $801 and down from $9FFF: you run out of memory when the two parts meet. When a Basic program is executing, the 6502 is not executing any of this data directly, that is, if we could somehow catch the 6502 in the act of running a Basic program, we would never find program counter values less than $A000.

Instead, the 6502 is busy running subroutines that look at the data table (Basic program) to figure out what to do and where to go next. If Basic is told to do something unreasonable, such as divide by zero or jump to a nonexistent line, he's not in the same situation as the 6502 jumping to non-code areas of memory, or encountering undefined opcodes. Basic remains in control, calmly printing error messages and waiting for further instructions. That's why faulty Basic programs are much less likely to crash the computer than bugged machine language programs.

How to Organize Basic and Machine Language

To call a machine language subroutine from Basic, use the SYS instruction. Give it the decimal version of your routine's starting address. To call a subroutine at $C000, use SYS 49152. When your machine language subroutine executes a final RTS, Basic will pick up execution with the statement immediately after the call. A full 4K of RAM is available from $C000–$CFFF for just this purpose.

Parameter Passing

Just before your subroutine takes control, the A, X, Y, and P registers are loaded from locations $30C–$30F, respectively. If you have previously poked values that your machine language routine needs into these addresses, you can thereby communicate facts from Basic to machine language. Similarly, when your subroutine returns, the registers are stored back in $30C–$30F.

If a few more bytes of information are needed, you can poke them into memory, at the place where the machine language subroutine expects to find data. Heaven knows, peeking and poking is second nature to most C-64 Basic programmers already. If the application requires a lot of data transfer between Basic and machine language, the machine language subroutine can act on the desired Basic variables directly. This is more difficult, as it requires a thorough understanding of how Basic stores variables internally.

The End of the Road

That's it for the tutorial part of the Visible Computer. Obviously, at this point you haven't learned everything there is to know about machine language. As with any discipline, learning machine language involves more than reading one book. Here are three sure-fire ways to improve your programming skills:

Read good books. Several good ones are listed below. There are good ones I've left out, but beware of the Judging-by-the-Cover syndrome in programming books. There are some *bad* ones out there.

Study other people's assembly language programs. The Kernal listing in Abacus Software's *The Anatomy of the Commodore 64* is a rich collection of ways to get things done in machine language. Computer magazines publish all manner of assembly language programs every month, usually with extensive comments and description. Pick one and dig in.

Give yourself projects. Pick a task that seems suited to your capabilities (although sometimes the most innocent projects prove to be bottomless pools of complications). Maybe you could write a bubble sort capable of sorting more than one page of data. Or one that let you pass the size and address of the array when you call it. Try to modify ASCII Organ to use the joystick to control volume, or note duration. Think small and ease your way into more complex stuff.

Suggested Reading

6502 Programming

By Rodnay Zaks. Sybex, Inc. 2344 Sixth Street, Berkeley, California 94710.

A detailed reference guide, with extensive discussions of signed numbers, and demonstration programs implementing various arithmetic and sorting problems.

6502 Assembly Language Programming

by Lance A. Leventhal. Osborne/McGraw Hill. 630 Bancroft Way, Berkeley, California 94701.

Another reference volume; get either the Sybex book or this one, but not both.

Programming a Microcomputer: 6502

By Claxton Foster. Addison-Wesley Publishing Company, Inc. Reading, Massachusetts 01867.

A funny little book, with some of the worst diagrams, but best descriptions you'll find anywhere. You'll need to read between the lines of this book somewhat, as its target vehicle is not the Commodore 64, but the KIM single board computer—but that's half the fun. Highly recommended.

Compute's First Book of Commodore 64

Numerous Authors. Compute Publications, Inc. Greensboro, North Carolina 27403.

Some of the meatiest articles from *Compute* and *Compute's Gazette* reprinted. Several articles on linking Basic and machine language. Micromon-64 instructions (and right to use) included.

Compute's First Book of Commodore 64 Sound and Graphics

Numerous Authors. Compute Publications, Inc. Greensboro, North Carolina 27403.

More *Compute* and *Compute's Gazette* reprints.

Anatomy of the C-64

Angerhausen, et al. Abacus Software, P.O. Box 7211, Grand Rapids, Michigan 49510

A C-64 reference book. Especially noteworthy for its fully commented ROM listings.

The Commodore 64 Music Book

James Vogel and Nevin B. Scrimshaw. Birkhauser Boston, Inc.

130 pages about how to put SID through his paces.

