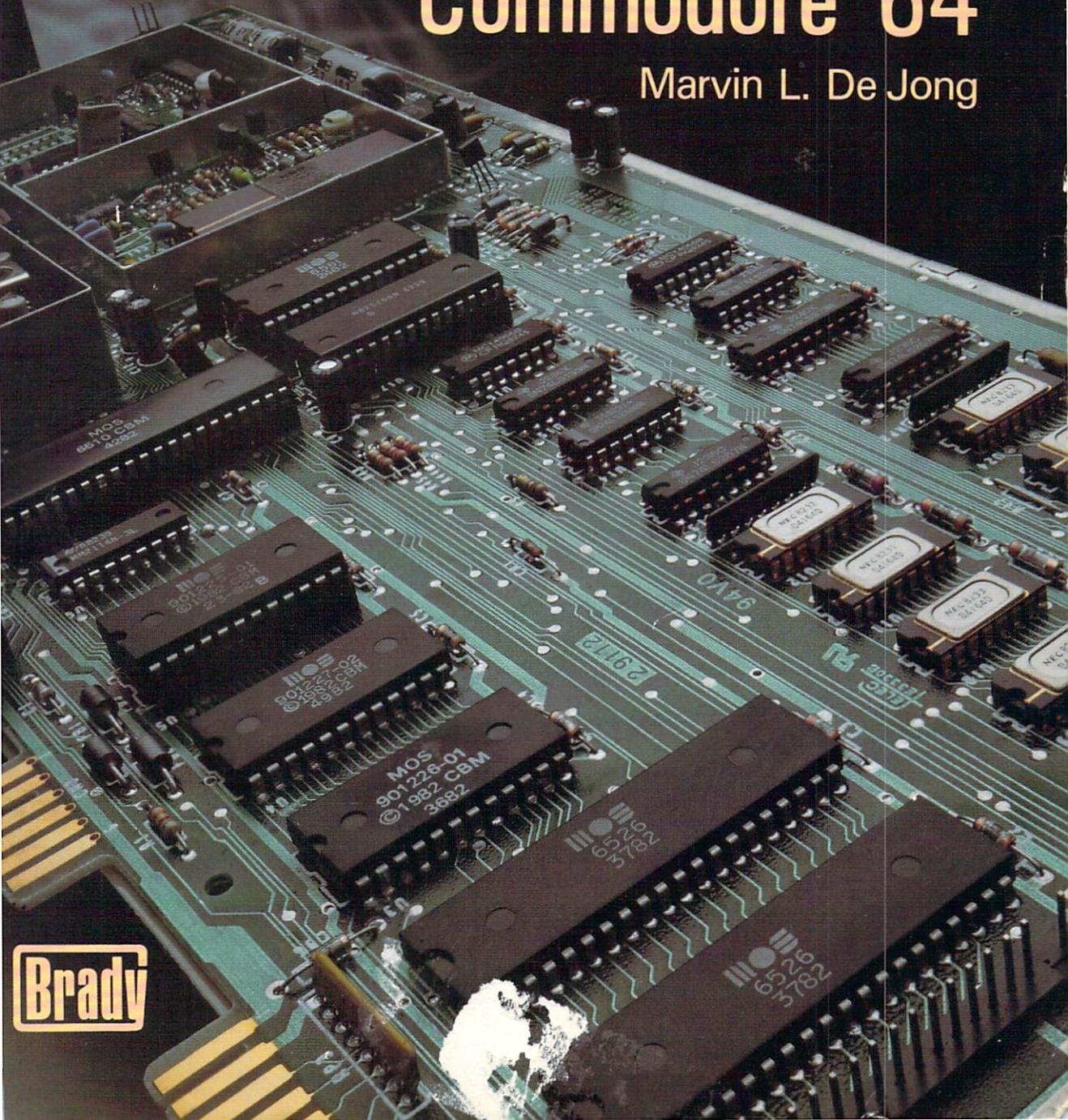


ASSEMBLY LANGUAGE PROGRAMMING with the Commodore 64

Marvin L. De Jong



Brady

Assembly Language Programming with the Commodore 64

Publishing Director: David Culverwell
Acquisitions Editor: Terrell Anderson
Production Editor/Text Design: Michael J. Rogers
Art Director/Cover Design: Don Sellers
Assistant Art Director: Bernard Vervin
Manufacturing Director: John A. Komsa

Copy Editor: Keith R. Tidman
Typesetter: Pagecrafters, Inc., Oswego, NY
Printer: Fairfield Graphics, Fairfield, PA
Typefaces: Helvetica (display), Palatino (text), and
 Typewriter (computer-related material)

Assembly Language Programming with the Commodore 64

Marvin L. De Jong

*The School of the Ozarks
Pt. Lookout, MO*

Brady Communications Company, Inc.

*A Prentice-Hall Publishing Company
Bowie, MD 20715*

Assembly Language Programming with the Commodore 64

Copyright ©1984 by Brady Communications Company, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage and retrieval system, without permission in writing from the publisher. For information, address Brady Communications Company, Inc., Bowie, Maryland 20715.

Library of Congress Cataloging in Publication Data

De Jong, Marvin L.

Assembly language programming with the Commodore 64.

Bibliography: p.

Includes index.

1. Commodore 64 (Computer)—Programming. 2. Assembler language (Computer program language) I. Title.

QA76.8.C64D4 1984 001.64'2 84-6347

ISBN 0-89303-319-7

Prentice-Hall International, Inc., London

Prentice-Hall Canada, Inc., Scarborough, Ontario

Prentice-Hall of Australia, Pty. Ltd., Sydney

Prentice-Hall of India Private Limited, New Delhi

Prentice-Hall of Japan, Inc., Tokyo

Prentice-Hall of Southeast Asia Pte. Ltd., Singapore

Whitehall Books, Limited, Petone, New Zealand

Editora Prentice-Hall Do Brasil LTDA., Rio de Janeiro

Printed in the United States of America

85 86 87 88 89 90 91 92 93 94

2 3 4 5 6 7 8 9 10

Contents

Preface / ix

1 The Commodore 64 Microcomputer System / 1

- I. Introduction / 1
- II. Components of the Commodore 64 / 2
- III. Microcomputer System Buses / 6
- IV. The Microcomputer in Action / 9
- V. An Overview of the 6510 Registers / 18
- VI. Additional Reading / 21
- VII. Summary / 21

2 Writing, Assembling, and Executing Programs / 23

- I. Introduction / 23
- II. Microcomputer Instructions / 23
- III. Addressing Modes / 25
- IV. The Components and Form of an Assembly Language Program / 31
- V. Loading and Executing Simple Programs / 34
- VI. Summary / 39
- VII. Exercises / 40

3 Data Transfer Instructions—Using an Editor/Assembler / 41

- I. Introduction / 41
- II. Data Transfer Instructions: Load and Store / 43
- III. Data Transfer Instructions: Register Transfers / 46
- IV. Writing and Assembling Programs / 48*

 - A. Using the editor / 49
 - B. Using the assembler / 51
 - C. Executing the machine-language program / 52
 - D. Debugging programs / 53

- V. Summary / 53
- VI. Exercises / 54

4 Arithmetic Operations in Assembly Language / 57

- I. Introduction / 57
- II. The Processor Status Register and the Carry Flag / 58
- III. Flag Modification Instructions / 59
- IV. The ADC Instruction / 60
- V. Multiple-Byte Addition / 63
- VI. The SBC Instruction / 64
- VII. Multiple-Byte Subtraction / 66
- VIII. Decimal-Mode Arithmetic / 67*
- IX. Signed-Number Arithmetic / 69*
- X. The JSR and RTS Instructions / 71
- XI. Summary / 72
- XII. Exercises / 73

*The beginning programmer may find that the sections marked with an asterisk are more difficult. Skip these sections if that is the case.

5 Logic Operations in Assembly Language / 77	
I. Introduction / 77	
II. The AND, ORA, and EOR Instructions / 78	
III. Simple Programs to Illustrate the Logic Instructions / 80	
IV. Using the Logic Instructions / 81	
V. Summary / 86	
VI. Exercises / 87	
6 Branches and Loops / 91	
I. Introduction / 91	
II. Reviewing the Flags / 93	
III. The Branch Instructions / 94	
IV. Loop Counters / 99	
V. Test Instructions / 101	
VI. Programming Examples for Branches and Loops / 103	
VII. Summary / 114	
VIII. Exercises / 115	
7 Shift and Rotate Routines / 117	
I. Introduction / 117	
II. Definitions of the Shift and Rotate Instructions / 117	
III. Programs that Use the Shift and Rotate Instructions / 119	
A. Code shift / 119	
B. Displaying the code in a memory location / 120	
C. Printing a byte on the screen / 121	
D. Getting a byte from the keyboard / 122	
E. Simple arithmetic with shifts / 124	
F. Watching the clock / 125	
IV. More Advanced Arithmetic Operations / 126*	
V. Summary / 131	
VI. Exercises / 131	
8 Indexed Addressing Modes / 133	
I. Introduction / 133	
II. The Absolute Indexed Addressing Modes / 134	
III. The Zero-Page Indexed Addressing Mode / 138	
IV. The Indirect Indexed Addressing Mode / 140	
V. The Indexed Indirect Addressing Mode / 146	
VI. Additional Programming Examples / 146	
VII. Summary / 150	
VIII. Exercises / 150	
9 Jumps, Subroutine Calls, and Interrupts / 153	
I. Introduction / 153	
II. The JMP Instruction / 155	
III. Subroutine Calls and Returns / 159	
IV. Operation of the Stack / 160	
V. Stack Operation Instructions / 162	
VI. IRQ-Type Interrupts / 166*	
VII. The Commodore 64 Interrupt Structure / 169*	

VIII. Using Interrupts to Read the Keyboard / 171*

IX. NMI-Type Interrupts / 175*

X. Summary / 176

XI. Exercises / 176

10 Programming the 6581 Sound Interface Device / 179

- I. Introduction / 179
- II. A Brief Introduction to Music Synthesis / 179
- III. The Registers of the 6581 SID Chip / 184
- IV. Setting the Voice Frequencies / 190
- V. Note Duration and Volume: Gating the SID / 194
- VI. The Program to Play a Song / 196
- VII. A BASIC Music Interpreter / 199
- VIII. Summary / 200
- IX. Exercises / 200

11 Applications Using the 6567 Video Interface Chip / 203

- I. Introduction / 203
- II. 6567 VIC Programming Fundamentals / 203
 - A. Bank switching the VIC / 203
 - B. Screen memory / 204
 - C. Color memory / 206
 - D. Character memory / 207
 - E. Character ROM / 208
 - F. A demonstration program / 208
- III. Bit-Mapped Graphics / 210
 - A. Fundamentals of the bit-mapped graphics mode / 210
 - B. Preparing for bit-mapped graphics / 210
 - C. Calculating the location of a pixel's bit / 212
 - D. Switching out the ROM / 219
 - E. Plotting points in the bit-mapped mode / 220
 - F. A bit-mapped graphics demonstration routine / 221
- IV. Summary / 223
- V. Exercises / 224

12 Input/Output: The 6526 Complex Interface Adapter / 225

- I. Introduction / 225
- II. Input/Output Fundamentals / 226
 - A. Input ports / 226
 - B. Output ports / 227
 - C. Memory-mapped I/O and interfacing / 227
- III. Input/Output with the 6526 CIA / 228
 - A. Data-direction registers and I/O ports / 228
 - B. A joystick as an input device / 230
 - C. A Commodore 64 control port as an output port / 234
 - D. More input/output circuits / 238
- IV. Counting and Timing with the 6526 CIA / 242
 - A. Counting/timing fundamentals / 242
 - B. Counting/timing programming examples / 246
 - C. A counting/timing application / 251

V. Summary / 256
VI. Exercises / 256

Appendix A Decimal, Binary, and Hexadecimal Number Systems / 259

- I. Introduction / 259
- II. Decimal Numbers / 261
- III. Binary Numbers / 262
- IV. Hexadecimal Numbers / 265
- V. Problems / 268

Appendix B The Computer Assisted Instruction Program / 269

Appendix C The 6510 Instruction Set Summary / 273

Glossary / 279

Index / 287

**Documentation for Diskette to Accompany Assembly Language
Programming with the Commodore 64 / 293**

Preface

Welcome to life in the computer fast lane—programming with assembly language. Programs written with assembly language have the execution speed required to create lively graphics and sounds in games and educational software. Precision measurement, control, and communications applications also require the speed provided by assembly language programming.

In writing this book, we assumed that you have a beginning knowledge of the BASIC programming language. You do not have to be an expert, but it will help you to learn assembly language if you can write programs in either BASIC or some other high-level language. Most programmers do not begin their programming career by learning assembly language: they begin with a high-level language such as BASIC, FORTRAN, or Pascal. Once you learn assembly language, you will have the best of both worlds: the speed provided by assembly language and the computational ease of BASIC.

We have marked some sections and the contents of this book with an asterisk to indicate that these sections contain more difficult material that may be omitted if you are a beginning programmer. Later, when you have acquired greater programming experience, you may wish to return to these sections to learn these additional concepts.

How do you learn assembly language? Not simply by reading a book about it. It seems there are several important factors. First, you must become acquainted with the instructions available in assembly language, just as you had to learn the BASIC commands. This implies more than simply memorizing the instruction set of the 6510 microprocessor inside the Commodore 64. It means that you can predict the outcome when instructions are executed. You must eventually be able to think like a microprocessor.

Next, you must study and understand simple programs. It is always educational to attempt to understand how another person's programs work. You will be given many examples in this book so that you can see how each instruction is used. In this context, it is important to have a microcomputer with which to execute the programs. It is probably impossible to learn assembly language in the abstract, that is, without a microcomputer, just as it is impossible to learn to play the piano without actually having a piano on which to practice. We have tried to make the sample programs relevant. That is, you should learn something about your computer as you learn about assembly language program-

ming. Almost all of the programs produce an easily observable result so that you can see immediately if the program is properly executing.

Finally, *you* must write, debug, and execute programs. We have provided programming exercises and problems to give you this practice, and they are given without answers. Programming problems in the real world also come without answers, and the most important activity you can undertake in learning to write programs is to write your own. If you will pardon the metaphor, sooner or later you have to remove the training wheels. Writing, debugging, and executing assembly language programs provides challenges, frustrations, and rewards just like any other human activity. Expect to make a few mistakes along the way, and expect that your success will be roughly equivalent to your effort. Always avoid writing grandiose programs: at the beginning of the book, be content with writing programs having several lines, moving to programs with 10 to 20 lines near the end of the book. Our examples and exercises should suggest many possible applications. It is up to you to try writing these programs until at last you become an excellent programmer.

Good luck to you.

Marvin L. De Jong
Department of Mathematics-Physics
The School of the Ozarks
February, 1984

Limits of Liability and Disclaimer of Warranty

The author and publisher of this book have used their best efforts in preparing this book and programs contained in it. These efforts include the development, research, and testing of the programs to determine their effectiveness. The author and the publisher make no warranty of any kind, expressed or implied, with regard to these programs, the text, or the documentation contained in this book. The author and the publisher shall not be liable in any event for claims of incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of the text or the programs. The programs contained in this book are intended for the use of the original purchaser.

Note to Authors

Do you have a manuscript or software program related to personal computers? Do you have an idea for developing such a project? If so, we would like to hear from you. The Brady Company produces a complete range of books and applications software for the personal computer market. We invite you to write to David Culverwell, Publishing Director, Brady Communications Company, Inc., Bowie, MD 20715.

Trademarks of Material Mentioned in This Text

Radio Shack is a trademark of Tandy Corporation.

Commodore 64 is a trademark of Commodore Business Machines.

Dedication

**This book is dedicated to the memory of Bones,
Fang, Yertaw Wogalbukog, and the Mouse Dog.**

The Commodore 64 Microcomputer System

I. Introduction

Human language, whether written or spoken, is a mechanism we use to communicate our thoughts to other people. On occasion, these thoughts may be a set of instructions for another person to follow in order to accomplish a specific task. In that sense, human language is very much like a computer programming language. A *computer programming language* is a mechanism we use to communicate a set of instructions to the computer to make it accomplish a task. With few exceptions, programming languages are written rather than spoken. You are already somewhat familiar with the programming language known as BASIC. It is likely that you began your programming experience on the Commodore 64 by learning the BASIC programming language. Now you are about to learn a new language called *assembly language*. We will not attempt a one-sentence definition of assembly language at this point. That would be as useful as defining French as that language used by French people. For the moment, think of assembly language as another way of communicating with the computer. This entire book is a definition of assembly language.

It may be worthwhile to point out that many computer programmers feel that learning to use assembly language is more difficult than learning BASIC. A programming language that "approximates" human language is frequently called a *high-level language*. A high-level language makes use of familiar words like IF...THEN..., GOTO, and PRINT, which most people can understand. BASIC is a high-level language. On the other hand, in assembly language you will encounter more foreign terms like LDA, ROL, and BIT. Assembly language is much closer to the "language" that the components of the microcomputer use to communicate with each other in order to perform a task. Since we tend to regard computers as much lower than ourselves, we call assembly language a *low-level language*.

Even if you have almost no knowledge of what is happening inside the microcomputer, it is quite easy to write computer programs using a high-level language such as BASIC. Indeed, that is one of the motivations for using BASIC; you do not have to think in great detail about what is taking place inside the microcomputer. On the other hand, to write assembly language programs you must at least know something about the "brain" of the Commodore 64, its 6510 microprocessor. It is also extremely useful to have an elementary understanding of the structure of the microcomputer and how it works. The purpose of this chapter is to provide you with the information about the microcomputer that will be important to you as you learn assembly language. If you are already mildly familiar with the 6510 microprocessor and you already have an elementary understanding of how the Commodore 64 works, then you can proceed directly to the next chapter; otherwise, it will be profitable to spend some time studying this first chapter. Feel free, however, to experiment with several starting points in the book, especially with any of the first three chapters. Indeed, you may wish to skim these three chapters and then start at what, in your judgment, is the appropriate point.

All BASIC programs are written using familiar base-ten (decimal) numbers. On the other hand, experienced assembly language programmers think and work with *hexadecimal numbers* and *binary numbers*. Hexadecimal numbers have a base of sixteen while binary numbers have a base of two. Appendix A of this book explains these two number systems in more detail. Please refer to this appendix if you have had no experience with hexadecimal or binary numbers.

II. Components of the Commodore 64

A microcomputer is a system of components including:

- A **microprocessor**
- ROM (**read only memory**)
- R/W (**read/write**) memory
- I/O (**input/output**) devices

These components are interconnected by three sets of electrical conductors known as *buses*. We will briefly describe the components, then the buses. A sketch of the microcomputer system is shown in Figure 1-1.