Appendix A
Behind the Scenes of TVC

TVC Memory Map

```
Page Number

   ---------------------------------
   FF
   ::  Kernal
   E0
   ::  I/O
   D0
   ::  Character Generator
   C8
   ::  Screen Memory
   C4
   ::  Primary User Memory
   C0
   ::  Basic Interpreter
   A0
   ::  User Stack
   9F
   ::  User Zero Page
   9E
   ::  TVC ML Routines, Screen Images
   90
   ::  TVC (Basic Program)
   04
   ::  Basic/Kernal Work Areas
   02
   ::  Stack
   01
   ::  Zero Page
   00
   ---------------------------------
```

The Visible Computer: 6502 is a machine language/Basic hybrid. The Basic part does most of the work; it would have been almost as easy to write The Visible Computer: 6502 on the IBM PC, a machine based on the 8088 processor. Machine language routines are used primarily to calculate the result of arithmetic, logical, and shift instructions, as Basic is singularly unsuited to this kind of bitwise manipulation.

To protect itself from the user, TVC maintains separate zero and stack pages. You can verify this by comparing reads of $0000-$01FF and $9E00-$9FFF.

Although TVC allows master users access to the real zero page, there is no way to share the stack. The simulator always uses the bogus stack page at $9F00, and a subroutine passed to the 6502 via the GO command will use the real 6502 stack. Any data written to the stack page during a GO will not appear to be there when you get back to TVC.

TVC reads addresses $FFFE and $FFFF as $00, $C3, during EDIT and Window reads, and after a keyboard interrupt. This facilitates vectoring simulated interrupts to a user handling program.

Locations $00 and $01, the 6510's on-chip I/O registers, cannot be written to, even in master mode.

Disclaimer

The Visible Computer: 6502 is a tool for teaching machine language programming; a secondary function is the debugging of 6502 programs. It is not intended to be a rigorous copy of the 6502's internal workings. Although it correctly executes all 151 defined opcodes, it may arrive at identical results through different mechanisms. The term "microstep" has a conceptual kinship to microcode, but any similarity between the real working microcode of a 6502 and the microsteps of TVC is coincidental.

Appendix B
Screen Reference

TVC Screen Codes

Code	Char	Code	Char	Code	Char	Code	Char
0	= @	42	= *	84	=	126	=
1	= a	43	= +	85	=	127	=
2	= b	44	= ,	86	=	128	=
3	= c	45	= -	87	=	129	=
4	= d	46	= .	88	=	130	=
5	= e	47	= /	89	=	131	=
6	= f	48	= 0	90	=	132	=
7	= g	49	= 1	91	=	133	=
8	= h	50	= 2	92	=	134	=
9	= i	51	= 3	93	=	135	=
10	= j	52	= 4	94	=	136	=
11	= k	53	= 5	95	=	137	=
12	= l	54	= 6	96	=	138	=
13	= m	55	= 7	97	=	139	=
14	= n	56	= 8	98	=	140	=
15	= o	57	= 9	99	=	141	=
16	= p	58	= :	100	=	142	=
17	= q	59	= ;	101	=	143	=
18	= r	60	= <	102	=	144	=
19	= s	61	= =	103	=	145	=
20	= t	62	= >	104	=	146	=
21	= u	63	= ?	105	=	147	=
22	= v	64	=	106	=	148	=
23	= w	65	=	107	=	149	=
24	= x	66	=	108	=	150	=
25	= y	67	=	109	=	151	=
26	= z	68	=	110	=	152	=
27	= [	69	=	111	=	153	=
28	= £	70	=	112	=	154	=
29	=]	71	=	113	=	155	=
30	= ↑	72	=	114	=	156	=
31	= ←	73	=	115	=	157	=
32	=	74	=	116	=	158	=
33	= !	75	=	117	=	159	=
34	= "	76	=	118	=	160	=
35	= #	77	=	119	=	161	=
36	= $	78	=	120	=	162	=
37	= %	79	=	121	=	163	=
38	= &	80	=	122	=	164	=
39	= '	81	=	123	=	165	=
40	= (	82	=	124	=	166	=
41	=)	83	=	125	=	167	=

168 =
169 =
170 =
171 =
172 =
173 =
174 =
175 =
176 =
177 =
178 =
179 =
180 =
181 =
182 =
183 =
184 =
185 =
186 =
187 =
188 =
189 =
190 =
191 =
192 =
193 =
194 =
195 =
196 =
197 =
198 =
199 =
200 =
201 =
202 =
203 =
204 =
205 =
206 =
207 =
208 =
209 =
210 =
211 =
212 =
213 =
214 =
215 =
216 =
217 =
218 =
219 =
220 =
221 =
222 =
223 =
224 =
225 =
226 =
227 =
228 =
229 =
230 =
231 =
232 =
233 =
234 =
235 =
236 =
237 =
238 =
239 =
240 =
241 =
242 =
243 =
244 =
245 =
246 =
247 =
248 =
249 =
250 =
251 =
252 =
253 =
254 =
255 =

Standard Color Codes

Color	Code	Color	Code
Black	0	Orange	8
White	1	Brown	9
Red	2	Light Red	10
Cyan	3	Gray 1	11
Purple	4	Gray 2	12
Green	5	Light Green	13
Blue	6	Light Blue	14
Yellow	7	Gray 3	15

Appendix C
Monitor Commands Reference

Monitor mode is indicated by the "#" (number sign) prompt on the last line of the display. This indicates TVC's readiness to accept one of the following commands.

Monitor commands have the general form:

<command> [argument1] [argument2]

You *must* separate a command and its arguments by one or more spaces. You *must not* use spaces within a command or argument.

This command list uses the following conventions:

address	A 16 bit number valid in the current monitor base.
value	A number valid in the current base.
register	An on-screen register. They are: DL, DB, IR, A, S, P, X, Y, PC, AD, MEMA, and MEMD.
filename	A valid disk file name, without embedded spaces: "TEST-FILE"; "PROGRAM1".

Slashes ("/") separate equivalent command parameters.
Square brackets ("[]") enclose optional parameters.

The Commands

BASE

Change display or monitor base.

Syntax: BASE ***register***/ALL/MEM/MON HEX/BIN/DEC

Controls how numbers will be displayed on the screen, or interpreted when entered in the monitor. In place of ***register*** one may use ALL to change the base of all registers, MEM to change base of memory display, and MON to change monitor input base.

Example: BASE PC BIN

BORDER

Syntax: BORDER ***value***

Change screen border color.

BORDER changes the border color to the color value specified by parameter ***value***, where 0 <= ***value*** <= 15. Follows standard C–64 color code convention (see appendix B).

Example: BORDER 6

CALC

Turn on calculator.

Syntax: CALC

This command invokes a four function, three base, integer calculator. The four functions are: + , -, *, and /. As with monitor commands, the operands and operator must be separated by spaces.

Your first keystroke has a special effect. A Control H, B, or D (for Hex, Binary, Decimal), changes the calculator base and redisplays the number in that base. F1 exits back to the monitor (or to a simulator pause, depending on how you got here). Any other character clears the line and waits for your input.

To use the calculator for base conversion, enter the number you want converted, and use one of the base conversion keystrokes. To convert $3CF into decimal: Set calculator base to hex with Ctrl-H. Enter: 3CF <return>. Type Ctrl-D to see the number in decimal, Ctrl-B for binary.

To multiply $3FF by $10, enter: 3FF * 10.

If an operation produces a value greater than 65,535, or a negative value less than –32,767, a range error is given. Negative values are displayed in two's complement form.

Answers are displayed with the same routines that refresh the registers, and therefore include leading zeros and, with binary numbers, embedded spaces. You need not include leading zeros, and *must not* include spaces within numbers.

Example: CALC

EDIT

Edit memory.

Syntax: EDIT ***address***

Entering EDIT mode displays the EDIT prompt, followed by the selected location and its contents. As with the calculator function, the first keystroke has special significance.

The return and cursor right keys advance to the next address. Cursor left displays the previous address. F1 exits EDIT mode back to the monitor. Any other character enters the standard input routine. When return is pressed, your entry is checked for validity in the current monitor base. If valid and within range, it replaces the value formerly at that address.

If that value is part of the instruction pointed at by the program counter, the next instruction line is updated. See the MASTER command for a discussion of what locations can be written to.

Example: EDIT $C100

ERASE

Erase display.

Syntax: ERASE

Clears display, but does not prevent subsequent processes from writing to it.

Example: ERASE

GO

Transfer program execution to 6502.

Syntax: GO

Master mode only. If the next command is a JSR, execution of that subroutine is passed directly to the 6502.

Assuming the routine does no damage in the process of running, and there is no way TVC can protect itself in this situation, TVC will regain control when the 6502 executes an RTS at the end of the subroutine.

During 6502 execution, "6502 Mode" appears on the error line of the monitor status area. When control is returned, the programmer's registers, the disassembly window, and the next instruction line are updated.

Example: GO

L

Disassemble Memory.