The words "microcomputer" and "microprocessor" are often confused. Your Commodore 64 is a microcomputer consisting of the components mentioned above. On the other hand, the microprocessor is just one component of the microcomputer. The **6510 microprocessor** is a 40-pin integrated circuit that is responsible for controlling everything that happens in the microcomputer. Photographs of the 6510 are shown in Figure 1-2. The 6510 microprocessor moves information (data) about in memory, it performs all

the arithmetic operations that take place, and it makes simple decisions based on certain events (for example, depending on which keys are pressed on the keyboard).

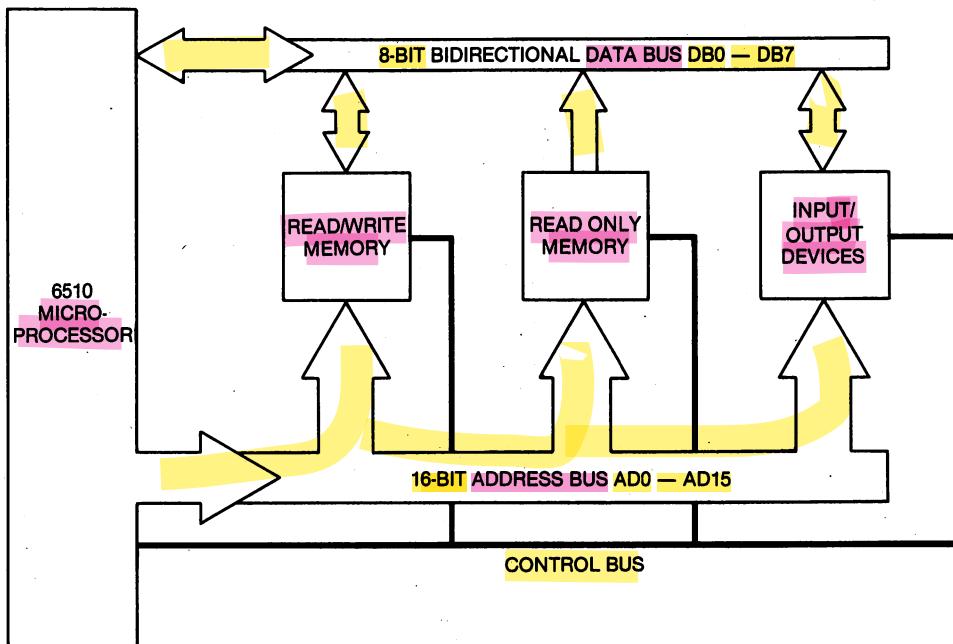


Figure 1-1. Simplified block diagram of the Commodore 64 microcomputer system.

Of course, the microprocessor has no will of its own, nor does it think for itself. It operates under the control of *programs* written by human beings. This book will teach you how to control the microprocessor by writing assembly language programs. These programs must be translated into information the microprocessor can understand, and then they must be stored in the *memory* of the microcomputer system. The 6510 microprocessor is capable of identifying 65,536 different memory locations. Each memory location stores eight bits (binary digits) of information. A binary digit has two possible values, one or zero. The eight bits are designated

D7, D6, D5, D4, D3, D2, D1, D0

This means that bit zero, or D0, is the least-significant bit, while bit seven, D7, is the most-significant bit. Just as in the case of decimal numbers where the least-significant digit is the rightmost digit and the most-significant digit is the leftmost digit, a binary number such as

10010100

has its least-significant bit on the right and its most-significant bit on the left. (Computer scientists, unlike other of earth's creatures, always start counting with the number zero rather than the number one. Thus, the eighth bit in a memory location is called "bit seven.")

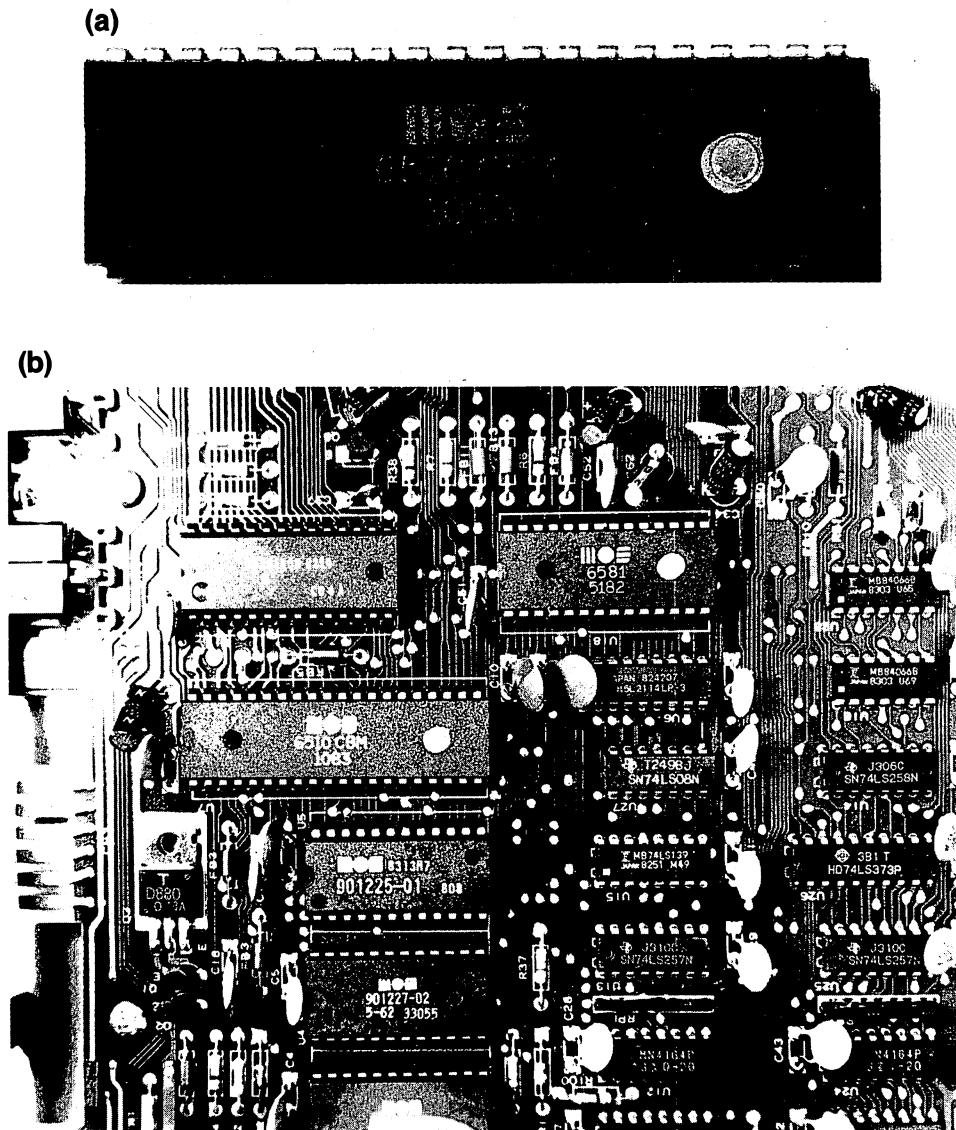


Figure 1-2. Photograph of the Commodore 6510 microprocessor. (a) The 6510 microprocessor. (b) The 6510 microprocessor inside the Commodore 64.

The eight bits of information in a memory location collectively form an *eight-bit code* or an *eight-bit number*. The eight-bit code is frequently called a *byte* of information (or data) or, more simply still, a *byte*.

There are two types of memory, ROM (read only memory) and R/W (read/write) memory. The latter form of memory is frequently called RAM (random access memory). As the name implies, the microprocessor cannot modify the information stored in ROM, it can only copy information from this form of memory into the microprocessor. This process is known as a *read* operation.

* ROM

Two important and lengthy programs have already been written and stored in the ROM inside your computer. These programs are the *operating system* and the *BASIC interpreter*. Both the operating system and the BASIC interpreter are written in the native language of the 6510 microprocessor, namely *6510 machine language*. We will shortly describe machine language in more detail. For the time being, simply be aware that it consists of eight-bit codes stored in memory. The 6510 microprocessor is designed and constructed so that when power is first applied, the microprocessor will begin executing a machine-language program stored at a particular location in memory. This location corresponds to the beginning of the operating system. Among many other things, the operating system allows you to enter information on the keyboard, and display and edit information on the screen.

OS

BASIC
interpreter

The BASIC interpreter is a machine language program that translates the BASIC commands that you give the computer into its native language. The microprocessor begins to execute programs stored in ROM as soon as you turn on the computer, and the Commodore 64 system moves quickly to the BASIC interpreter when power is first applied, giving you the READY prompt.

Now consider *read/write (R/W) memory*. The 6510 cannot only read information in R/W memory but it can also transfer information from itself to R/W memory. This process is known as a *write* operation. This form of memory is called "R/W memory" because with it both read and write operations are possible. While information in R/W memory is *volatile*, meaning that it disappears when power is removed, the information in ROM is *nonvolatile*. Removing the power to the microcomputer does not affect the information stored in ROM.

* RAM

Information from the world outside of the computer is transferred to it by means of *input devices*. The keyboard of the Commodore 64 is the most obvious input device. It is not connected directly to the microprocessor; instead, a special integrated circuit, known as a 6526 CIA (complex interface adapter), is used to connect the 6510 to the keyboard. A device that is connected between the microprocessor and another component of the microcomputer system is known as an *interfacing* device or *interfacing adapter*. The 6526 VIA is very much like R/W memory in the sense that information can be transferred back and forth between it and the microprocessor. Other input devices include cassette tape players, disk drives, and modems. Again, these devices are not usually connected directly to the microprocessor. It is simpler to interface these input devices to the microprocessor either with additional electronic circuitry or with a 6526 CIA, or both.

* input devices

keyboard
=> 6526 CIA

Information is transferred from the computer to the world outside it by means of *output devices*. The most obvious output device is the television set or *video monitor* that you have connected to the Commodore 64. Again, greater simplicity and versatility are achieved when a special integrated circuit, known as the 6567 VIC (video interface chip), is used to interface the 6510 to the video display. Another integrated circuit, the 6581 SID (sound interface device), is used to provide sound output. Information is transferred back

* output devices

monitor
=> 6567 VICsound
=> 6581 SID

and forth between the 6510 microprocessor and both the 6567 VIC and the 6581 SID in exactly the same way as information is transferred to and from R/W memory. Circuitry on the the 6567 VIC and the 6581 SID transforms this information into video and sound signals, respectively.

A diagram that shows the addresses of the locations occupied by ROM, R/W memory, and the I/O devices is called a *memory map* of the micro-computer system. A memory map of the Commodore 64 system is shown in Figure 1-3. To show some of the important features of the memory map, Figure 1-3 is not drawn to scale. Because the Commodore 64 micro-computer system switches blocks or *banks* of memory in and out of active use, a technique called *memory management*, it has several possible memory maps. For simplicity, in Figure 1-3 we show only the most common memory map that we will encounter. Notice that the addresses are expressed with hexadecimal numbers. Also note that hexadecimal numbers are prefaced with a “\$” symbol.

III. Microcomputer System Buses

We turn next to an explanation of how the components of a microcomputer system are interconnected. Refer once again to Figure 1-1. Consider, first, a read operation in which a byte (eight bits of information) is copied from a memory location (or I/O device) into the microprocessor. The 6510 must first identify the memory location that it wishes to read. Like your house, a memory location is identified by a number called its *address*. The microprocessor signals the memory location it wishes to read by placing its address on the microcomputer's *address bus*. The address bus is a set of 16 electrical conductors that connect the microprocessor to R/W memory, ROM, and the I/O interfacing circuits.

The microprocessor works in the binary number system; that is, it places either a one or a zero on each of the 16 conductors to produce a 16-bit binary number. Clearly, a number cannot be sent along a wire conductor. However, there is common agreement among the components of a microcomputer—agreement built into it by its designers—that a voltage near five volts corresponds to a binary one, while a voltage near zero volts corresponds to a binary zero. If an address line is to correspond to a “one,” then the microprocessor makes the voltage on the address line five volts. If an address line is to correspond to a “zero,” then the microprocessor makes the voltage on the line zero.

The 16 address lines are numbered

AD15, AD14, AD13, . . . , AD2, AD1, AD0

Imagine these 16 lines as being *ordered* with AD0 on the right and AD15 on the left. Then they can be thought of as representing a 16-bit binary number

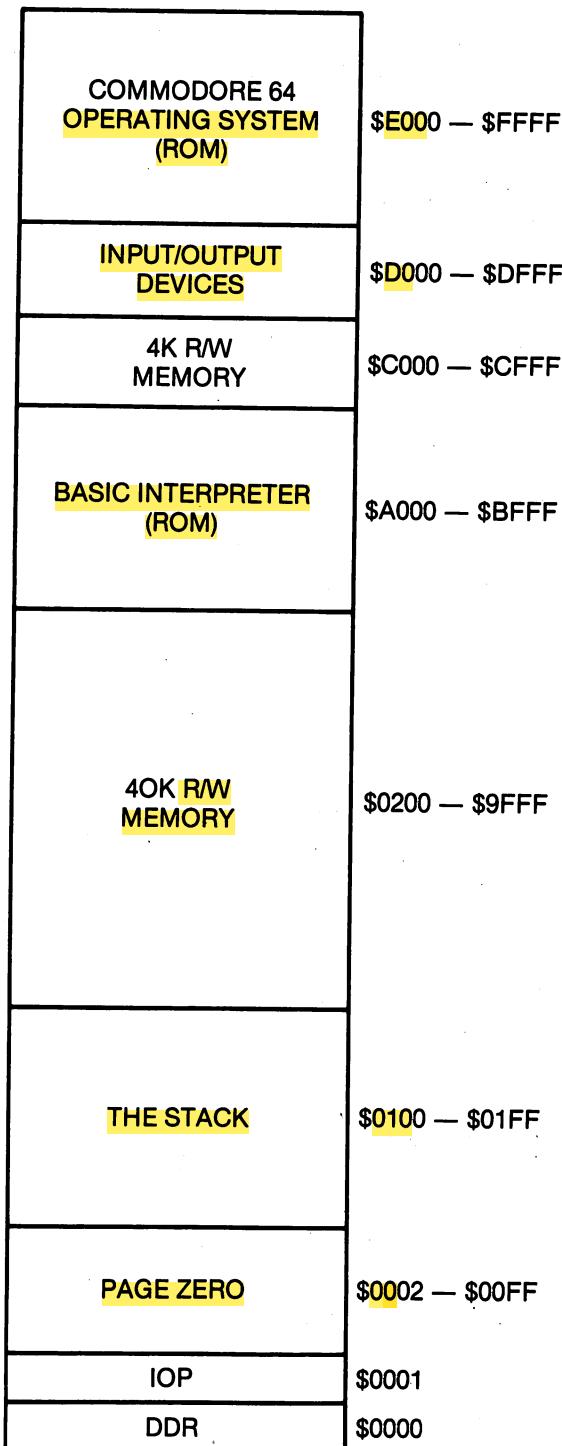


Figure 1-3. Memory map of the Commodore 64 microcomputer. To show certain details, this map is not drawn to scale.

where AD0 is the *least-significant bit*, the rightmost bit, and AD15 is the *most-significant bit*, the leftmost bit. Then the available addresses are:

BINARY	DECIMAL	HEXADECIMAL
0000 0000 0000 0000	0	\$0000
0000 0000 0000 0001	1	\$0001
0000 0000 0000 0010	2	\$0002
.	.	.
.	.	.
1111 1111 1111 1111	65535	FFFF

where we have collected the bits into groups of four for easy reading. The smallest address is 0000 0000 0000 0000 and the largest possible address is 1111 1111 1111 1111.

How many memory locations can be identified with this scheme? Suppose you had a dumb computer with only one address line. Then you would have two addresses, namely, zero and one, so you could address two memory locations. If the computer had two address lines, then you could have four addresses, namely, 00, 01, 10, and 11, corresponding to the four binary numbers that you can construct from two binary digits. This, only slightly smarter, computer could address four memory locations. A computer with three address lines could address eight locations. Notice that a pattern develops since

$$\begin{aligned}2^1 &= 2 \\2^2 &= 4 \\2^3 &= 8\end{aligned}$$

In other words, the number of memory locations that can be addressed by a microcomputer with n address lines is 2^n . Thus, if a microprocessor had 10 address lines, it could identify 2^{10} or 1,024 memory locations. A group of 1,024 memory locations is commonly called a "K" of memory. If you had 2K of memory you would have 2,048 memory locations. The 6510 microprocessor inside the Commodore 64 has 16 address lines so it can address 2^{16} or 65,536 locations. $65,536 = 64 \times 1,024$ so the Commodore 64 can address 64K of memory.

Actually, the "64" in the name of this computer refers to the amount of R/W memory available in the Commodore 64. With the memory management scheme mentioned earlier, the Commodore can switch in large banks of R/W memory to replace ROM. Large amounts of R/W memory are important for certain tasks, such as word processing. Large amounts of R/W memory are also necessary if you wish to use a high-level language other than BASIC, such as Pascal. When another language is used, the BASIC interpreter in ROM is of no use, and it is extremely useful to be able to replace it with R/W memory.

Suppose the microprocessor identifies the particular memory location it wishes to read by placing the address on the address bus. How do the eight bits of information get from the memory location to the microprocessor? There is another bus that connects the various components in the microcomputer system. This eight-conductor bus is called the *data bus*. The eight data bus conductors are numbered

DB7, DB6, DB5, DB4, DB3, DB2, DB1, DBO

When a memory location is **read**, each of the eight bits in a memory location controls the voltage level on a corresponding line of the data bus. The microprocessor interprets the voltage levels as binary ones or zeros, and therefore it knows **what eight-bit code is stored in the memory location it identified**.

A **byte** can be transferred from the microprocessor to a R/W memory location in a similar fashion. First, the microprocessor places on the address bus the address of the memory location to which it wishes to write. Then the microprocessor modifies the voltage levels on the data bus to represent the eight-bit code it wishes to transfer to memory. The memory device interprets these voltage levels as binary ones and zeros and stores them.

Notice that information can go in **two directions on the data bus**. In a **read** operation, an **eight-bit code** is transferred from memory to the microprocessor. In a **write** operation, an **eight-bit code** is transferred from the microprocessor to memory. This is why the data bus is called a **bidirectional bus**. We have described this in Figure 1-1 by having the arrows on the data bus point in two directions, indicating that information can flow in two directions.