Syntax: [***address***] L

Disassembles 5 instructions beginning at ***address***. If no address is specified, disassembly picks up where it left off previously.

Example: C000 L

LC / RC

Set first address of right or left memory columns.

Syntax: LC/RC ***address***

The effect of this command will not been seen unless the memory window is open.

Example: RC 900

LOAD

Load binary information from disk.

Syntax: LOAD ***filename***

Enters programs and data stored in disk files into memory, beginning at $C000. If a file is larger than the $400 byte work space, it is truncated at that point.

If the memory window is open, LOAD redisplays these addresses, even if they were not affected by the load. The next instruction line is also updated.

Example: LOAD MAGNUMOPUS

LOAD MEMORY

A shortcut to editing ram.

Syntax: ***address value***

If ***address*** is a location that may be written to, ***value*** replaces the current contents of ***address***. See the MASTER command for more information.

Example: A00 FF

LOAD REGISTER

Manually load register with selected value.

Syntax: ***register value***

If one of 16 bit registers is specified, ***value*** can range from 0–65,535. Otherwise, loads greater than 255 produce range errors.

Example: PC 300

MASTER

Enter/exit master mode.

Syntax: MASTER ON/OFF

Master mode is for experienced users of TVC who desire more flexibility in debugging and executing programs. It is indicated by the letters "M" and "Z" on the status line, and has the following effects:

1. Enables the GO command.
2. Allows writing to all memory locations, not just $C000–$C7FF.
3. Maps actual (system) zero page.

Example: MASTER OFF

POP

Pop program counter from stack.

Syntax: POP

Simulates an RTS by loading the program counter with the stack's two topmost bytes and incrementing the stack pointer by two. The value placed in the PC as a result of this instruction will be meaningful only if the top of the stack contains the return address of the calling routine. If S contains $FE or greater, POP is ignored.

Useful as a way of backing out of slow, monotonous routines (such as a delay loop), or in figuring out how you came to be in a section of code.

Example: POP

PRINTER

Turn printer on or off.

Syntax: PRINTER ON/OFF [*value*]

Determines whether or not disassembly will be sent to a printer. Printing occurs only after the simulator's execution of each instruction. If you have selected this option, a "P" will appear on the status line. Parameter ***value*** is an eight bit number sent to the printer as a secondary address parameter.

If a printer is not connected, is off-line, or if you have a nonstandard interface, when you use this function, TVC may lock up, forcing you to reload to regain control. The printer must be configured as device 4.

Example: PRINTER ON 80

RESTART

Restart TVC.

Syntax: RESTART

Restores TVC to its load time default values. Does not affect user memory.

Example: RESTART

RESTORE

Restore display.

Syntax: RESTORE

Undoes the work of the ERASE command by redrawing screen, according to the current window and register base settings.

Example: RESTORE

SAVE

Save binary data to disk.

Syntax: SAVE ***filename*** [***value***]

Saves ***value*** bytes, where 1 <= ***value*** <= $400, beginning with $C000 to the floppy disk in drive 8 under the name ***filename***. If ***value*** is unspecified, $400 bytes are saved.

The standard conditions must be met for this command to succeed. Namely, the drive door closed on an initialized, un-write protected disk with some room on it. Do *not* attempt to defeat the write protec-

tion of the TVC disk. To overwrite a file of the same name, use SAVE @0:***filename***.

Examples: SAVE MAGNUMOPUS C0

SCREEN

Syntax: SCREEN ***value***

Change screen background color.

Changes the screen background color to the color value specified in parameter ***value***, where 0 <= ***value*** <= 15. Follows standard C–64 color code convention (see appendix B).

Example: SCREEN 6

STEP

Set simulator step mode.

Syntax: STEP 0/1/2/3

Sets the stepping rate of the 6502 simulator. The effect of each step value is summarized in appendix D.

Example: STEP 3

WINDOW

Set screen window.

Syntax: WINDOW OPEN/CLOSE/MEM

This command controls what is shown in the "window" area of the display (approximately the central third). There are three options:

CLOSE, the default setting, displays the entire processor/memory combination. OPEN clears memory area. MEM displays 16 memory locations. (See RC and LC functions). The programmer's registers (PC-A-X-Y-P-S) always remain onscreen.

Example: WINDOW MEM

Appendix D
6502 Simulator Reference

The 6502 simulator is the portion of The Visible Computer that runs 6502 machine language. The simulator interactively executes the 151 defined instructions of the 6502 instruction set, by animating the microsteps necessary to perform each. Undefined opcodes are trapped and refused.

The Message Window

If the simulator is active, the first line of the message window will display either "FETCH" (if the fetch cycle is in progress), or the mnemonic and addressing mode of the instruction under execution.

Microsteps

The second line of the message window displays the microstep currently being executed. Microsteps are individual small tasks accomplished in sequence to complete a given instruction. The nine TVC microsteps are:

CALC ADDRS	Use the X or Y register to modify AD.
COMPUTE	Perform an arithmetic, shift, or logical operation.
COND FLAGS	Condition the flags.
DEC	Decrement a register.
INC	Increment a register.
READ	Read into the data latch the contents of the address in AD.
T: (transfer)	Transfer a number from one register to another. The source register is unchanged.
TEST FLAG	Examine P register flag.
WRITE	Write the number in the data latch to memory location AD.

Controlling the Simulator

The simulator is largely controlled by the monitor command STEP. The effect of each of the four step values is outlined here.

Step mode 3: The slowest, most instructive mode. The simulator pauses at each microstep. Pressing the spacebar will cause execution to proceed to the next microstep. When the instruction is complete, the monitor is entered.

Step mode 2: Pauses do not occur automatically at microsteps. A pause may be forced by pressing the space bar. The monitor is entered after the completion of a full instruction.

Step mode 1: Like mode (2) but instead of entering the monitor after completion of an instruction, the next instruction in memory is executed. F1 will force monitor entry after completion of the current instruction.

Step mode 0: Similar to step mode (1) but without update of the display. Only the disassembly and next instruction areas are kept current. As with (1), you can force monitor entry with F1. When you enter the monitor, the programmer's registers are updated to their proper values. Because this mode skips time consuming display routines, it gives the greatest execution speed, approximately .5 instructions per second.

Speed Control

The number keys control execution speed. 1 produces the fastest execution, 9 the slowest. Speed control is ignored in step mode 0.

Pausing the Simulator

You can force the simulator to pause by pressing the spacebar. To resume execution, press the spacebar again. Pressing C activates the calculator, the only monitor function available from within the simulator. Exiting the calculator returns you to the pause state.

Interrupt Request

Typing I while the simulator is active generates an interrupt request. If interrupts are enabled (I flag = 0), the B bit will be cleared, PC and P pushed on the stack, and an interrupt handling program at $C300 entered.

Appendix E
Error Messages

BAD OPCODE — The simulator encountered an undefined instruction.

BASE — TVC is unable to digest a numeric value you have given it. Make sure you use values valid in the selected base, without embedded spaces.

COMMAND — The command interpreter cannot understand your instruction. Try again, and watch your syntax.

DISK FULL — You are trying to SAVE a file to a disk that has no room for it. Use another disk.

DIV BY 0 — The calculator was told to divide by zero.

MISMATCH — Disk mismatch error. Happens when you try to use a Commodore disk not in 1541 format.

NAME — You tried to save a file with the same name as an existing file. To replace original file, use SAVE @0:***filename***.

NOT FOUND — File not found. Check your spelling, and remember, no embedded spaces.

NOT JSR — A GO command has been issued without JSR as the next instruction.

NOT MASTER — You tried to execute GO without being in master mode.

RANGE — You entered a number too large for the situation. For example, trying to load the X register with $101.

READ ERROR — Covers a multitude of sins related to disk operations, including: Drive doors left open, disks inserted upside down or not at all, uninitialized diskettes, and real problems like faulty disk drives.

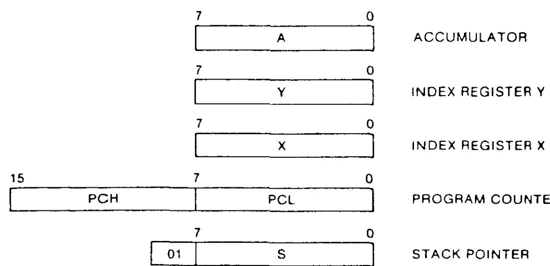
W.PROTECT — You attempted a SAVE on a write protected disk. Note: Do not attempt to defeat the write protec-

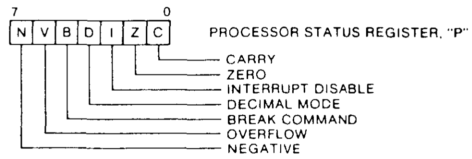
tion of the TVC disk. Use an ordinary initialized diskette to save your files.

ER XXXXX-XX An internal error has occurred in TVC. This error is caused by either a bug in the program or something your activities in master mode have done to damage it. If you feel the first case is likely we'd like to know about it. Drop us a letter listing the exact error message and a description of what you were doing when you got the error. You must reload TVC to recover from an internal error.