Here are a few more subtle problems to consider. How does a memory location **distinguish between read and write operations** when it is addressed by the microprocessor? Moreover, how does it know **when it should respond**? The answers lie in additional information that the microprocessor supplies on the **control bus**. One line on the control bus is called the **R/W (read/write) line**. If a **read** operation is to take place, the microprocessor will place a **five-volt** signal on the R/W line. If a **write** operation is to take place, the microprocessor will bring the R/W line to **zero volts**.

Another line on the control bus carries a **clock signal**. All read and write operations are synchronized with this clock signal so that the various components of the system know when they are to respond. In the Commodore 64 system, the clock ticks at a rate of 1,022,727 times a second. In other words, it takes a little less than one millionth of a second (one microsecond) for a read or write operation to take place. The clock signal actually is a square wave that oscillates back and forth between zero volts and five volts at a regular rate. The clock rate is not described in terms of ticks, but rather in terms of **clock cycles**. A clock cycle is described in Figure 1-4. In the case of the Commodore 64, the clock rate is 1,022,727 cycles per second. A cycle per second is a Hertz, abbreviated Hz. Thus, the clock frequency is **1.022727 MHz** (Mega is one million).

IV. The Microcomputer in Action

You now know how information is moved around inside the microcomputer. Although read and write operations are of the greatest importance to the operation of a microcomputer, very little is accomplished by **simply moving eight-bit codes back and forth between the 6510 microprocessor and memory**. **Information must eventually be processed**. We turn next to a brief description of some of the **processes or operations** that can be achieved by the 6510 and an explanation of how they are accomplished.

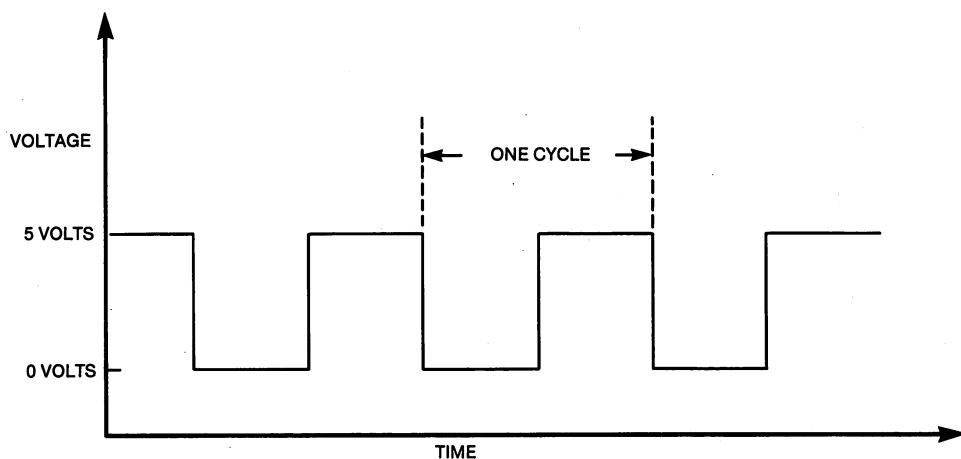


Figure 1-4. The clock signal of the Commodore 64 microcomputer system. The clock frequency is 1.022727 MHz.

To process information, the 6510 microprocessor was designed and built to carry out a certain number of *instructions*. These instructions make up the *instruction set* of the microprocessor. An instruction consists of **one, two, or three** eight-bit codes. Because it is difficult for human beings to remember eight-bit codes, the *instruction set* is also described with a group of **three-letter "words"** called *mnemonics*. Some examples are **LDA, STA, BIT, ROL, and JSR**. The mnemonics of the 6510 instruction set and an English-language description of each instruction is given in Table 1-1. This book is, in one sense, **an elaboration of Table 1-1. The mnemonics are central to programming in assembly language.**

Table 1-1. English-language description of the 6510 instruction set.

Mnemonic	Description
Data Transfer Instructions	
LDA	Copy the code in a memory location into the accumulator .
LDX	Copy the code in a memory location into the X register.
LDY	Copy the code in a memory location into the Y register.
STA	Transfer the code in the accumulator to a location in memory.
STX	Transfer the code in the X register to a location in memory.
STY	Transfer the code in the Y register to a location in memory.
TAX	Transfer the code in the accumulator to the X register.
TXA	Transfer the code in the X register to the accumulator.
TAY	Transfer the code in the accumulator to the Y register.
TYA	Transfer the code in the Y register to the accumulator.

Table 1-1. English-language description of the 6510 instruction set (continued).

Mnemonic	Description
Arithmetic and Logical Instructions	
ADC	Add the number in a memory location to the number in the accumulator . Add the value of the carry flag . Place the sum in the accumulator .
SBC	Subtract the number in a memory location from the number in the accumulator . Complement and subtract the value of the carry flag . Place the difference in the accumulator .
AND	Form the logical AND of the code in a memory location and the code in the accumulator . Place the result in the accumulator .
ORA	Form the logical OR of the code in a memory location and the code in the accumulator . Place the result in the accumulator .
EOR	Form the EXCLUSIVE OR of the code in a memory location and the code in the accumulator . Place the result in the accumulator .
Test Instructions	
CMP	Subtract the number in a memory location from the number in the accumulator . Modify the flags in the P register .
CPX	Subtract the number in a memory location from the number in the X register . Modify the flags in the P register .
CPY	Subtract the number in a memory location from the number in the Y register . Modify the flags in the P register .
BIT	Form the logical AND of the code in a memory location and the code in the accumulator . Modify the Z flag . Copy bit seven of the memory location into the N flag . Copy bit six of the memory location into the V flag .
Register Shift and Modify Instructions	
INC	Increment the number in a memory location by one.
DEC	Decrement the number in a memory location by one.
INX	Increment the number in the X register by one.
DEX	Decrement the number in the X register by one.
INY	Increment the number in the Y register by one.
DEY	Decrement the number in the Y register by one.
ASL	Shift the code in a memory location or the accumulator left by one bit. Place a 0 in bit zero. Shift bit seven into the carry flag .
LSR	Shift the code in a memory location or the accumulator right by one bit. Place a 0 in bit seven. Shift bit zero into the carry flag .
ROL	Rotate the code in a memory location or the accumulator left by one bit. Rotate the carry flag into bit zero and bit seven into the carry flag .
ROR	Rotate the code in a memory location or the accumulator right by one bit. Rotate the carry flag into bit seven and bit zero into the carry flag .

Table 1-1. English-language description of the 6510 instruction set (continued).

Mnemonic	Description
Flag Set and Clear Instructions	
CLC	Clear the carry flag C to zero.
SEC	Set the carry flag C to one.
CLD	Clear the decimal mode flag to zero.
SED	Set the decimal mode flag to one.
CLI	Clear the interrupt disable flag I to zero.
SEI	Set the interrupt disable flag I to one.
CLV	Clear the overflow flag V to zero.
Branch Instructions	
BCC	Branch if the carry flag C is clear.
BCS	Branch if the carry flag C is set.
BNE	Branch if the zero flag Z is clear.
BEQ	Branch if the zero flag Z is set.
BPL	Branch if the negative flag N is clear.
BMI	Branch if the negative flag N is set.
BVC	Branch if the overflow flag V is clear.
BVS	Branch if the overflow flag V is set.
Unconditional Jumps and Returns	
JMP	Jump to a new address and continue program execution.
JSR	Jump to a subroutine to continue execution.
RTS	Return from a subroutine to the calling program.
BRK	Break (jump) to execute the IRQ-type interrupt routine.
RTI	Return from the IRQ-type interrupt routine.
Stack Operation Instructions	
PHA	Push the code in the accumulator onto the stack.
PLA	Pull the code from the stack and place it in the accumulator.
PHP	Push the code in the P register onto the stack.
PLP	Pull the code from the stack and place it in the P register.
TXS	Transfer the number in the X register to the stack pointer S.
TSX	Transfer the number in the stack pointer S to the X register.
No Operation	
NOP	No operation takes place.

Repeating ourselves, the microprocessor accepts instructions in the form of eight-bit codes, not the mnemonics and descriptions given in Table 1-1. Human beings, not the microprocessor, use mnemonics.

An instruction can also be described with a diagram. We have chosen to describe the LDA and STA instructions in Figures 1-5 and 1-6. In the case of the LDA instruction, an eight-bit code is transferred over the data bus from memory to the 6510 microprocessor, using the ability of the 6510 to *read*, or copy, an eight-bit code in memory. The LDA instruction is analogous to the BASIC language PEEK instruction. In the case of the STA instruction, an eight-bit code is transferred over the data bus from the 6510 microprocessor to memory, using the ability of the 6510 to *write* an eight-bit code to memory. The STA instruction is analogous to the BASIC language POKE instruction.

The eight-bit codes that make up an instruction are stored in sequential memory locations. The addresses of these memory locations serve much the same purpose as line numbers in a BASIC program. A *machine-language program* consists of an ordered set of instructions designed to accomplish a specific objective. In running a machine-language program, the microprocessor reads, in succession, the eight-bit codes that comprise the instructions to be executed, and the microprocessor performs the desired operations in sequence.

It is possible, therefore, to distinguish between two kinds of information that are stored in memory, *programs* and *data*. For example, if you are using a word processing program, the word processing program itself occupies one portion of memory, while the words you type on the keyboard are stored as data in another portion of memory. In the area of memory devoted to the program, the microprocessor is busily engaged in reading and executing machine-language instructions. The data it processes is, with some exceptions, stored at another place in memory.

An example of an actual machine-language program should make all this clear. Our program will add two numbers and store the result. First let us load the program, then we will load the data, and then we will execute the program. Following that, we will briefly explain the program. You are now going to load and execute your first machine-language program. We will use BASIC POKE commands to store three instructions of a machine-language program in memory. Although the POKE instruction uses decimal numbers, the BASIC interpreter converts these decimal numbers into binary numbers in order to store them in the memory location specified by the first number in the POKE instruction. Perform the following POKE commands:

```
POKE 49152, 173  
POKE 49153, 52 )  
POKE 49154, 194 )  
POKE 49155, 109 )  
POKE 49156, 53 )  
POKE 49157, 194 )  
POKE 49158, 141 )  
POKE 49159, 0 )  
POKE 49160, 207 )  
POKE 49161, 96 )
```

The machine-language *program* is now in memory. Do not be concerned with the origin of the codes that you POKEed into memory. That mystery will be solved in Chapter 2.

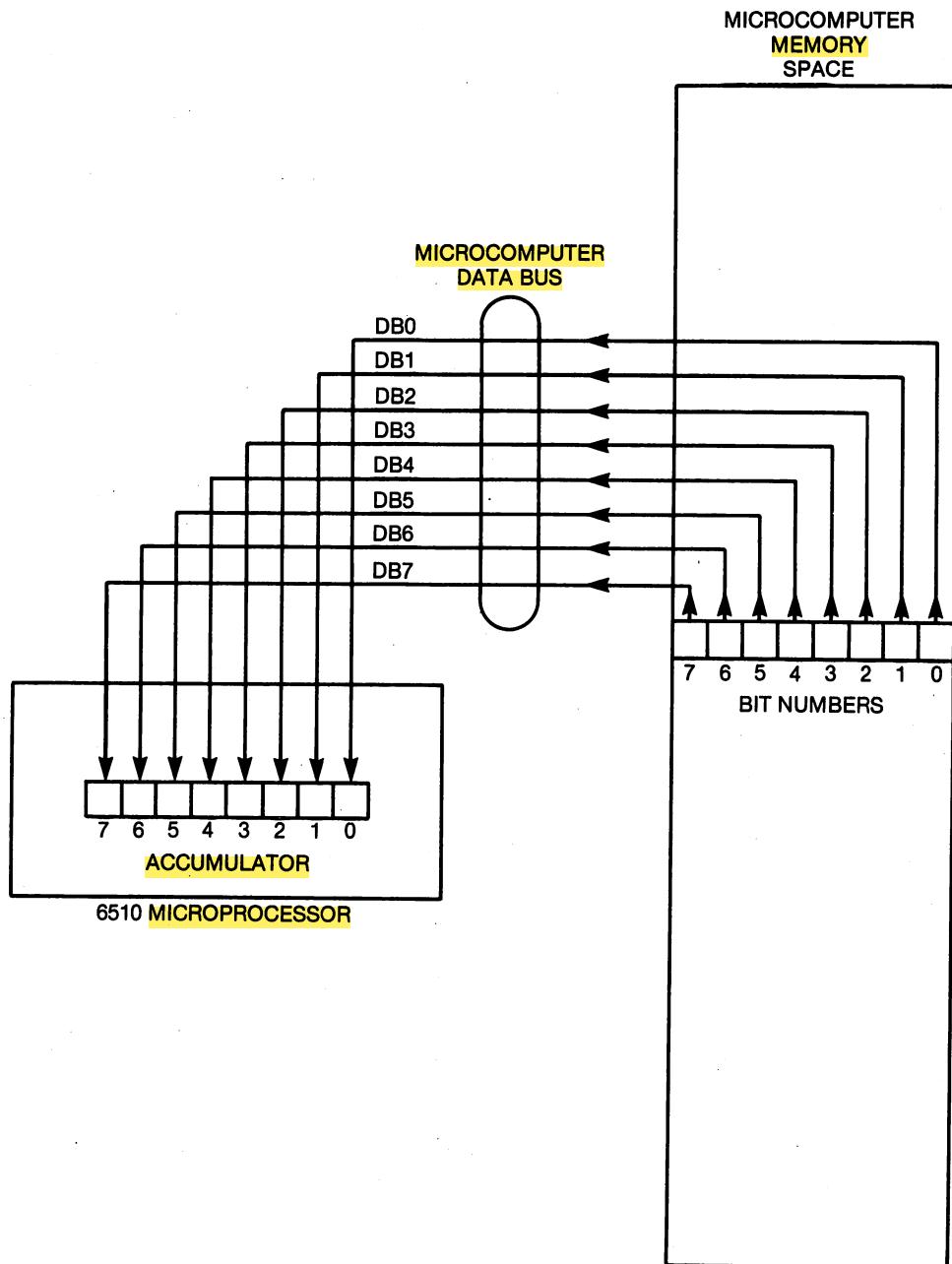


Figure 1-5. Diagram of the LDA instruction.

Now let us POKE the *data* into memory. The data consists of the two numbers to be added. Let us add two and three to get five. One addend goes in location \$C234 (49716) and the other in location \$C235 (49717). Perform these two commands:

POKE 49716, 2

POKE 49717, 3

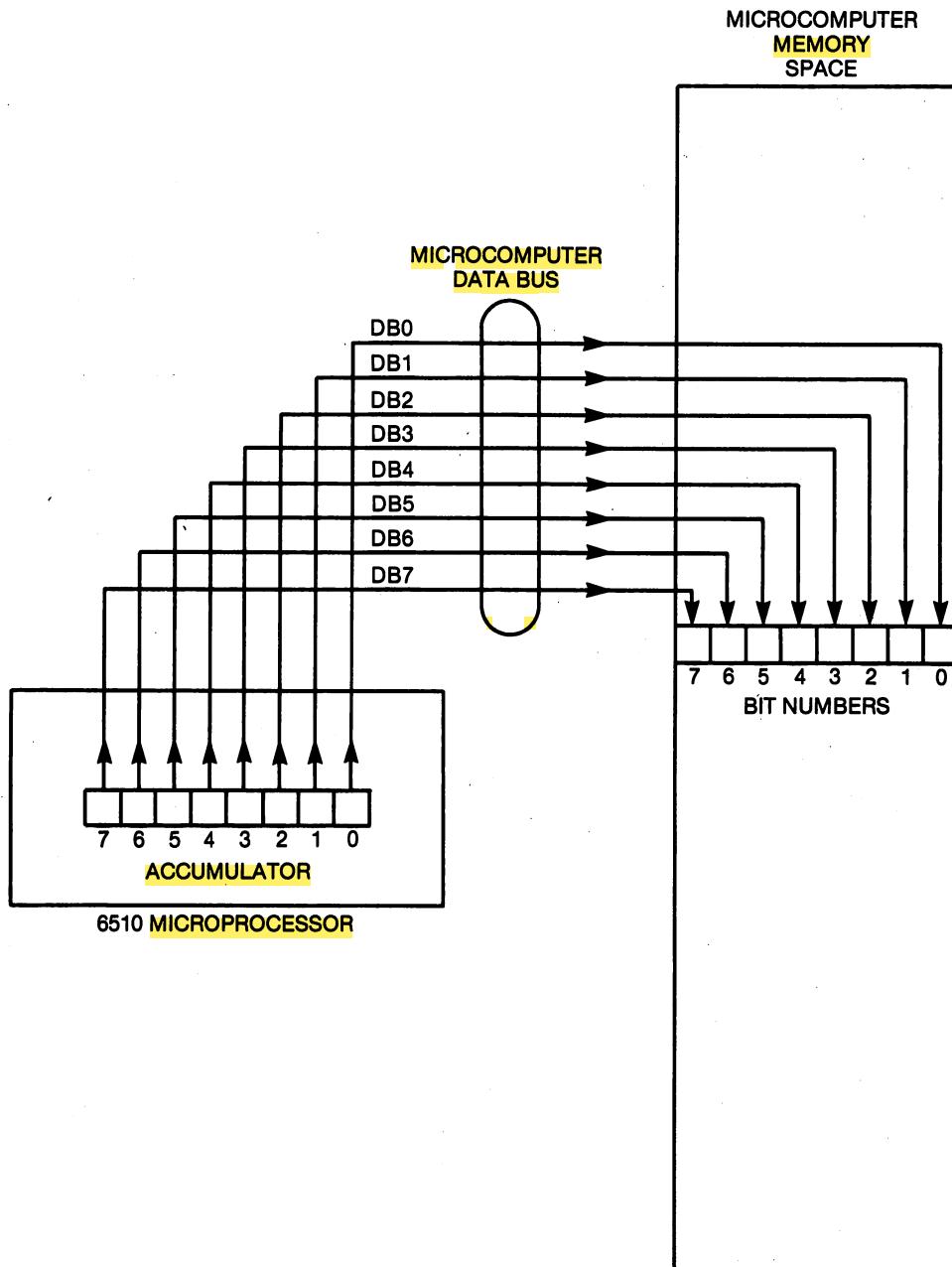


Figure 1-6. Diagram of the STA instruction.

To execute this machine-language program, perform the BASIC command
SYS 49152

which directs BASIC to perform a machine-language program whose instructions begin at the address \$C000 (49152).