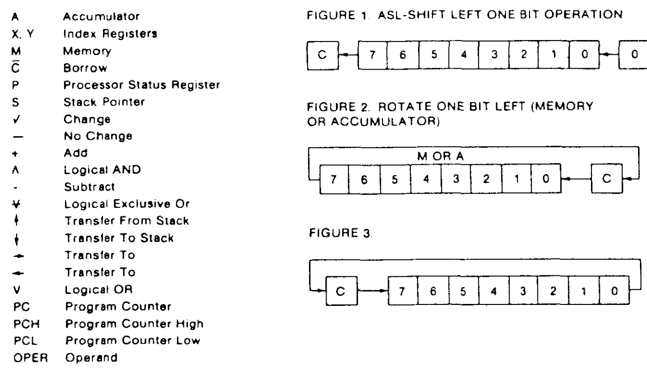
Appendix F
6502 Reference

PROGRAMMING MODEL





THE FOLLOWING NOTATION APPLIES TO THIS SUMMARY:



A	Accumulator
X, Y	Index Registers
M	Memory
$\bar{C}$	Borrow
P	Processor Status Register
S	Stack Pointer
✓	Change
—	No Change
+	Add
∧	Logical AND
-	Subtract
⊻	Logical Exclusive Or
↑	Transfer From Stack
↓	Transfer To Stack
→	Transfer To
←	Transfer To
V	Logical OR
PC	Program Counter
PCH	Program Counter High
PCL	Program Counter Low
OPER	Operand
#	Immediate Addressing Mode

FIGURE 1. ASL-SHIFT LEFT ONE BIT OPERATION

FIGURE 2. ROTATE ONE BIT LEFT (MEMORY OR ACCUMULATOR)

FIGURE 3.

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg. N Z C I D V
ADC						
Add memory to accumulator with carry	A+M+C → A,C	Immediate	ADC #Oper	69	2	√√√– –√
		Zero Page	ADC Oper	65	2	
		Zero Page,X	ADC Oper,X	75	2	
		Absolute	ADC Oper	6D	3	
		Absolute,X	ADC Oper,X	7D	3	
		Absolute,Y	ADC Oper,Y	79	3	
		(Indirect,X)	ADC (Oper,X)	61	2	
		(Indirect),Y	ADC (Oper),Y	71	2	
AND						
"AND" memory with accumulator	A ∧ M → A	Immediate	AND #Oper	29	2	√√– – – –
		Zero Page	AND Oper	25	2	
		Zero Page,X	AND Oper,X	35	2	
		Absolute	AND Oper	2D	3	
		Absolute,X	AND Oper,X	3D	3	
		Absolute,Y	AND Oper,Y	39	3	
		(Indirect,X)	AND (Oper,X)	21	2	
		(Indirect),Y	AND (Oper),Y	31	2	
ASL						
Shift left one bit (Memory or Accumulator)	(See Figure 1)	Accumulator	ASL A	0A	1	√√√– – –
		Zero Page	ASL Oper	06	2	
		Zero Page,X	ASL Oper,X	16	2	
		Absolute	ASL Oper	0E	3	
		Absolute,X	ASL Oper,X	1E	3	
BCC						
Branch on carry clear	Branch on C=0	Relative	BCC Oper	90	2	– – – – – –
BCS						
Branch on carry set	Branch on C=1	Relative	BCS Oper	B0	2	– – – – – –
BEQ						
Branch on result zero	Branch on Z=1	Relative	BEQ Oper	F0	2	– – – – – –
BIT						
Test bits in memory with accumulator	A ∧ M, M_7 → N, M_6 → V	Zero Page	BIT* Oper	24	2	M_7√– – – M_6
		Absolute	BIT* Oper	2C	3	
BMI						
Branch on result minus	Branch on N=1	Relative	BMI Oper	30	2	– – – – – –
BNE						
Branch on result not zero	Branch on Z=0	Relative	BNE Oper	D0	2	– – – – – –
BPL						
Branch on result plus	Branch on N=0	Relative	BPL oper	10	2	– – – – – –
BRK						
Force Break	Forced Interrupt PC+2 ↓ P ↓	Implied	BRK*	00	1	– – – 1 – –
BVC						
Branch on overflow clear	Branch on V=0	Relative	BVC Oper	50	2	– – – – – –

* Bits 6 and 7 are transferred to the status register. If the result of A AND M is 0 then Z=1, otherwise Z=0.

* A BRK cannot be masked by setting I.

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg. N Z C I D V
BVS						
Branch on overflow set	Branch on V=1	Relative	BVS Oper	70	2	------
CLC						
Clear carry flag	0 → C	Implied	CLC	18	1	---0--
CLD						
Clear decimal mode	0 → D	Implied	CLD	D8	1	-0----
CLI						
	0 → I	Implied	CLI	58	1	---0--
CLV						
Clear overflow flag	0 → V	Implied	CLV	B8	1	0-----
CMP						
Compare memory and accumulator	A − M	Immediate	CMP #Oper	C9	2	√√√---
		Zero Page	CMP Oper	C5	2	
		Zero Page, X	CMP Oper,X	D5	2	
		Absolute	CMP Oper	CD	3	
		Absolute,X	CMP Oper,X	DD	3	
		Absolute,Y	CMP Oper,Y	D9	3	
		(Indirect,X)	CMP (Oper,X)	C1	2	
		(Indirect),Y	CMP (Oper),Y	D1	2	
CPX						
Compare memory and index X	X − M	Immediate	CPX #Oper	E0	2	√√√---
		Zero Page	CPX Oper	E4	2	
		Absolute	CPX Oper	EC	3	
CPY						
Compare memory and index Y	Y − M	Immediate	CPY #Oper	C0	2	√√√---
		Zero Page	CPY Oper	C4	2	
		Absolute	CPY Oper	CC	3	
DEC						
Decrement memory by one	M − 1 → M	Zero Page	DEC Oper	C6	2	√√----
		Zero Page,X	DEC Oper,X	D6	2	
		Absolute	DEC Oper	CE	3	
		Absolute,X	DEC Oper,X	DE	3	
DEX						
Decrement index X by one	X − 1 → X	Implied	DEX	CA	1	√√----
DEY						
Decrement index Y by one	Y − 1 → Y	Implied	DEY	88	1	√√----

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg. N Z C I D V
EOR						
"Exclusive-Or" memory with accumulator	A ⊻ M → A	Immediate	EOR #Oper	49	2	√√----
		Zero Page	EOR Oper	45	2	
		Zero Page,X	EOR Oper,X	55	2	
		Absolute	EOR Oper	4D	3	
		Absolute,X	EOR Oper,X	5D	3	
		Absolute,Y	EOR Oper,Y	59	3	
		(Indirect,X)	EOR (Oper,X)	41	2	
		(Indirect),Y	EOR (Oper),Y	51	2	
INC						
Increment memory by one	M + 1 → M	Zero Page	INC Oper	E6	2	√√----
		Zero Page,X	INC Oper,X	F6	2	
		Absolute	INC Oper	EE	3	
		Absolute,X	INC Oper,X	FE	3	
INX						
Increment index X by one	X + 1 → X	Implied	INX	E8	1	√√----
INY						
Increment index Y by one	Y + 1 → Y	Implied	INY	C8	1	√√----
JMP						
Jump to new location	(PC+1) → PCL	Absolute	JMP Oper	4C	3	------
	(PC+2) → PCH	Indirect	JMP (Oper)	6C	3	
JSR						
Jump to new location saving return address	PC+2 ↓ (PC+1) → PCL (PC+2) → PCH	Absolute	JSR Oper	20	3	------
LDA						
Load accumulator with memory	M → A	Immediate	LDA #Oper	A9	2	√√----
		Zero Page	LDA Oper	A5	2	
		Zero Page,X	LDA Oper,X	B5	2	
		Absolute	LDA Oper	AD	3	
		Absolute,X	LDA Oper,X	BD	3	
		Absolute,Y	LDA Oper,Y	B9	3	
		(Indirect,X)	LDA (Oper,X)	A1	2	
		(Indirect),Y	LDA (Oper),Y	B1	2	
LDX						
Load index X with memory	M → X	Immediate	LDX #Oper	A2	2	√√----
		Zero Page	LDX Oper	A6	2	
		Zero Page,Y	LDX Oper,Y	B6	2	
		Absolute	LDX Oper	AE	3	
		Absolute,Y	LDX Oper,Y	BE	3	
LDY						
Load index Y with memory	M → Y	Immediate	LDY #Oper	A0	2	√√----
		Zero Page	LDY Oper	A4	2	
		Zero Page,X	LDY Oper,X	B4	2	
		Absolute	LDY Oper	AC	3	
		Absolute,X	LDY Oper,X	BC	3	