The machine-language program stores the answer in memory location **\$CF00 (52992)**. After executing the program, perform the BASIC command

PRINT PEEK(52992)

The PEEK instruction reads a binary number in memory and converts it into a decimal number. You should get an answer of five. (You will get six for an answer if the 6510 still contained a "carry" from a previous addition. More about that later.)

You have just stored and executed a machine-language program. The purpose of this exercise is to begin to acquaint you with the idea that machine-language programs consist of binary codes stored in sequential memory locations. Now let us describe this program in a little more detail.

The program adds an eight-bit number stored in the memory location whose address is **\$C234** to an eight-bit number stored in the memory location whose address is **\$C235**, and it stores the sum of these numbers in the memory location whose address is **\$CF00**. Using a read operation, the number in location **\$C234** is first copied into the eight-bit register in the 6510 microprocessor where all arithmetic operations take place. This eight-bit register is known as the **accumulator**. An **LDA** instruction accomplishes this task (refer to Figure 1-5). Next, the microprocessor must add the number just read to the number stored in location **\$C235**. An **ADC** instruction accomplishes this. The ADC instruction is analogous to the "+" operation in the BASIC language. Finally, the 6510 must use a write operation to store the sum in location **\$CF00**. An **STA** instruction performs this task, completing the program (refer to Figure 1-6).

The machine-language program that achieves this simple programming task is shown in Table 1-2 in binary and hexadecimal. The decimal version was given above in the series of POKE commands. The mnemonics for the instructions are also given, to make the program more understandable. The three instructions have been delineated by spaces for easy identification. Below the program, we have given a simple example of the actual data that we used when we executed the program. Study the machine-language program in Table 1-2. Observe that each of the three instructions in this simple program consists of three bytes. The first byte identifies the nature of the operation that is going to take place. The second and third bytes give the address of the memory location of the data. Notice that the least-significant eight bits of the address are given first and the most-significant eight bits of the address are given last. Although you may think this is awkward, there are good reasons why the microprocessor was designed to work in this way.

It is important to realize that in addition to informing the microprocessor what instruction is desired, the program must specify, by one means or another, where the data is to be found. In the example just illustrated, each instruction specifies the full 16-bit address of the memory location of the data.

Table 1-2. A program to add two numbers.

Program Memory Address (Hexadecimal)	Program Memory Contents (Binary)	Program Memory Contents (Hexadecimal)	Assembly- Language Mnemonic
\$C000	10101101	\$AD	LDA
C001	00110100	34	
C002	10100010	C2	
C003	00111101	6D	ADC
C004	00110101	35	
C005	10100010	C2	
C006	10001101	8D	STA
C007	00000000	00	
C008	10101111	CF	
<hr/>			
Data Memory Address (Hexadecimal)	Data Memory Contents (Binary)	Data Memory Contents (Hexadecimal)	
\$C234	00000010	\$02	
\$C235	00000011	\$03	
.	.	.	
.	.	.	
.	.	.	
\$CF00	00000101	\$05	(after execution)

Here is how the program in Table 1-2 is executed. The 6510 begins by reading the first byte of the LDA instruction. After reading this byte, which takes less than one microsecond, it *interprets* this eight-bit code while it is reading the next program byte. Once it interprets the first code, it knows that the second and third bytes of the LDA instruction are the address of the data it is to read from memory. It takes three clock cycles of the microcomputer system clock to read all three bytes of the instruction. During the next clock cycle, it reads the byte of data stored at location \$C234. Thus, the entire three-byte LDA instruction takes four clock cycles. Three clock cycles were used to read the instruction, and the fourth was used to carry out the intent of the instruction.

Next, the microcomputer reads and interprets the first byte of the add (ADC) instruction. Having interpreted the eight-bit code, it knows that the next two program bytes are the address of the memory location of the number it is to add to the number in the accumulator. It reads all three bytes of the instruction in three clock cycles, one for each byte. During the fourth clock cycle, it adds the number in location \$C235 to the number in the accumulator and replaces the number in the accumulator with this sum. The ADC instruction also takes four clock cycles.

Finally, the microprocessor reads and interprets the first byte of the STA instruction. The eight-bit code informs it that it must write the contents of the accumulator to the address given in the next two program bytes. This takes one clock cycle. During the next two clock cycles, it reads both bytes of the address of the destination of the number in the accumulator. During the fourth and final clock cycle, it stores (writes a copy of) the number in the accumulator into location \$CF00, completing the program in 12 clock cycles, or just under 12 microseconds.

Admittedly this is a simple program, and we have taken the liberty of slightly oversimplifying the operation of the microprocessor. By now you should have some inkling of the nature of machine language and the sequence of events associated with the execution of a machine-language program. Also, you are now in an excellent position to proceed to the next section, where we will introduce some of the other registers in the 6510 microprocessor. Furthermore, you are also almost ready to begin programming. In the next few chapters we will describe much more efficient ways of creating, storing, and executing machine-language programs.

V. An Overview of the 6510 Registers

The registers in the 6510 microprocessor that are of interest to the assembly-language programmer are shown in Figure 1-7. The bit numbers are identified at the top of the accumulator and the program counter. We will briefly discuss each of these registers. Think of this as a global view. When you look at a globe you do not see or understand many details. These will become obvious only with more study and the use of the registers in programming examples.

The Accumulator—A

You have already been introduced to the accumulator. It is used by the assembly-language programmer more often than any other register. When performing an arithmetic operation, such as addition, two numbers called *operands* are combined to form the sum. In performing arithmetic or logical operations with the 6510, one of the two operands *must* be in the accumulator; the other operand will be in memory. The result of the operation is always placed in the accumulator.

Recall that when programming with BASIC you combined *variables* in various arithmetic operations. In assembly-language programming, one of the variables is always A, the symbol for the number in the accumulator. The other variable will be M, a symbol for a number in some memory location. In addition to being used for arithmetic and logical operations, the accumulator may be used for data transfers to and from memory by means of the LDA and STA instructions described in Table 1-1 and Figures 1-5 and 1-6.

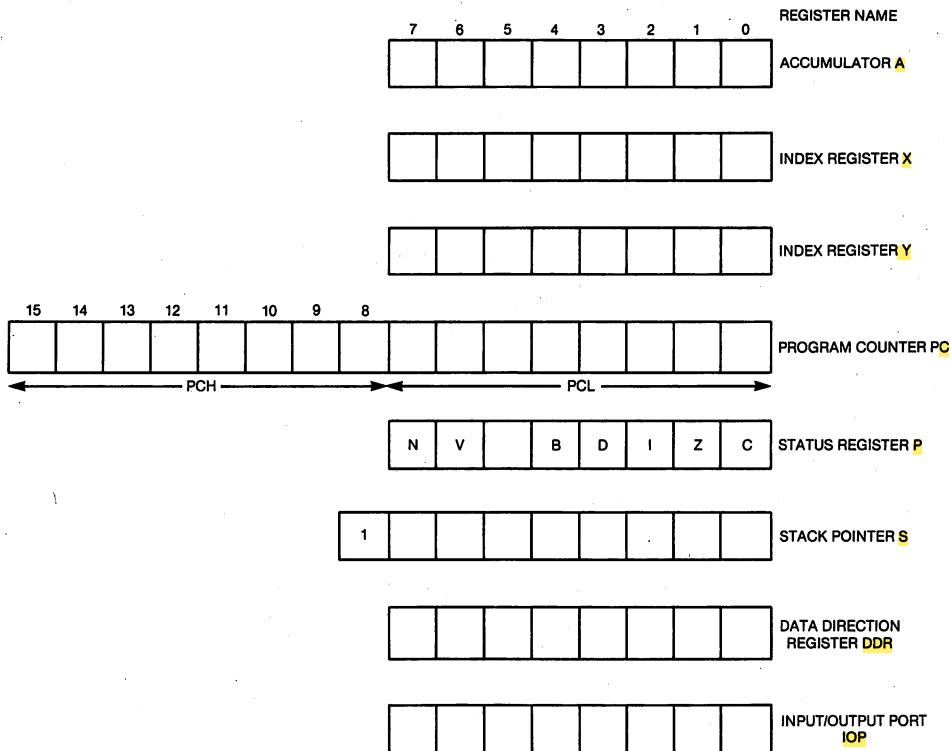


Figure 1-7. Register structure of the 6510 microprocessor.

The X Index Register—X

In the BASIC language, the programming structure known as a *loop* is frequently implemented with a FOR...NEXT... instruction. Some variable serves as a *loop counter*. The variable is incremented by one until it reaches a certain value and then the program exits the loop. In assembly language, the number in the X register will frequently serve as a loop counter.

The BASIC language allows the use of *array variables*, of the form VAR(I), where I is called a *subscript* or *index*. When programming in assembly language, you will frequently use the number in the X register as a subscript, or index. That is why it is called an *index register*.

The X register can also be used for transferring eight-bit codes to and from memory with the LDX and STX instructions (refer to Table 1-1).

The Y Index Register—Y

The Y register is used in the same way as the X register; that is, used as a loop counter and as a subscript or index. The Y register can also be used for transferring data with the LDY and STY instructions (refer to Table 1-1).

The Program Counter—PC

A computer program written in any language is an *ordered* set of instructions. The BASIC language uses *line numbers* to order the instructions; how are the instructions ordered in machine-language and assembly-language programs? We have already hinted at the answer. The instructions are stored one after the other in memory and are ordered by the *addresses* of the memory locations that the program occupies.

The program counter is the register in the 6510 microprocessor that keeps track of the address of the next program byte to be read by the 6510. Since addresses are 16-bit numbers, the program counter is a 16-bit register. If you could watch the program counter during the execution of a program, the program counter would appear to count, except when branches or jumps occur: then it would appear to jump forward or backward in its counting.

The Processor Status Register—P

Refer again to Figure 1-7. Notice that the P register is the only one that has specific bits identified. Each bit (with the exception of bit five, which is not used) is called a *flag* or *condition code*. These flags are modified, set or cleared, by operations and events that take place during program execution. A flag is set if it has the value one, it is clear if it has the value zero. For example, if the addition of two eight-bit numbers produces a *carry*, then the carry flag, C, will be set; otherwise, it will be cleared. If, for example, an AND instruction gives a result of *zero*, then the zero flag, Z, is set; otherwise, it is cleared. So-called *branch instructions* are used to test the flag conditions and force the program to branch to another instruction, very similar to the IF...THEN... instruction in BASIC. The flags will be discussed as the need arises in subsequent chapters.

The Stack Pointer—S

This register is used to *point* to the locations in memory whose addresses lie from \$0100 to \$01FF, called the *stack*. The number in the stack pointer identifies the least-significant byte (\$00 - \$FF) of the address. The most-significant byte of the address is *understood* to be \$01. That is why the "one" appears in bit eight of the stack pointer (refer to Figure 1-7). The stack is used to store information when subroutines and interrupts are executed. It will be described in more detail in Chapter 9.

The Input/Output Port Registers—DDR and IOP

The 6510 microprocessor is a member of the 6502 family of microprocessors. The 6510 differs from some of the other family members because it has an onboard input/output port. This port can input or output up to eight bits of

information. The address of the I/O port is \$0001. Its data-direction register is located at \$0000. The number in the DDR (data-direction register) simply determines whether a particular bit of the port will be used to input information or to output information. There is a one-to-one correspondence between the bits in the DDR and the bits in the IOP. A one in a particular bit in the DDR makes the corresponding bit in the IOP an *output bit*. A zero in a particular bit in the DDR makes the corresponding bit in the IOP an *input bit*. The Commodore 64 system uses this I/O port to switch ROM in and out of the memory space and to perform cassette read and write operations. It is not considered to be a user port, but in Chapter 11 we will illustrate how it is used to switch banks of memory in and out of active use.

This completes our overview of the structure and operation of the Commodore 64 microcomputer system. You should have a much greater appreciation of how it works. This background will become more and more useful as you begin programming in assembly language, a language in which the programmer is more intimately associated with the operation of the microcomputer than any other language. Clearly, to use BASIC you did not need to know anything about the registers in the microprocessor, but this knowledge is indispensable for assembly-language programming.

VI. Additional Reading

Since the 6510 microprocessor is a member of the 6502 family of microprocessors, most of the literature associated with the 6502 is useful reading material for Commodore 64 assembly-language programmers. We recommend the following books:

Commodore 64 Programmer's Reference Guide (PRG). Commodore Business Machines, Indianapolis, Indiana: Howard W. Sams & Co., Inc., 1982. (Indispensable.)

6502 Assembly Language Programming. Lance A. Leventhal. Berkeley, California: Osborne/McGraw-Hill, Inc., 1979. (Excellent reference for assembly-language programmers with some experience.)

6502 Software Design. Leo J. Scanlon. Indianapolis, Indiana: Howard W. Sams & Co., Inc., 1980. (Excellent source for a variety of 6502 assembly-language routines.)

6502 Software Gourmet Guide & Cookbook. Robert Findley. Elmwood, Connecticut: Scelbi Publications, 1979. (Contains a number of valuable 6502 programming examples, including floating-point arithmetic routines.)

Programming & Interfacing the 6502, With Experiments. Marvin L. De Jong. Indianapolis, Indiana: Howard W. Sams & Co., Inc., 1980. (An introductory text to assembly-language programming and microcomputer hardware.)

VII. Summary

The Commodore 64 microcomputer consists of a 6510 microprocessor, ROM, R/W memory, and various I/O devices. These are interconnected by the address bus, the data bus, and the control bus. Each of the 65,536 memory

locations is identified by its address. Each memory location stores eight bits of information, called an eight-bit code, an eight-bit number, or a byte. Memory is used to store both data and machine-language programs. The microprocessor can read a code stored in memory, operate on it, or store (write) a code in memory under the direction of a machine-language program. A program is an ordered set of instructions stored sequentially in memory. Instructions consist of eight-bit codes that the microprocessor reads, interprets, and executes. In order to be understood by human beings, the instructions are described by a mnemonic and an English language description. The instruction set of the 6510 consists of 56 different instructions. Six registers inside the 6510 are involved in the execution of these instructions. These registers are the accumulator, X register, Y register, program counter, processor status register, and the stack pointer.

2

Writing, Assembling, and Executing Programs

I. Introduction

In addition to being challenging and entertaining, learning to program your computer using assembly language is useful. Assembly-language programs are never executed; they are always translated to machine language. In fact, programming with assembly language is simply an easy way to write machine-language programs. Almost every machine-language program is written using assembly language. For certain applications such as animated graphics, music synthesis, and speech recognition, the high execution speed associated with machine-language programs is essential. Programs designed to interface the microcomputer to I/O devices such as printers, modems, and laboratory instruments are usually written using assembly language. Obviously, both the operating system and the BASIC interpreter were written using assembly language. These assembly-language programs were then translated into machine language before being stored in the ROM of the Commodore 64.

To summarize, assembly language is used whenever speed is important, but it is also used in many other programs, such as word processors and spread sheets.

II. Microcomputer Instructions

A microcomputer machine-language program is an *ordered* set of instructions. An *instruction* consists of one, two, or three eight-bit codes stored in memory, which when read by the microprocessor will cause it to carry out one of the 56 operations listed in Table 1-1.

The *first* eight-bit code of an instruction determines which of the 56 specific operations in the instruction set will be executed. This eight-bit code is called the *operation code* or, more concisely, the *op code* of the instruction. Writing

eight-bit codes is lengthy, and you seldom see the binary representation of the op code. It is more concise to express these codes in hexadecimal. Hexadecimal, a numbering system with a base of 16, is described in more detail in Appendix A. The hexadecimal representations of the op codes are listed in Table 2-1, which summarizes the 6510 instruction set.

As an aid in programming, each instruction is also given a three-letter *mnemonic* that suggests what operation is involved. These instruction mnemonics were listed in Table 1-1 and are given again in Table 2-1. Before very long you will have all 56 mnemonics memorized.

In addition to the English-language description given for each instruction in Table 1-2, we can give the *logical description* of the instruction. The logical description is simply a *symbolic* (and, therefore, concise) description of the instruction. The logical descriptions of each instruction are given in the second column of Table 2-1. In Example 2-1 we summarize the four ways of identifying an instruction for three different instructions, LDA, ADC, and STA.

Example 2-1. Illustrating the Four Ways to Identify an Instruction

Mnemonic	Logical Expression	Op Code	English Description
LDA	$M \rightarrow A$	\$AD	Copy the code in memory location M into the accumulator A.
ADC	$A + M + C \rightarrow A$	\$8D	Add the number in A, the number in memory location M, and the contents of the carry flag. Store the sum in A.
STA	$A \rightarrow M$	\$6D	Write the code in A to memory location M.

Assembly-language programs are written using mnemonics. The process of translating mnemonics and other parts of the assembly-language program to machine language is called *assembling* the program. An *assembler* is a computer program that performs the translation for you. For short programs, hand assembly using pencil, paper, and the instruction set summary in Table 2-1 is perfectly suitable. In my opinion, you can learn a great deal about assembly language *without* the added expense of an assembler. It is also more difficult to try to learn assembly language while trying to understand the idiosyncrasies of the assembler. Although other experienced programmers may disagree, my advice is to wait to purchase an assembler until you have mastered the first half of this book. Later, when you begin to write long programs, you will want to purchase an assembler. We will mention more about assemblers in Chapter 3.

III. Addressing Modes

As stated earlier, an instruction can consist of up to three eight-bit codes. We have described the function of the first eight-bit code but we have not yet accounted for the other bytes of the instruction. Refer once again to the instruction set summarized in Table 2-1. The instruction mnemonic is found in the first column and the logical description is found in the second column. The next 13 columns contain column headings such as "IMMEDIATE," "ABSOLUTE," "ZERO PAGE," and so on, each of which is a different addressing mode. The *addressing mode* is related to *how the microprocessor locates* (in memory) the number involved in the instruction. The "number involved" in the instruction is usually called the operand. The *operand* is the eight-bit code or number that is the *object* of the instruction. (Just as every preposition in English grammar must have an object to which it refers, an assembly-language instruction has an object, in this case a number rather than a noun, to which it refers.)

Some simple examples will suffice to illustrate the concept of an operand. In the case of an LDA instruction, the operand is the number copied from memory into the accumulator. In the case of the ADC instruction, the operand is the number that is added to the number in the accumulator. In the case of the STA instruction, the operand is the number that is transferred from the accumulator to memory. Notice that what happens to the operand depends on the instruction. Sometimes it is read (LDA), sometimes it is stored (STA), sometimes it is one of the two numbers involved in an arithmetic operation (ADC or SBC), sometimes it is one of the two codes involved in a logical operation (AND, ORA, EOR), sometimes it is shifted (ASL, LSR), and so forth.

Before describing several of the addressing modes, we must make two simple definitions. Each 16-bit address is composed of two bytes. The least-significant byte of the address, which identifies address bits AD0-AD7, is called the *ADL* (*address low*) of the address. The most-significant byte of the address, which identifies address bits AD8-AD15, is called the *ADH* (*address high*) of the address. Sometimes the ADH is called the high byte of the address and the ADL is called the low byte of the address. Example 2-2 illustrates these ideas.

Example 2-2. Identifying the ADL and ADH of an Address

Identify the ADL and the ADH of the address \$3C1A.

Solution: The ADL is \$1A and the ADH is \$3C. The least-significant byte of the address is represented by the last two hexadecimal digits, therefore they are the ADL. The most-significant byte of the address is represented by the first two hexadecimal digits, therefore they are the ADH.

Table 2-1. The 6510 instruction set summary.
(Reprinted with permission of Semiconductor Products Division of Rockwell International)

INSTRUCTIONS		IMMEDIATE			ABSOLUTE			ZERO PAGE			ACCUM			IMPLIED			(IND. X)		(IND.) Y			
MNEMONIC	OPERATION	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#
A D C	A + M + C → A (4) (1)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2
A N D	A AND A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2
A S L	C ← [] 0 → 0				0E	6	3	06	5	2	0A	2	1									
B C C	BRANCH ON C = 0 (2)																					
B C S	BRANCH ON C = 1 (2)																					
B E Q	BRANCH ON Z = 1 (2)																					
B I T	AAM				2C	4	3	24	3	2												
B M I	BRANCH ON N = 1 (2)																					
B N E	BRANCH ON Z = 0 (2)																					
B P L	BRANCH ON N = 0 (2)																					
B R K	BREAK															00	7	1				
B V C	BRANCH ON V = 0 (2)																					
B V S	BRANCH ON V = 1 (2)															18	2	1				
C L C	0 → C															D8	2	1				
C L D	0 → D																					
C L I	0 → I															58	2	1				
C L V	0 → V															BB	2	1				
C M P	A - M	C9	2	2	CD	4	3	C5	3	2									C1	6	2	D1
C P X	X - M	E0	2	2	EC	4	3	E4	3	2									5	2		
C P Y	Y - M	C0	2	2	CC	4	3	C4	3	2												
D E C	M - 1 → M				CE	6	3	C6	5	2												
D E X	X - 1 → X															CA	2	1				
D E Y	Y - 1 → Y															88	2	1				
E O R	A ⊕ M → A (1)	49	2	2	4D	4	3	45	3	2									41	6	2	51
I N C	M + 1 → M				EE	6	3	E6	5	2												
I N X	X + 1 → X															E8	2	1				
I N Y	Y + 1 → Y															C8	2	1				
J M P	JUMP TO NEW LOC				4C	3	3															
J S R	JUMP SUB				20	6	3															
L D A	M → A (1)	A9	2	2	AD	4	3	A5	3	2									A1	6	2	B1

Table 2-1. The 6510 instruction set summary (continued).
(Reprinted with permission of Semiconductor Products Division of Rockwell International)

Z PAGE, X				ABS. X				ABS. Y				RELATIVE				INDIRECT				Z. PAGE. Y				PROCESSOR STATUS CODES								
OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	7	6	5	4	3	2	1	0	MNEMONIC
																								N	V	•	B	D	I	Z	C	
75	4	2	7D	4	3	79	4	3																N	V	•	•	•	•	•	Z C	ADC
35	4	2	3D	4	3	39	4	3																N	•	•	•	•	•	Z •	A ND	
16	6	2	1E	7	3							90	2	2										N	•	•	•	•	•	Z C	A SL	
												B0	2	2										•	•	•	•	•	•	•	BCC	
												F0	2	2										•	•	•	•	•	•	•	B CS	
												30	2	2										•	•	•	•	•	•	•	B EQ	
												D0	2	2										M, M _E	•	•	•	•	•	Z •	B IT	
												10	2	2										•	•	•	•	•	•	•	B MI	
																									•	•	•	•	•	•	•	B NE
																									•	•	•	•	•	•	•	B PL
																									•	•	•	1	•	1	•	BRK
																									•	•	•	•	•	•	•	B VC
																									•	•	•	•	•	•	•	B VS
																									•	•	•	•	•	0	•	CLC
																									•	•	•	•	0	•	•	CLD
																									•	•	•	•	0	•	•	CLI
D5	4	2	DD	4	3	D9	4	3																•	0	•	•	•	•	•	CLV	
																									N	•	•	•	•	•	Z C	C MP
																									N	•	•	•	•	•	Z C	C PX
																									N	•	•	•	•	•	Z C	C PY
D6	6	2	DE	7	3																				N	•	•	•	•	•	Z •	DEC
																									N	•	•	•	•	•	Z •	DEX
																									N	•	•	•	•	•	Z •	DE Y
55	4	2	5D	4	3	59	4	3																N	•	•	•	•	•	Z •	E OR	
F6	6	2	FE	7	3																				N	•	•	•	•	•	Z •	INC
																									6C	5	3					
B5	4	2	BD	4	3	B9	4	3																N	•	•	•	•	•	Z •	IN X	
																									N	•	•	•	•	•	Z •	IN Y
																									•	•	•	•	•	•	•	J MP
																									•	•	•	•	•	•	•	JS R
																									N	•	•	•	•	•	Z •	L DA

Table 2-1. The 6510 instruction set summary (continued).
(Reprinted with permission of Semiconductor Products Division of Rockwell International)

INSTRUCTIONS		IMMEDIATE		ABSOLUTE		ZERO PAGE		ACCUM		IMPLIED		(IND, X)		(IND), Y					
MNEMONIC	OPERATION	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#
L D X	M → X (1)	A2	2	2	AE	4	3	A6	3	2									
L D Y	M → Y (1)	A0	2	2	AC	4	3	A4	3	2									
L S R	0 → [] → C				4E	6	3	46	5	2	4A	2	1						
N O P	NO OPERATION													EA	2	1			
O R A	A V M → A	09	2	2	0D	4	3	05	3	2						01	6	2	11
P H A	A → Ms S - 1 → S													48	3	1			
P H P	P → Ms S - 1 → S													08	3	1			
P L A	S + 1 → S Ms → A													68	4	1			
P L P	S + 1 → S Ms → P													28	4	1			
R O L	[] → u → [C]				2E	6	3	26	5	2	2A	2	1						
R O R	[C] → [] → u				6E	6	3	66	5	2	6A	2	1						
R T I	RTRN INT													40	6	1			
R T S	RTRN SUB													60	6	1			
S B C	A - M - C → A (1)	E9	2	2	ED	4	3	E5	3	2						E1	6	2	F1
S E C	1 → C													38	2	1			
S E D	1 → D													F8	2	1			
S E I	1 → I													78	2	1			
S T A	A → M				8D	4	3	85	3	2						81	6	2	91
S T X	X → M				8E	4	3	86	3	2									6
S T Y	Y → M				8C	4	3	84	3	2									2
T A X	A → X													AA	2	1			
T A Y	A → Y													A8	2	1			
T S X	S → X													BA	2	1			
T X A	X → A													8A	2	1			
T X S	X → S													9A	2	1			
T Y A	Y → A													9B	2	1			

Table 2-1. The 6510 instruction set summary (continued).
(Reprinted with permission of Semiconductor Products Division of Rockwell International)

Z PAGE, X			ABS. X			ABS. Y			RELATIVE			INDIRECT			Z PAGE, Y			PROCESSOR STATUS CODES								MNEMONIC			
OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	7	6	5	4	3	2	1	0				
																		N	V	*	B	D	I	Z	C				
B4	4	2	BC	4	3	BE	4	3							B6	4	2	N	•	•	•	•	•	•	•	Z	•	LDX	
56	6	2	5E	7	3													N	•	•	•	•	•	•	•	Z	•	LDY	
15	4	2	1D	4	3	19	4	3										0	•	•	•	•	•	•	•	Z C	•	LSR	
																		•	•	•	•	•	•	•	•	•	•	•	NOP
																		N	•	•	•	•	•	•	•	Z	•	ORA	
																		•	•	•	•	•	•	•	•	•	•	•	PHA
																		•	•	•	•	•	•	•	•	•	•	•	PHP
																		N	•	•	•	•	•	•	•	Z	•	PLA	
																		(RESTORED)											PLP
36	6	2	3E	7	3													N	•	•	•	•	•	•	•	Z C	•	ROL	
76	6	2	7E	7	3													N	•	•	•	•	•	•	•	Z C	•	ROR	
																		(RESTORED)											RTI
F5	4	2	FD	4	3	F9	4	3										N	V	•	•	•	•	•	•	Z (3)	•	SBC	
																		•	•	•	•	•	•	•	•	1	•	SEC	
																		•	•	•	•	1	•	•	•	•	•	SED	
95	4	2	9D	5	3	99	5	3										96	4	2									SEI
94	4	2																											STA
																													STX
																													STY
																		N	•	•	•	•	•	•	•	Z	•	TAX	
																			N	•	•	•	•	•	•	•	Z	•	TAY
																			N	•	•	•	•	•	•	•	Z	•	TSX
																			N	•	•	•	•	•	•	•	Z	•	TXA
																				•	•	•	•	•	•	•	•	•	TXS
																			N	•	•	•	•	•	•	•	Z	•	TYA
X INDEX X									+ ADD			M ₇ MEMORY BIT 7																	
Y INDEX Y									- SUBTRACT			M ₆ MEMORY BIT 6																	
A ACCUMULATOR									^ AND			n NO. CYCLES																	
M MEMORY PER EFFECTIVE ADDRESS									V OR			# NO. BYTES																	
Ms MEMORY PER STACK POINTER									▼ EXCLUSIVE OR																				

Another definition that must be made before proceeding is related to the special significance that the 6510 gives to the block of memory locations starting with address \$0000 and ending with address \$00FF. This block of memory locations is called *page zero* of memory, or zero page. Of course, the ADH of any memory location in page zero is \$00.

Now we are ready to proceed with our study of addressing modes. We begin by describing three of the simplest addressing modes:

- In the absolute addressing mode, the second and third bytes of the instruction are the ADL and ADH, respectively, of the address of the operand.
- In the zero-page addressing mode, the second byte of the instruction is the ADL of the address of the operand. The 6510 automatically makes the ADH = \$00.
- In the immediate addressing mode, the operand is the second byte of the instruction. In this case, the operand actually resides in the program.

Notice from the instruction set summarized in Table 2-1 that each addressing mode has a *unique* op code. Example 2-3 illustrates these ideas.

Example 2-3. Use of the LDA Instruction in Three Addressing Modes

Illustrate an LDA instruction for each of the three addressing modes just described.

Solution: Assume the operand is in location \$1234. Then the complete LDA instruction in the absolute addressing mode is, in order, AD 34 12. If the operand is in page zero of memory, then the zero-page addressing mode can be used. In this case, the LDA instruction consists of two bytes. Assume the operand is in location \$003F. Then the LDA instruction is, in order, A5 3F. In the immediate addressing mode, the second byte of the instruction is *the operand*. Suppose we want to load the number \$01 into the accumulator. The LDA instruction is, in order, A9 01.

In executing an instruction, the 6510 microprocessor first reads the op code of the instruction. The microprocessor interprets this op code to determine the nature of the operation *and* the addressing mode. Having interpreted the op code, the microprocessor knows how many bytes are in the instruction and it knows the significance of those bytes, that is whether they are a full address, the low-order byte of an address in page zero of memory, or a byte of data, as in the case of the immediate addressing mode. Thus, if the first byte of an instruction is \$AD, the 6510 knows that an LDA instruction is to be executed in the absolute addressing mode and that the next two bytes are the address of the operand.

To repeat, the first byte of any instruction is always the op code. The remaining bytes, if any, are used to determine the address of the memory location of the data to be operated on by the microprocessor. The op code contains sufficient information to identify both the instruction and the addressing mode.

(A detailed study of the op codes in Table 2-1 will, in fact, reveal which bits in the op code identify the addressing mode and which bits identify the type of instruction. It is not a coincidence that all three instructions illustrated in Example 2-1 have op codes that end in "D," since all three used the absolute addressing mode. Nor is it a coincidence that the LDA op codes all begin with an "A" or a "B." You need not be concerned, however, with how the engineers who designed the 6510 give it the ability to interpret the bit patterns in op codes.)

The number of bytes in each instruction is also specified in the instruction set summary in Table 2-1. At the intersection of the row containing the instruction mnemonic and the column containing the addressing mode you will find the op code, the number of bytes in the instruction (n column), and the number of clock cycles (# column) it takes the 6510 microprocessor to execute the instruction.

Notice that some instructions, such as TAX, TXA, PHA, require only one byte. How can the microprocessor find the operand with no information about its location? From the English-language description of the TAX instruction you know its execution will transfer the code from the accumulator to the X register. Both the source and the destination of the operand are *implied* by the instruction itself, and no additional information regarding the location of the operand is required. This form of addressing is called the *implied addressing mode*. An instruction that uses the implied addressing mode requires only a single byte, the op code.

Other addressing modes will be covered in subsequent chapters.

IV. The Components and Form of an Assembly-Language Program

To understand an assembly-language program you have to see one. Suppose our program objective is to place the number \$17 in memory location \$D018, and the program is to be stored in memory starting at location \$C000. This is clearly a rather simple objective that can be achieved with a simple program. An assembly-language program that will accomplish this task is given in Example 2-4.

Example 2-4. A Simple Assembly-Language Program

Label	Mnemonic	Operand Field	Comments
BEGIN	LDA	#\$17	; Load the accumulator with the ; number \$17.
END	STA	VIC18	; Store the number in the ; location symbolized by VIC18.

Refer to Example 2-4 and note the four columns of the program: *label*, *mnemonic*, *operand field*, and *comments*. The label is a name for the *address* of the first byte of the instruction. Many instructions will not have labels, but we have chosen the label BEGIN for the start of our program and the label END for the last instruction. The second column contains one of the 56 mnemonics found in the instruction set.

The third column is the *operand field*:

- It is empty when the implied addressing mode is used.
- It contains the operand when the immediate addressing mode is used. The immediate mode will use the “#” symbol.
- It contains a symbol when the zero page or absolute addressing modes are used.
- It contains a label when a branch, JMP, or JSR instruction is involved. Two of these cases are illustrated in Example 2-4. Others will be illustrated in subsequent chapters.

We intend to load the accumulator with \$17, using the LDA instruction in the immediate addressing mode. The operand is, therefore, in the program itself, and assembly-language programming tradition uses the “#” symbol to indicate the immediate addressing mode. The STA instruction uses absolute addressing since the memory location is not in zero page. VIC18 is a *symbol* for the location \$D018 where we wish to store the number in the accumulator. We chose the symbol “VIC18” because \$D018 is the address of the \$18th register of the 6567 VIC II chip, which we will refer to as the 6567 VIC, or more simply, the VIC.

The fourth column contains comments intended to inform the reader of the purpose of the instruction. You do not ordinarily describe the instruction itself as we have done in Example 2-4—the instruction mnemonic should do that. However, since you are in the early stages of learning assembly-language programming, some redundancy may help you to understand the instruction.

The program in Example 2-4 is, indeed, an assembly language program. It is of little use, however, unless it is translated into machine language, a process called assembling the program. This program can easily be assembled by hand using the instruction set summarized in Table 2-1. Rather than translating the assembly language version into binary numbers or machine code, it will be more convenient for us to represent these codes in hexadecimal. The traditional form of an assembly-language program *includes* the hexadecimal version, usually placed to the left of the assembly-language program.

Carefully study Example 2-5. It is the same program listed in Example 2-4, but the hexadecimal codes have been added and the comments have been modified. The hexadecimal translation of the “LDA #\$17” assembly-language instruction is

A9 17

since \$A9 is the op code for the LDA instruction in its immediate addressing mode, and \$17 is the operand of this instruction. Since the first instruction begins at location \$C000, we specify this address to the left of the instruction. Thus, the entire “LDA #\$17” instruction translates into

C000 A9 17

Refer again to Example 2-5.

Now we will assemble (translate into hexadecimal code) the second instruction. Recall that we specified that "VIC18" symbolizes the address \$D018. Thus, the machine-language translation of the STA VIC18 instruction is

8D 18 D0

since \$8D is the op code for the STA instruction in this addressing mode. The "18" and the "D0" are the ADL and the ADH of the address \$D018. In the machine-language translation, the address must appear with the least-significant byte of the address first, and the most-significant byte second. The first instruction of our program occupied memory locations with addresses \$C000 and \$C001. Thus, this second instruction begins at the location whose address is \$C002, and it is written

C002 8D 18 D0

Study the second line of the program listed in Example 2-5. We have now hand-assembled the program given in Example 2-4.

Example 2-5. Completed Version of the Program in Example 2-4

Operand						
Address	Instruction	Label	Mnemonic	Field	Comments	
\$C000	A9 17	BEGIN		LDA #\\$17	;	"LDA" in immediate addressing mode.
\$C002	8D 18 D0	END		STA VIC18	;	"STA" in absolute addressing mode.

Notice that in Example 2-5 the address given on each line corresponds to the memory location that contains the op code of the instruction. The op code \$A9 is in location \$C000, while the number \$17 is in the next location, \$C001. Likewise, \$8D is in location \$C002, \$18 is in location \$C003, and the last byte of the program, \$D0, is in location \$C004. Notice that all the bytes of an instruction are given on a single line. Thus, each line of the program will have one, two, or three bytes, depending on the addressing mode of the instruction.

Refer once more to Example 2-5. The assembly-language version of the program is frequently called the *source* program or source code. The sequence of eight-bit codes that make up the machine-language version of the program is called the *object* program or object code.

Although it is quite simple to assemble short programs with pencil and paper, for longer programs it is more convenient to use a computer program called an assembler. We will describe the use of an assembler in the next chapter. The programs in this book will all be assembled with an assembler, and the printed output will be photocopied for the book. Example 2-6 illustrates the output of our assembler when the program in Example 2-4 was assembled. Example 2-6 should be compared with Example 2-5, the hand-assembled version.