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg. N Z C I D V
LSR						
Shift right one bit (memory or accumulator)	(See Figure 1)	Accumulator	LSR A	4A	1	0√√–––
		Zero Page	LSR Oper	46	2	
		Zero Page,X	LSR Oper,X	56	2	
		Absolute	LSR Oper	4E	3	
		Absolute,X	LSR Oper,X	5E	3	
NOP						
No operation	No Operation	Implied	NOP	EA	1	––––––
ORA						
"OR" memory with accumulator	A V M → A	Immediate	ORA #Oper	09	2	√√––––
		Zero Page	ORA Oper	05	2	
		Zero Page,X	ORA Oper,X	15	2	
		Absolute	ORA Oper	0D	3	
		Absolute,X	ORA Oper,X	1D	3	
		Absolute,Y	ORA Oper,Y	19	3	
		(Indirect,X)	ORA (Oper,X)	01	2	
		(Indirect),Y	ORA (Oper),Y	11	2	
PHA						
Push accumulator on stack	A ↓	Implied	PHA	48	1	––––––
PHP						
Push processor status on stack	P ↓	Implied	PHP	08	1	––––––
PLA						
Pull accumulator from stack	A ↑	Implied	PLA	68	1	√√––––
PLP						
Pull processor status from stack	P ↑	Implied	PLP	28	1	From Stack
ROL						
Rotate one bit left (memory or accumulator)	(See Figure 2)	Accumulator	ROL A	2A	1	√√√–––
		Zero Page	ROL Oper	26	2	
		Zero Page,X	ROL Oper,X	36	2	
		Absolute	ROL Oper	2E	3	
		Absolute,X	ROL Oper,X	3E	3	
ROR						
Rotate one bit right (memory or accumulator)	(See Figure 3)	Accumulator	ROR A	6A	1	√√√–––
		Zero Page	ROR Oper	66	2	
		Zero Page,X	ROR Oper,X	76	2	
		Absolute	ROR Oper	6E	3	
		Absolute,X	ROR Oper,X	7E	3	

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg. N Z C I D V
RTI Return from interrupt	P↑ PC↑	Implied	RTI	40	1	From Stack
RTS Return from subroutine	PC↑, PC+1 → PC	Implied	RTS		1	------
SBC Subtract memory from accumulator with borrow	A - M - C̄ → A	Immediate	SBC #Oper	E9	2	√√√--√
		Zero Page	SBC Oper	E5	2	
		Zero Page,X	SBC Oper,X	F5	2	
		Absolute	SBC Oper	ED	3	
		Absolute,X	SBC Oper,X	FD	3	
		Absolute,Y	SBC Oper,Y	F9	3	
		(Indirect,X)	SBC (Oper,X)	E1	2	
		(Indirect),Y	SBC (Oper),Y	F1	2	
SEC Set carry flag	1 → C	Implied	SEC	38	1	--1---
SED Set decimal mode	1 → D	Implied	SED	F8	1	----1-
SEI Set interrupt disable status	1 → I	Implied	SEI	78	1	---1--
STA Store accumulator in memory	A → M	Zero Page	STA Oper	85	2	------
		Zero Page,X	STA Oper,X	95	2	
		Absolute	STA Oper	8D	3	
		Absolute,X	STA Oper,X	9D	3	
		Absolute,Y	STA Oper,Y	99	3	
		(Indirect,X)	STA (Oper,X)	81	2	
		(Indirect),Y	STA (Oper),Y	91	2	
STX Store index X in memory	X → M	Zero Page	STX Oper	86	2	------
		Zero Page,Y	STX Oper,Y	96	2	
		Absolute	STX Oper	8E	3	
STY Store index Y in memory	Y → M	Zero Page	STY Oper	84	2	------
		Zero Page,X	STY Oper,X	94	2	
		Absolute	STY Oper	8C	3	
TAX Transfer accumulator to index X	A → X	Implied	TAX	AA	1	√√----
TAY Transfer accumulator to index Y	A → Y	Implied	TAY	A8	1	√√----
TSX Transfer stack pointer to index X	S → X	Implied	TSX	BA	1	√√----
TXA Transfer index X to accumulator	X → A	Implied	TXA	8A	1	√√----
TXS Transfer index X to stack pointer	X → S	Implied	TXS	9A	1	------
TYA Transfer index Y to accumulator	Y → A	Implied	TYA	98	1	√√----

This material reprinted from the *Apple II Reference Manual* through the courtesy of Apple Computer Inc.

Software Masters™
3330 Hillcroft/Suite BB
Houston, Texas 77057