Example 2-6. Assembled Version of Example 2-4

10 BEGIN	LDA #\$17	; LOAD A WITH THE NUMBER \$17.	C000 A9 17
11 END	STA VIC18	; STORE IT IN LOCATION VIC18.	C002 8D 18 DD

Notice that in Example 2-6 the hexadecimal listing is placed to the *right* of the assembly-language version. Most assemblers place the hexadecimal listing on the left. Although our format represents a departure from tradition, it will pose no problem. In fact, for instructional purposes it may be an advantage: you can focus your attention on the left and neglect the hexadecimal codes on the right until you wish to load these codes into memory. In reference to the hexadecimal listing, also notice that the "\$" prefix is dropped. Observe that each instruction has a line number on the extreme left of the listing. Line numbers will be useful to identify certain instructions when they are discussed within the text of the book. Finally, observe that the column headings are dropped.

One way to load the program in Example 2-6 into memory and execute it is to convert the instruction codes to decimal, POKE the codes into memory using BASIC, and then execute the program as a subroutine using the BASIC SYS instruction. To do that, we must end the machine-language program with a return from subroutine (RTS) instruction whose op code is \$60 (decimal 96). The program in Example 2-7 illustrates this somewhat awkward way of loading and running the machine-language program given in Example 2-6.

Example 2-7. A BASIC Program to Load and Run the Program in Example 2-6

```

5 REM EXAMPLE 2-7
10 RESTORE : ADDR=49152
20 FOR I=0 TO 5 : READ CODE
30 POKE ADDR + I, CODE : NEXT
40 DATA 169,23,141,24,208,96
50 SYS 49152
60 END

```

The program in Example 2-7 READs (line 20) the machine-language codes as DATA (line 40) and POKEs these codes into memory (line 30) starting at the location whose address is 49152 (\$C000). The SYS 49152 command on line 50 causes the machine-language program to be executed, and control returns to the BASIC program on line 60. Run the program in Example 2-7. What happens? Do the letters on the screen change from uppercase to lowercase? If so, you have successfully assembled and executed your first assembly-language program.

V. Loading and Executing Simple Programs

By now you realize that some tools to write, edit, assemble, load, and execute programs would be desirable. The program in Example 2-7 is an awkward way to load and execute a program. It required the translation of all hexadecimal codes into decimal for the DATA statement on line 40.

The most efficient approach to writing assembly-language programs is to use a software package called an *editor/assembler*. The editor allows you to write and edit assembly-language programs. The assembly-language program can then be saved on a cassette tape or, preferably, a disk. The assembler reads the assembly-language program and converts it to eight-bit binary codes that can also be saved on tape or disk. In its binary form, the program can be read and executed by the computer. The assembler also prints the machine-language version in hexadecimal to make it easy for human beings to read. In addition, some editor/assemblers come with a *monitor* program. This is simply a program that allows you to read hexadecimal codes from any location in memory, enter hexadecimal numbers into memory, execute machine-language programs from the monitor, and perform a number of other tasks.

Acquiring an editor/assembler can be expensive, and to use the editor/assembler you must learn its commands and directives, which can be time consuming. Since you will want to try some of the programs listed in the next several chapters and some of your own programs without waiting to purchase an editor/assembler, we will provide some simple tools to allow you to proceed.

You can easily assemble programs using pencil and paper. A programming form similar to the one in Figure 2-1 is useful. Begin by entering the assembly-language program starting in the label column and ending in the comments

Figure 2-1. An assembly-language programming form with the program in Example 2-6 included.

column. Next, use the instruction set summarized in Table 2-1 to convert the instructions into hexadecimal codes and enter them in the left-hand columns. We have illustrated the code for the program in Example 2-6, and we have included an RTS instruction at the end. The RTS instruction ensures that the machine-language program will return to the BASIC calling program.

To load and execute machine-language programs, a BASIC program that accepts hexadecimal codes from the keyboard and POKEs them into memory is given in Example 2-8. During the next few chapters this will be a very useful program, so load it into memory and then save it on tape or disk using a

SAVE "EXAMPLE 2-8"

command for tape storage and a

SAVE "@0:EXAMPLE 2-8",8

command for disk storage.

Example 2-8. A Program to POKE Hex Codes Into Memory

```

5 REM EXAMPLE 2-8
10 PRINT "INPUT THE ADDRESS IN 4 HEX DIGITS."
20 GOSUB100:ADDR=NUM:K=0
30 PRINT "INPUT THE PROGRAM CODES IN HEXADECIMAL."
50 GOSUB 100
60 IF NUM = 96 THEN 80
70 POKE ADDR + K, NUM : PRINT X$,NUM : K = K+1
75 PRINT ADDR+K-1,K-1,NUM:GOTO 50
80 POKE ADDR + K,96
90 PRINT ADDR+K,K,96 : END
100 NUM = 0 : INPUT X$ : I = LEN(X$)
110 FOR J = 1 TO I
120 A$ = MID$(X$,J,1) : Z = ASC(A$)
130 IF Z > 64 THEN Z = Z - 55
140 IF Z > 15 THEN Z = Z - 48
150 NUM = NUM + Z*16↑(I - J):NEXT
160 RETURN

```

RUN the program in Example 2-8. Begin by entering the address of the memory location where your program starts. This will frequently be \$C000. Then enter the hexadecimal program codes from your assembled program on the form shown in Figure 2-1. The program in Example 2-8 automatically converts them to decimal for the POKE instruction and stores them in memory starting at the location you specify. The block of memory starting at \$C000 (49152) is an ideal place to locate machine-language programs.

The last code entered should be the RTS op code \$60. Remember that a machine-language program called from BASIC with a SYS command must end with an RTS instruction. Entering the \$60 op code will conclude the program.

To run the machine-language program enter the SYS xxxx command, where xxxx is the starting address, in decimal, of your machine-language program.

Suppose you want to load and execute the program in Example 2-6. Begin by loading and running the program in Example 2-8. First enter the starting address of the machine-language program, \$C000. Then you will enter each

hexadecimal code in Example 2-6 in the order A9 17 8D 18 D0, and conclude by entering 60, the RTS op code. Press RETURN after each of the six hexadecimal codes. When finished, type SYS 49152 and the letters on the screen will change from uppercase to lowercase when the machine-language program is executed. Try it.

We conclude this section with several additional programs that will be useful to you. The program in Example 2-9 prints the hexadecimal codes found in any block of memory. You can use this program to see whether you have correctly entered the hexadecimal codes of a machine-language program. The starting and ending address of the block of memory you wish to examine are entered in hexadecimal. The program then prints the code in each memory location as two hexadecimal digits. The output is formatted to give 16 columns. See Table 4-5 for an example of the output.

Example 2-9. A Program to Print the Hexadecimal Codes in a Block of Memory

```
5 REM EXAMPLE 2-9
10 PRINT "INPUT THE BEGINNING ADDRESS IN HEX."
20 GOSUB 100:A1 = NUM:K=0
30 PRINT "INPUT THE ENDING ADDRESS IN HEX."
40 GOSUB 100:A2 = NUM:K=0
50 GOTO 200
100 NUM=0:INPUT X$:I=LEN(X$)
110 FOR J=1 TO I
120 A$=MIDS(X$,J,1):Z=ASC(A$)
130 IF Z>64 THEN Z=Z-55
140 IF Z>15 THEN Z=Z-48
150 NUM=NUM+Z*16↑(I-J):NEXT
160 RETURN
200 OPEN 1,4
210 J=0
220 FOR I=A1 TO A2
230 Y=PEEK(I)
240 X=INT(Y/16):Z=(Y/16-X)*16:Z=INT(Z)
250 IF X<10 THEN X=X+48: GO TO 270
260 X=X+55
270 IF Z<10 THEN Z=Z+48 :GO TO 290
280 Z=Z+55
290 F$=CHR$(X)+CHR$(Z)
300 PRINT#1,F$SPC(1);
310 J=J+1:IF J=16 THEN PRINT#1," ":J=0
320 NEXT
330 PRINT#1," "
340 END
```

The program in Example 2-10 allows you to save a block of memory on either tape or disk. This program is used to save your machine-language programs once they have been assembled and debugged. You load and RUN the program in Example 2-10, and begin by entering a name for the file that will store the hexadecimal codes of your machine-language program. It will be useful to keep your file names meaningful. A file name of "EXAMPLE 2-6" is more meaningful than "AZ1L%Q." Next, inform the program that you are going

to use either DISK or TAPE by entering "D" or "T," respectively. Finally, enter the starting and ending address of your machine-language program, and it will store the codes.

Of course, you will need a means to retrieve this information and place it in the same memory locations from which it was obtained. The program in Example 2-11 accomplishes this task. Load and RUN Example 2-11. Begin by specifying the magnetic storage medium you are using, namely, "D" or "T." Then enter the name of the file you wish to place into memory, and the program does the rest for you. It will be convenient to store both of these programs on disk or tape. They will be particularly useful to you until you purchase and learn to use an editor/assembler.

Example 2-10. A Program to Save Codes in Memory on Tape or Disk

```
5 REM EXAMPLE 2-10:BINARY SAVE
10 INPUT "THE FILE NAME IS ";F$
11 PRINT "DISK (D) OR TAPE (T)?"
12 INPUT A$
13 IF A$ = "D" THEN 20
14 IF A$ = "T" THEN 200
15 GO TO 11
20 OPEN 15,8,15
30 INPUT#15,A,B$,C,D.
40 IF A THEN PRINT A,B$,C,D:CLOSE 15:STOP
50 PRINT "INPUT THE BEGINNING ADDRESS IN HEX."
60 GOSUB 1000:A1=NUM:K=0
70 PRINT "INPUT THE ENDING ADDRESS IN HEX."
80 GOSUB 1000:A2=NUM:K=0
100 OPEN2,8,2,"@0:"+F$+",S,W"
110 PRINT#2,A1
111 PRINT#2,A2
120 FOR I=A1 TO A2
130 Z=PEEK(I)
140 PRINT#2,Z
150 NEXT
160 CLOSE 2:CLOSE 15
170 END
200 REM SAVE DATA ON DATASSETTE.
210 PRINT "INPUT THE BEGINNING ADDRESS IN HEX."
220 GOSUB 1000:A1=NUM:K=0
230 PRINT "INPUT THE ENDING ADDRESS IN HEX."
240 GOSUB 1000:A2=NUM:K=0
250 OPEN 1,1,1,"+F$"
260 PRINT#1,A1
270 PRINT#1,A2
280 FOR I=A1 TO A2
290 Z=PEEK(I)
300 PRINT#1,Z
310 NEXT
320 CLOSE 1
330 END
1000 NUM=0:INPUT X$:I=LEN(X$)
1100 FOR J=1 TO I
1200 A$=MIDS(X$,J,1):Z=ASC(A$)
1300 IF Z>64 THEN Z=Z-55
1400 IF Z>15 THEN Z=Z-48
1500 NUM=NUM+Z*16↑(I-J):NEXT
1600 RETURN
```

Example 2-11. A Program to Load Codes From Tape or Disk into Memory

```
5 REM EXAMPLE 2-11: BINARY LOAD
10 INPUT "THE FILE NAME IS ",F$
11 PRINT "TAPE (T) OR DISK (D)"
12 INPUT A$
13 IF A$ = "D" THEN 20
14 IF A$ = "T" THEN 200
15 GO TO 11
20 OPEN 15,8,15
30 INPUT#15,A,B$,C,D
40 IF A THEN PRINT A,B$,C,D:CLOSE 15:STOP
100 OPEN 2,8,2,"@0:"+F$+",S,R"
110 INPUT#2,A1
111 INPUT#2,A2
115 PRINT A1,A2
120 FOR I=A1 TO A2
130 INPUT#2,Z :PRINT Z:POKEI,Z
150 NEXT
160 CLOSE 2:CLOSE 15
170 END
200 REM "LOAD FROM TAPE"
210 OPEN 1,1,0,"+F$"
220 INPUT#1,A1
230 INPUT#1,A2
240 PRINT A1,A2
250 FOR I=A1 TO A2
260 INPUT#1,Z:PRINT Z:POKEI,Z
270 NEXT
280 CLOSE 1
300 END
```

VI. Summary

An assembly-language program consists of labels for program addresses, instruction mnemonics from Table 2-1, and symbols for locations of operands. Comments may also be included. An assembly-language program is not executed. It must be translated into machine language, and then the machine-language program is executed.

Once the assembly-language program has been written on the form shown in Figure 2-1, it is translated to hexadecimal codes using the instruction set summarized in Table 2-1 and the addresses assigned to the symbols in the assembly-language program. If you have an editor/assembler, you will type the program on the keyboard and see it on your screen. Also, the assembler will translate the program into machine language and save it on disk or store it in memory. Once in memory, the machine-language program can be called as a subroutine from a BASIC program with a SYS xxxx command, where xxxx is the starting address of the machine-language program expressed in decimal. Machine-language programs called in this way should end with the \$60 op code for the RTS instruction.

VII. Exercises

1. Use the op codes in Table 2-1 and these memory assignments:

ADD1 is \$C234

ADD2 is \$C235

SUMM is \$CF00

to assemble this group of instructions:

LDA ADD1 ;Copy the number in ADD1 into the accumulator A.

ADC ADD2 ;Add the number in ADD2 to the number in A.

STA SUMM ;Store the sum in SUMM.

Use the sample programming form in Figure 2-1. The answer can be found in Table 1-2, since this is the same program as the simple addition program described in Chapter 1.

2. Decide to start the program described in the previous exercise at \$C000. Use the program in Example 2-8 to enter this address, and load the hexadecimal codes into memory. You will load, in order, AD 34 C2 6D 35 C2 8D 00 CF, and then end by entering the RTS op code 60.
3. To provide some data for the program, perform these two BASIC commands:

POKE 49716,3

POKE 49717,4

49716 is \$C234 and 49717 is \$C235. Thus, the program with which we are working will add three and four. Now execute the machine-language program with a SYS 49152 command. Then use a

PRINT PEEK(52992)

command from BASIC to see what sum was stored in location \$CF00 (52992). You should find the answer seven. Do not be surprised if you obtain eight for an answer. This may happen since it is possible for the 6510 microprocessor to have a carry in it. More about this in Chapter 4.

4. Use the BASIC program in Example 2-10 to store your machine-language program. Use ADD for the file name, a starting address of \$C000, and an ending address of \$C009.
5. Turn off the computer. Turn it on again, and then attempt to retrieve your program from the magnetic medium you are using. Use the program in Example 2-11 to do this. Enter the file name ADD.
6. Finally, use the program in Example 2-9 to display the numbers in the memory locations you have just filled to see that the correct hexadecimal codes have been loaded back into memory. RUN Example 2-9; entering \$C000 for the starting address and \$C009 for the ending address. Compare the codes that are printed with the program codes given above or in Table 1-2.

3

Data Transfer Instructions— Using an Editor/Assembler

I. Introduction

The instruction set of the 6510 microprocessor can be divided into several subsets, as in Table 1-1, which provides the English-language descriptions of the 6510 instructions. The easiest instructions to understand are those found in the group identified in Table 1-1 as Data Transfer Instructions. The data transfer instructions you will learn in this chapter include the LDA, STA, LDX, STX, LDY, STY, TAX, TXA, TAY, and TYA instructions. The op codes for the addressing modes we will use in this chapter are shown in Table 3-1.

Table 3-1. Op codes for the instructions used in Chapter 3.

<i>Mnemonic</i>	<i>Op Codes for Each Addressing Mode</i>				
	<i>Logical Description</i>	<i>Immediate</i>	<i>Absolute</i>	<i>Zero-page</i>	<i>Implied</i>
LDA	A→M	\$A9	\$AD	\$A5	
STA	M→A		8D	85	
LDX	M→X	A2	AE	A6	
STX	X→M		8E	86	
LDY	M→Y	A0	AC	A4	
STY	Y→M		8C	84	
TAX	A→X				AA
TXA	X→A				8A
TAY	A→Y				98
TYA	Y→A				A8

The logical descriptions of each instruction are also given in Table 3-1. Think of the arrow in the logical description column as implying the *transfer* of an eight-bit code. The data transfer instructions do not modify numbers or codes in any way: they transfer codes either from the microprocessor to memory,

from memory to the microprocessor, or from one register to another in the microprocessor. You have already been introduced to the LDA and STA instructions, so you are on familiar ground.

Although it is possible to write short assembly-language programs without an editor/assembler, for long programs it is more convenient to use one. You will also be introduced to an editor/assembler in this chapter. The sections in this chapter related to the editor/assembler were included to give you a feeling for the process of using an editor/assembler on the computer and for its features. We used the disk-based Develop-64 system which is part of a package called *The Machine Shop*, available from:

FSI Software
P.O. Box 7096
Minneapolis, MN 55407
(800) 328-0145

Our choice should be interpreted neither as an endorsement of this product nor a criticism of the editor/assemblers we did not use. Perhaps it would be best for you to start with a very inexpensive assembler, such as Commodore's own assembler. Consult your Commodore dealer for details. A cartridge-based assembler called MIKRO 64 is available from

Skyles Electric Works
231 E. South Whisman Road
Mountain View, CA 94041

More advanced programmers would probably choose the MAE (Macro Assembler Editor) carried by several dealers and mail-order firms, or the PAL, available from:

Pro-Line Software, Ltd.
755 The Queensway East, Unit 8
Mississauga, Ontario L4Y-4C5

Once again, my advice to beginning programmers is to assemble programs using Table 2-1, a pencil, and a programming form similar to the one in Figure 2-1. The hexadecimal codes can be loaded into memory with the simple program in Example 2-8, and programs can be executed from BASIC with the SYS command. *Do not purchase an assembler until you are making satisfactory progress in your study of assembly language, perhaps finishing Chapter 6.* At that point, you will know if you are seriously or just mildly interested in assembly-language programming. Only if you are seriously interested should you make the considerable investment in an editor/assembler. By the time you have mastered the first half of this book, you will also know enough about assembly language to make a wiser choice than you can make at this point in your programming career. Meanwhile, continue to study advertisements and reviews of assemblers that appear in computer magazines. Experienced programmers in Commodore 64 user groups can frequently be of help in choosing an assembler. Economics frequently dictate the choice of an assembler. Other important considerations are the quality of the documentation that accompanies the assembler software and the speed and ease with which the assembler works.

II. Data Transfer Instructions: Load and Store

The LDA and STA instructions were diagrammed in Figures 1-5 and 1-6. Once again, refer to those figures for a moment. When the LDA instruction is executed, an eight-bit code is copied from memory into the accumulator. When the STA instruction is executed, the eight-bit code in the accumulator is transferred to a memory location. Recall that we compared the LDA instruction to a PEEK command in BASIC, and we compared the STA instruction to a POKE command in BASIC.

The LDX and STX instructions are identical to the LDA and STA instructions, except that the X register rather than the accumulator is used. If the X register is substituted for the accumulator in Figure 1-5, it becomes a diagram of the LDX instruction. Likewise, if the X register is substituted for the accumulator in Figure 1-6, it becomes a diagram of the STX instruction. Two similar statements can be made for the LDY and STY instructions.

It was probably obvious that the LDX and the LDY instructions work in the same way as the LDA instruction. Likewise, the STX and STY instructions work in the same way as the STA instruction. Thus, it is not surprising that the three "loads" have similar addressing modes and the three "stores" have similar addressing modes. Notice from the instruction set in Table 2-1 that the LDA, LDX, and LDY instructions use the immediate, absolute, and zero-page addressing modes. Also notice that the STA, STX, and STY instructions use the absolute and zero-page addressing modes.

Use of the LDA and STA instructions in a program was illustrated in several examples in Chapter 2. Example 3-1 illustrates the LDX and STX instructions in a data transfer operation. The LDX instruction is used in the immediate addressing mode, and in that mode it is a two-byte instruction. The STX instruction is used in the absolute addressing mode, and in that mode it is a three-byte instruction. The program stores the number seven in the \$21st register of the VIC chip. The contents of this register determine the video background color. After executing the program, the background should be yellow.

The purpose of our examples is to illustrate instructions and addressing modes rather than to produce eye-catching results. On the other hand, it is important to have the program do something so that you can tell if it executed correctly. It is also useful to show some real applications rather than some artificially constructed programs. Try running the program using the technique provided in Example 2-8. Notice that an RTS instruction has been included in Example 3-1 to ensure the proper operation of Example 2-8. Many of our sample programs will end in this way.

Example 3-1. Using the X Register for a Data Transfer

Object: Change the video background color to yellow.

10	START	LDX #\$07	;LDX IN IMMEDIATE ADDRESSING MODE.	C000	A2	07
11	STOP	STX VIC21	;STX IN ABSOLUTE ADDRESSING MODE.	C002	8E	21
12		RTS	;RETURN TO BASIC INTERPRETER.	C005	60	

The program in Example 3-2 illustrates the LDY and STY instructions. Its function is to set the border color to blue. "VIC20" is a symbol for the \$20th register in the VIC chip, a register that controls the border color. The addressing modes are identical to those used in Example 3-1.

Example 3-2. Using the Y Register for a Data Transfer

Object: Change the color of the video border to blue.

10 BEGIN	LDY #\$06	; LDY IN IMMEDIATE ADDRESSING MODE.	C000 A0 06
11	STY VIC20	; STY IN ABSOLUTE ADDRESSING MODE.	C002 8C 20 D0
12	RTS	; RTS USES IMPLIED ADDRESSING MODE.	C005 60

We have not yet illustrated the zero-page addressing mode. The next two examples use this mode. Any instruction that uses the zero-page addressing mode will be a two-byte instruction, whereas the absolute addressing mode requires three bytes. Refer to the instruction set summary in Table 2-1 and observe that the absolute addressing mode requires one more clock cycle to execute the instruction than the zero-page addressing mode. Thus, the zero-page addressing mode is more efficient in terms of both program space and execution time. Notice that the editor/assembler used to write the programs in Examples 3-3 and 3-4 prefixes the symbol for a zero-page memory location with a plus sign (+). The plus sign informs the assembler that the zero-page addressing mode is desired.

The program in Example 3-3 places a graphics character for a ball on the screen, obtains a color code for the ball from location \$0002, and stores the color code in the color memory. The color code is fetched from memory using the LDY instruction in the zero-page addressing mode. If you want to try the program, POKE a color code from 0 to 15 into location \$0002 before running the program. The colored ball will appear in the lower right-hand corner of the screen. When you return to BASIC, the ball may scroll upward.

Example 3-3. Using the LDY Instruction in the Zero-Page Addressing Mode

Object: Place a ball on the screen and color it.

10 START	LDY +COLOR	; FETCH THE BALL COLOR.	C000 A4 02
11	LDA #\$51	; \$51 IS SCREEN CODE FOR A BALL.	C002 A9 51
12	STA SCRMEM	; STORE CODE IN SCREEN MEMORY.	C004 8D E7 07
13	STY COLMEM	; PLACE COLOR CODE IN MEMORY.	C007 8C E7 DB
14	RTS		C00A 60

The next programming example illustrates the STA instruction in its zero-page addressing mode (refer to Example 3-4). This program writes a number to the output port at location \$0001. The effect of the program is to switch an 8K bank of R/W memory into the address space occupied by the 8K BASIC interpreter. Be prepared to lose any BASIC program you have in memory when you execute this program because you will have to switch off the Commodore 64 and turn it on again to get back into BASIC.

Why would you want to replace the interpreter with R/W memory? If, for example, a machine-language word processor is being executed, then you have no need for the BASIC interpreter, but you do need a great deal of R/W memory to store either the program or the data. In that case, it is useful to be able to replace the BASIC ROM with R/W memory.

Example 3-4. Using the STA Instruction in the Zero-Page Addressing Mode

Object: Replace the BASIC interpreter with R/W memory.

10 ORIGIN LDA #\$01 ;LDA IN IMMEDIATE ADDRESSING MODE.	C000 A9 01
11 STA +PORT ;STA IN ZERO PAGE ADDRESSING MODE.	C002 85 01
12 RTS	C004 60

Often a program begins by storing certain numbers in various memory locations in preparation for the program that follows. The sequence of instructions that handles these preparatory steps is called an *initialization* routine. Our last two examples in this section consist of short initialization routines, one for Voice #1 in the SID chip and one for Sprite #0 in the VIC chip. The SID and the VIC will be described in more detail in Chapters 10 and 11, respectively. The programming examples here will provide additional illustrations of data transfer instructions.

Complex chips like the VIC and SID frequently require that a series of registers in the chip contain specific values before proceeding. Data transfer instructions are used to store these values in the registers. Example 3-5 illustrates some of the steps required to initialize the SID chip to start "playing" a ramp waveform from Voice #1. Note the use of the LDA and STA instructions.

Before executing the program, you should clear the SID registers by running this BASIC program:

10 FOR I = 54272 TO 54296 : POKE I, 0 : NEXT

We will learn how to do this in assembly language in Chapter 8.

If your TV or video monitor lacks sound, you may have to connect the Commodore 64 audio output to an input in your audio system. Execute the program and you will hear a tone.

Example 3-5. Initializing SID Registers

Object: Initialize the SID so that Voice #1 plays a tone using the ramp waveform.

10 INITLZ LDA #\$0F ;\$0F CORRESPONDS TO MAXIMUM VOLUME.	C000 A9 0F
11 STA SID18 ;SET THE VOLUME ON SID.	C002 8D 18 D4
12 LDA #\$19 ;CHOOSE A NUMBER FOR THE HIGH BYTE	C005 A9 19
13 STA SID01 ;OF THE FREQUENCY FOR VOICE #1.	C007 8D 01 D4
14 LDA #\$EE ;SET THE ATTACK/DECAY RATE	C00A A9 EE
15 STA SID05 ;FOR VOICE #1 ON SID.	C00C 8D 05 D4
16 LDA #\$A0 ;SET THE SUSTAIN LEVEL FOR VOICE #1	C00F A9 A0

17	STA SID06	;ON SID.	C011 8D 06 D4
18	LDA #\$21	;SELECT A RAMP WAVEFORM FOR VOICE #1	C014 A9 21
19 END	STA SID04	;AND START THE SOUND.	C016 8D 04 D4
20	RTS		C019 60

Example 3-6 illustrates the initialization procedure required to place a sprite on the screen. In this case, Sprite #0 is given a blue color on a yellow background with a blue boundary. You should compare this routine with the BASIC routine on page 71 of the *Commodore 64 User's Guide* that comes with your computer.

Example 3-6. Using Data Transfer Instructions to Initialize a Sprite

Object: Initialize Sprite #0.

10	;		
11 ORG	LDA #\$0D	;SPRITE #0 MEMORY POINTER.	C000 A9 0D
12	STA \$07F8		C002 8D F8 07
13	LDA #\$01	;TURN ON SPRITE #0 ON THE VIC.	C005 A9 01
14	STA VIC15		C007 8D 15 D0
15	LDA #\$07	;SELECT A YELLOW BACKGROUND.	C00A A9 07
16	STA VIC21		C00C 8D 21 D0
17	LDA #\$06	;SELECT BLUE FOR THE SPRITE.	C00F A9 06
18	STA VIC27		C011 8D 27 D0
19	STA VIC20	;SELECT A BLUE BORDER COLOR.	C014 8D 20 D0
20	LDX #\$80	;\$80 = 128 WILL BE THE X COORDINATE	C017 A2 80
21	STX VIC00	;OF SPRITE #0.	C019 8E 00 D0
22	LDY #\$80	;CHOOSE 128 FOR THE Y COORDINATE	C01C A0 80
23	STY VIC01	;OF SPRITE #0.	C01E 8C 01 D0
24	RTS		C021 60

A BASIC routine to define a rectangular dot matrix for the sprite in the program in Example 3-6 is given in Example 3-7. An efficient way to accomplish the same objective in assembly language must be postponed until Chapter 8. RUN the program in Example 3-7 before executing the program in Example 3-6.

Example 3-7. A BASIC Program to Define a Rectangular Sprite

```

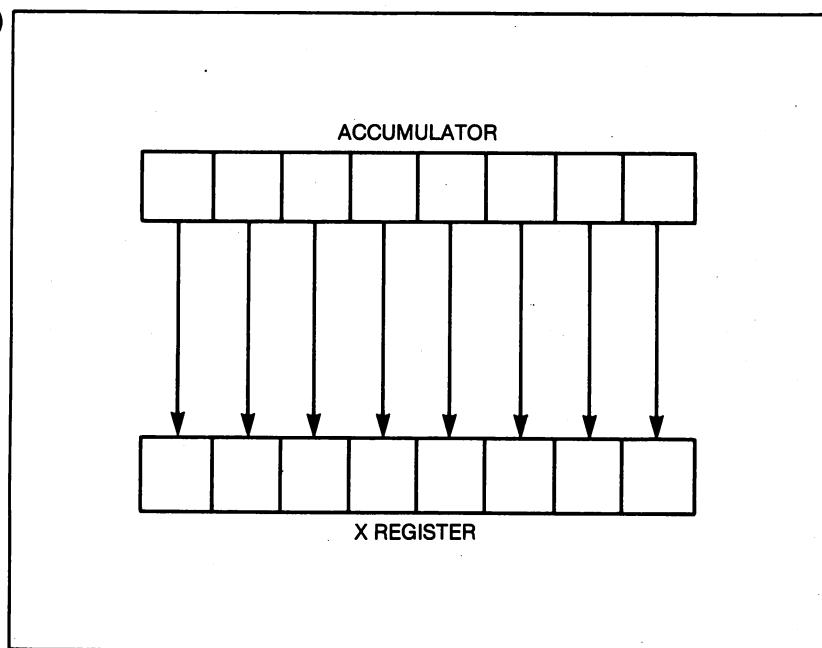
0 REM EXAMPLE 3-7
1 ADDR = 832
2 FOR I = 0 TO 62
3 POKE ADDR + I,255
4 NEXT

```

III. Data Transfer Instructions: Register Transfers

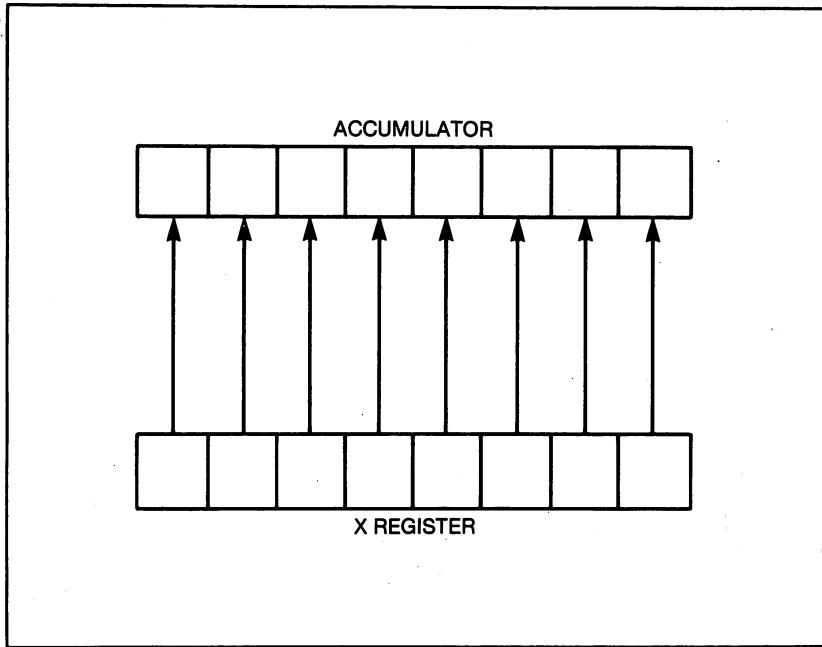
We turn next to the TAX, TXA, TAY, and TYA instructions. Referring to Table 3-1, note that these instructions use a single addressing mode, namely, *implied addressing*. Both the source and the destination are implied by the instruction. A diagram of the TAX and TXA instructions is shown in Figure 3-1. It should be simple to imagine corresponding diagrams for the TAY and TYA instructions.

(a)



6510 MICROPROCESSOR

(b)



6510 MICROPROCESSOR

Figure 3-1. (a) Diagram of the TAX instruction. (b) Diagram of the TXA instruction.

Certain programs require an initialization sequence of instructions that clears all three of these registers. *Clearing* a register means placing a zero in each of the eight bits. What is the most efficient way to do this? A candidate for the optimum way of clearing A, X, and Y is illustrated with Example 3-8.

Example 3-8. A Program to Clear A, X, and Y

Object: Clear all the bits in A, X, and Y to zero.

10	ORIGIN	LDY #\\$00	;CLEAR THE Y REGISTER.	C000 A0 00
11		TYA	;IMPLIED ADDRESSING MODE.	C002 98
12		TAX	;IMPLIED ADDRESSING MODE.	C003 AA
13		RTS		C004 60

Consult the instruction set summarized in Table 2-1 and verify that the program in Example 3-8 takes four bytes of program memory and will execute in six clock cycles. Can you write a program to do the same thing in less time or with fewer bytes of program memory?

Although this concludes the sample programs given in this chapter, we will give several programming problems at the end of the chapter. We turn next to the operation of an editor/assembler.

IV. Writing and Assembling Programs*

Now that you have mastered several instructions in the 6510 instruction set, you might be interested in seeing how the programming process proceeds with an editor/assembler. The *editor* part of an editor/assembler allows you to use the computer to write an assembly-language program. The assembler translates the assembly-language program you have written with the editor into the kind of program listings you have seen in Examples 3-1 to 3-9. In addition to translating assembly language into machine language, the assembler may also load the machine-language codes into memory where the program is ready to be executed. An editor/assembler software package is sometimes called a *development system*.

Of course, you may choose to purchase an editor/assembler other than the Develop-64 system that we will describe. In that case, this section will not be as useful, but you might browse through it simply to see how this particular editor/assembler works. At the very least, you might become aware of features you want to find in the editor/assembler that you decide to purchase.

Our remarks apply to the disk-based Develop-64 portion of The Machine Shop. We will use the program in Example 3-3 to describe the procedure of editing and assembling a program. The Develop-64 system comes with extensive documentation, so we will describe only the most important features.

*The beginning programmer may find this and future sections marked with an asterisk to be more difficult. Skip these if that is the case.

A. Using the editor

Place the Develop-64 disk in the disk drive and type

```
LOAD "D",8,1
```

Wait for the following message to appear on the screen:

```
DEVELOP-64 OVERWRITES THE CONTENTS OF A BLOCK OF MEMORY. CHOOSE ONE:
```

- 1) \$0800-\$47FF (2048-18431)
- 2) \$6000-\$9FFF (24414-40959)

PRESS 1 or 2:

To begin, press 1, and the main menu, shown below, will appear:

- 1) EXIT
- 2) EDIT
- 3) ASSEM
- 4) DECODE
- 5) DEBUG
- 6) LOAD
- 7) SAVE
- 8) CLEAR
- 9) CONFIG
- 0) DISK

Select the EDIT option by pressing the 2 key. The screen will clear and a menu will appear in the upper left hand corner with a flashing cursor to the right of the colon prompt symbol. This will be line 1 in your assembly-language program. A program line is terminated by pressing the "RETURN" key on the Commodore 64. Hereafter, we will not specify RETURN. The line number "1", and all subsequent line numbers, are automatically assigned.

Documenting your program with a name is good practice. For line 1 enter

```
;EXAMPLE 3-3
```

as your program name. The semicolon (;) prefix on a line indicates that the line contains a comment. This is line 1 of your program. Similar to the situation of REM lines in BASIC, comment lines are ignored during the assembly process.

After you press RETURN, you will get a prompt for line 2, and the flashing cursor will reappear.

Examine the program in Example 3-3. We used the symbol "COLOR" for the address \$0002. This is accomplished with the "EQU" assembler directive. A directive is an instruction to the assembler: it is not an assembly-language instruction. To identify COLOR with the address \$0002, type

```
COLOR EQU$0002
```

following the colon prompt. With the Develop-64 editor you leave a space between "COLOR" and "EQU" but you do not put a space between "EQU" and the "\$0002". Line 2 is now completed.

Two other symbols, COLMEM and SCRMEM, must still be defined. Enter two more lines as follows:

```
COLMEM EQU$DBE7  
SCRMEM EQU$07E7
```

which become lines 3 and 4 of your assembly-language program.

Next, we will define the address of the location that stores the first program byte. Following the colon prompt, enter line 5 with a leading space followed by EQU\$C000. It should look like this on screen:

```
5: EQU$C000
```

We have now determined that the program will begin at memory location \$C000.

Notice that all the work that we have done so far has not yet generated one line with a 6510 instruction. We have just taken care of the necessary preliminary steps. It will be easier to separate the preliminary lines from the program lines if we insert several empty lines. Following the colon prompt, type a semicolon (;) and press RETURN. This becomes line 6. Add several more lines with leading semicolons.

Now we are ready to enter the program. Following the quote prompt, enter the line

```
START LDY←COLOR ;GET COLOR CODE.
```

Notice once again that there is a space between the label "START" and the mnemonic "LDY", but there is no space between the LDY mnemonic and the symbol " \leftarrow COLOR". The left-arrow symbol is used by this assembler to indicate that the zero-page addressing mode is being requested by the programmer.

Now enter, in order, the following four program lines to complete the editing task:

```
LDA#$51 ;SCREEN CODE FOR BALL.  
STASCRNMEM ;STORE CODE IN SCREEN MEMORY.  
STYCOLMEM ;COLOR CODE INTO COLOR MEMORY.  
RTS
```

A space between the quote prompt and the mnemonic is absolutely necessary. The space indicates that the label field is empty. No spaces are typed between the mnemonics and information in the operand field. The editor will insert these spaces later.

You have now finished editing the program. After entering the last line, press the RETURN key once again.

While in the EDIT mode, you can use various commands outlined in the Develop-64 documentation to *modify* lines, *list* lines, or *insert* additional lines. This is similar to the editing that the Commodore 64 provides for BASIC programs, and it is a very important feature of an editor.

You will want to save your program, so select the SAVE option, item 7. When asked to choose between source (S) or binary (B), choose the default source option since you are saving your source code. Then, upon request, enter the file name, EXAMPLE 3-3. The disk drive will start and your program will be saved. The main menu will return to the screen.

B. Using the assembler

Select the 3) ASSEM option from the main menu if the source program is still in memory; otherwise, select the 6) LOAD option. If you select the load option, select the EXAMPLE 3-3 source file, load it into memory, and then select the 3) ASSEM option from the main menu. You are now in the assembler.

After selecting the ASSEM option, you will see a sub-menu with the options 1) EXIT, 2) ASSEMBLE, or 3) SYMBOL. Choose the 2) ASSEMBLE option. You will be asked if you wish to

ASSEMBLE TO MEMORY?

Enter a "Y" for : "YES". This causes your machine-language program to be stored in memory, where it may be executed later from BASIC with a SYS 49152 command.

The next prompt,

CREATE OBJ/PRG/NEITHER

should be answered with a "O" for OBJ, causing your machine-language program to be stored as a sequential file on disk. A prompt will ask you for the name of the file. Enter "EXAMPLE 3-3."

The next prompt will be

SCREEN PRINTER, OR ERRORS ONLY?

If you have a printer, give a "P" response. Otherwise use the "S" response. Your program will now be assembled, and the machine-language (object) program will be stored in memory and on the disk as a sequential file.

The entire output is shown in Example 3-9. (This output is from an earlier version of the Development-64 assembler. The latest version places the hexa-decimal listing on the left.) It should be clear that in previous examples we have suppressed the "EQUATE" section for the sake of simplicity. In the examples that follow, we will continue to suppress the "EQUATE" section for simple programs in order to simplify the listings. Also, observe that my printer replaces the Commodore 64 left-arrow (zero-page addressing mode) symbols with plus signs (+).

Example 3-9. The Output of the Assembler for Example 3-3

```

1      ;EXAMPLE 3-9
2 COLOR  EQU $02          ;COLOR NUMBER STORAGE.           0002
3 COLMEM EQU $DBE7        DBE7
4 SCRMEM EQU $07E7        07E7
5 START  EQU $C000        C000
6      ;
7      ;
8      ;
9      ;
10 START LDY +COLOR      ;FETCH THE BALL COLOR.       C000 A4 02
11      LDA #$51          ;$51 IS SCREEN CODE FOR A BALL.   C002 A9 51
12      STA SCRMEM        ;STORE CODE IN SCREEN MEMORY.    C004 8D E7 07
13      STY COLMEM        ;PLACE COLOR CODE IN MEMORY.     C007 8C E7 DB
14      RTS               CO0A 60

```

You can see that it will require some time to become proficient with an editor/assembler. The more powerful the development package, the more time it will take to learn to use the options. This effort may reduce the time you have to learn assembly language and it may complicate those efforts. This is why we recommend that you begin by hand-assembling simple programs.

In any case, it is worth adding that some preliminary steps should be taken before the editing/assembling process begins. Have the program objective clearly in mind. What input is required? What output? Decide what memory locations you will use to store the program. Any locations from \$C000 to \$CFFF are appropriate for the Commodore 64. These locations are reserved for machine-language programs. What memory locations are required to store the parameters used in the program, what are their addresses, and what symbols will you use to represent them? What SID, VIC, or CIA registers do you need to reference, and what symbols will you use? Write this information on paper.

Complex programs may be subdivided into smaller modules and these may require flowcharts. Before writing the actual assembly-language program, try writing your program in *pseudo-code*, a combination of English statements and assembly-language mnemonics. The main point is to express, in some kind of language, what you want the program to do *before* you begin writing the assembly-language version. The importance of these preliminary steps cannot be overemphasized if efficient programming is your objective.

C. Executing the machine-language program

You have assembled your program and you are ready for the moment of truth: does it work? If the assembler stored the program into memory starting at location 49152 (\$C000), then after leaving the Develop-64 program, you can execute it by typing SYS 49152. Almost all of our examples produce an observable effect, so you will know if the execution was successful. You can also use PEEK instructions to see whether the memory locations modified by the program were, in fact, modified.

If you did not have the assembler store the program into memory, then you must load the object program from disk or tape. A program that first loads Develop-64 object code from the disk and then executes the program is listed in Example 3-10. Notice that you must change program line 10 to select the machine-language (object) file you wish to execute. In our case, we selected the program we assembled above, namely, EXAMPLE 3-3. OBJ. Try it.

Example 3-10. A Program to Load and Execute a Machine-Language Routine

```
5 REM EXAMPLE 3-10
10 OPEN 1,8,2,"0:EXAMPLE 3-3.OBJ,S"
20 CLOSE 15:OPEN 15,8,15
30 INPUT#15,A,B$,C,D:IF A THEN PRINT A,B$,C,D:CLOSE1:CLOSE15:STOP
```

```
40 INPUT#1,N:IFN==1000 THEN CLOSE1:GOTO80
50 IF N<1THEN POKEP,-N:P=P+1:GOTO 40
60 P=N :GOTO 40
80 SYS 49152
90 PRINT "MISSION ACCOMPLISHED."
100 END
```

D. Debugging programs

Many development packages will execute a program in a *single-step* mode. In this mode, one instruction is executed at a time. After each instruction is executed, the codes in each of the registers in the 6510 are displayed on the screen. This is a useful technique to debug programs because you can see the effect of any instruction, and you can observe your program doing its thing much more slowly—in fact, as slowly as you wish.

Use the single-step mode of your development package to execute the program in Example 3-3. In the Develop-64 system, this is the DEBUG program. The first instruction in Example 3-3 is a LDY \$02 instruction. Suppose the number \$07 is in location \$0002. Execution of this instruction in the single-step mode will show that the number in the Y register is now seven. Now the next instruction in the program is displayed. This is the LDA#\$51 instruction. After it is executed, the number \$51 (81 in decimal) will be displayed for A. Continue to single-step through the program until you see how the single-step mode works. You can exit this mode by selecting the appropriate exit options in the menu. The single-step mode of operation is an extremely useful tool.

This completes our discussion of the Develop-64 editor/assembler. If and when you decide to seriously pursue assembly-language programming, we strongly recommend that you purchase an editor/assembler development package. We hope that this brief description of one such package will give you a better idea of what features you want in the package you purchase.

V. Summary

The LDA, LDX, and LDY instructions copy an eight-bit code from memory into the A, X, and Y registers, respectively. These three instructions use the immediate, absolute, or zero-page addressing modes. The STA, STX, and STY instructions transfer an eight-bit code from the A, X, and Y registers, respectively, to a location in memory. These three instructions use the absolute and zero-page addressing modes.

On the other hand, the TAX, TXA, TAY, and TYA are used to transfer eight-bit codes from one 6510 register to another. The source register is indicated by the second letter in the instruction and the destination register is indicated by the last letter in the instruction. Since memory is not accessed by these instructions, they use only the implied addressing mode.

Data transfer instructions are used to move information about in the memory of the microcomputer; to initialize registers in I/O chips, such as the VIC, SID, and CIA; and to read and write to input/output ports, such as the one at location \$0001 on the 6510 microprocessor.

Serious assembly-language programmers will eventually require a software package called an editor/assembler to help them develop assembly-language programs. This program development package should provide the capability to write, modify, list, assemble, execute, and debug programs.

VI. Exercises

1. Assemble this program:

```
LDA MEM1  
STA MEM2  
RTS
```

where MEM1 is a symbol for the address \$C010 and MEM2 is a symbol for the address \$0002. Use Example 2-8 to store the object code starting at location \$C000. Use a

POKE 49168,255

command in BASIC to place the number \$FF in location \$C010. Execute the program from BASIC with a

SYS 49152

command. Finally, use a

PRINT PEEK(2)

command to see if the number \$FF was transferred from location MEM1 to location MEM2. Identify the addressing mode used for each assembly-language instruction.

2. Assemble this program:

```
LDY #$00  
STY MEM2  
RTS
```

Follow the procedures outlined in Exercise 1 to load, execute, and test the program. After the program is executed, what number should you find with the PRINT PEEK (2) instruction? Identify the addressing mode used for each instruction.

3. Write a program to transfer the eight-bit code in location \$FFFF to location \$0002. Use the accumulator as the data transfer register. Before executing the program, POKE zero into location \$0002. After execution, PEEK into location \$0002 to see what number is there.
4. Write a program to change the background color to red. Use the X register as the data transfer register. Refer to Example 3-1. Two is the color code for red.

5. Write a program to load A, X, and Y with \$FF. Use the single-step mode to execute this program and observe how the numbers in the registers change as the program is executed. Use the TAX and TAY instructions.
6. If you have an editor/assembler, use it to assemble the program in Example 3-6.
7. Write a program to transfer the hours, minutes, seconds, and tenths of seconds in the time-of-day clock at locations \$DC0B, \$DC0A, \$DC09, and \$DC08, respectively, to locations \$00FB through \$00FE.
8. What three instructions that you learned in this chapter are analogous to the POKE instruction in BASIC? What three instructions are analogous to the PEEK instruction in BASIC?
9. Consider this BASIC instruction

20 A = Y

where A and Y are BASIC variables, *not* the codes in the A and Y registers. What does the program line do? How do the values of A and Y compare after the program line 20 is executed? Can you draw an analogy between this BASIC instruction and the TYA? Describe what happens when each of the following instructions are executed: TYA, TAY, TAX, TXA.

10. Suppose the 6510 microprocessor had TYX and TXY instructions in its instruction set. Give an English-language and logical description of these hypothetical instructions.

4

Arithmetic Operations in Assembly Language

I. Introduction

For centuries human beings have been motivated to find faster and easier ways to perform arithmetic computations. The abacus, logarithms, and the slide rule are examples. The electronic computer is the most recent product of that motivation, and arithmetic operations can now take place in a matter of microseconds. In mathematical applications the computer functions as a *computer*.

Two other, though less obvious and more recent, motivations have contributed to the development of the computer. These are the necessity to manipulate enormous amounts of information and the necessity to control machines more precisely, consistently, and quickly than human beings. In the first of these applications, the computer functions as a *data processor*; while in the second, it operates as a *controller*. In fact, the microprocessor was developed in response to the need for controllers, and only later did it come to be associated with personal computers. The microcomputer is now a pervasive influence in our culture.

In this chapter, we will be concerned with doing arithmetic at its most elementary level. Although a number of recently developed microprocessors have instructions for multiplying and dividing, the 6510 is only directly capable of adding and subtracting. To multiply and divide, you must write an assembly-language program consisting of many instructions. We will consider such programs in Chapter 7. In this chapter, you will learn to use the ADC, SBC, CLC, SEC, CLD, SED, and CLV instructions in their most elementary addressing modes. The logical description of these instructions and the op codes we will use in this chapter are given in Table 4-1.

Thus far we have been able to get along quite nicely without using the processor status register, which is also called the P register. Refer once again to Figure 1-7, where the internal registers of the 6510 microprocessor are diagrammed. The P register is of fundamental importance to arithmetic operations, and we will need to discuss it in this chapter.

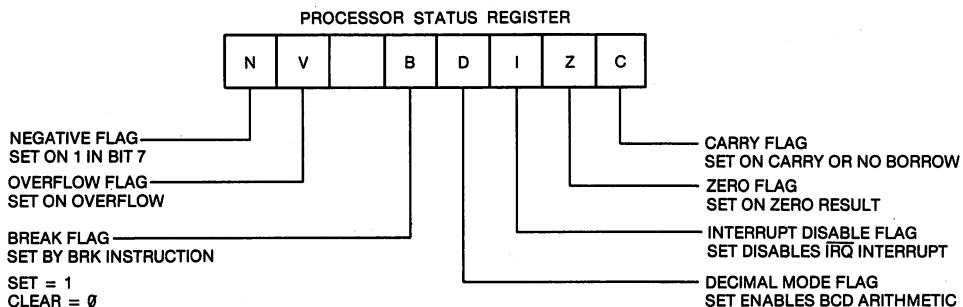
Table 4-1. Op codes for the instructions used in Chapter 4.

Mnemonic	Logical Description	Op Code for Each Addressing Mode			
		Immediate	Absolute	Zero-Page	Implied
ADC	A + M + C → A	\$69	\$6D	\$65	
SBC	A - M - C → A	E9	ED	E5	
CLC	0 → C				\$18
SEC	1 → C				38
CLD	0 → D				D8
SED	1 → D				F8
CLV	0 → V				B8
JSR	No simple description		20		
RTS					60

Finally, to provide practice in using the arithmetic instructions we will need to use one other instruction, namely, the JSR (jump to subroutine) instruction. You are already aware that in order to call a machine-language program with the BASIC language SYS command, you must end the program with a RTS (return from subroutine) instruction. The JSR and RTS instructions are companions, and they will be discussed in more detail in Chapter 9. It is possible, however, to use these two instructions without a profound understanding of them, and we have found it useful to do so.

II. The Processor Status Register and the Carry Flag

We begin with a brief discussion of the *processor status register*. It was illustrated in Figure 1-7, but it is shown with considerably more detail in Figure 4-1. Each bit in the P register is called a *status bit, flag, or condition code*. We will generally refer to the status bits as *flags*. As you shall see, the flags act as signals for certain conditions.

**Figure 4-1. Diagram of the processor status register.**

Definitions of two terms are appropriate here:

- A flag is set if there is a binary *one* in the corresponding bit of the P register.
- A flag is *clear* if there is a binary *zero* in the corresponding bit of the P register.

The importance of the carry flag to the addition operation is easily understood. Each memory location can hold an eight-bit number. The largest possible eight-bit number is \$FF (255 in decimal). The ADC instruction adds two eight-bit numbers. What happens if the sum exceeds \$FF (255)?

Recall from elementary-school arithmetic that adding the digits in one place value column frequently required a *carry* to the next column. The same situation is true in adding eight-bit numbers. When the sum of two eight-bit numbers exceeds \$FF, the *carry flag* in the P register will be *set*. In this situation, the carry flag acts as a *signal* that the sum exceeded the capacity of an eight-bit number to express it.

Assume that the numbers \$FF and \$80 are added with the ADC instruction. Since \$FF + \$80 = \$17F, the number found in the accumulator after the addition will be \$7F and the carry flag, C, will be set. If the sum of two numbers is less than \$FF, the carry flag is cleared.

We have given an informal discussion of the carry flag because it is highly important to the present discussion. The other flags will be described as the need arises. Refer to Chapter 7 for additional details.

III. Flag Modification Instructions

Associated with some of the flags in the P register are flag set and clear instructions. The flag modifiers of interest to us in this chapter are the CLC, SEC, CLD, SED, and CLV instructions. These instructions are diagrammed in Figure 4-2. When executed by the microprocessor, they either set or clear a flag in the processor status register. For example, the CLC instruction *clears* the carry flag, while the SEC instruction *sets* the carry flag.

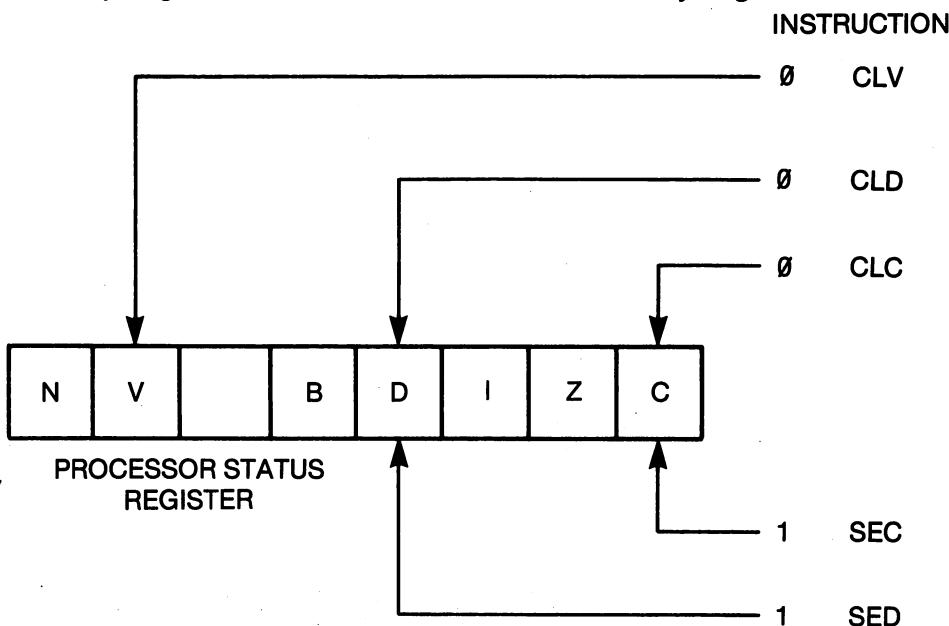


Figure 4-2. Diagram of the CLC, SEC, CLD, SED, and CLV instructions.

The CLC and SEC instructions are used to provide the correct value for the carry flag before performing an arithmetic operation. Refer to the instruction set in Table 2-1 or the discussion below to find that when the ADC instruction is executed, it adds the bit in the carry flag. Thus, the CLC instruction must be used to *clear the carry before adding* two eight-bit numbers. Otherwise, the sum may be incorrect. Also, you will soon see that the SEC instruction should be used to *set the carry before subtracting* one eight-bit number from another.

The use of the other flag modification instructions will be described as the need arises.

IV. The ADC Instruction

The four possibilities that can occur when two one-bit numbers, symbolized by A and M, are added as shown in Table 4-2. The sum produces a result R and a carry C. The carry is normally added to the more significant bit on the left when the numbers A and M are larger than single-bit numbers. The addition of two eight-bit numbers is illustrated in Figure 4-3. Notice that a carry from one bit position appears in the addition of the bits to its left. This proceeds until any *carry from bit seven appears in the carry flag* in the processor status register. Figure 4-3 is, in fact, a description of the 6510 ADC instruction. Notice that the value of the carry flag *before* execution of the ADC instruction is added to the least-significant bit. This explains why you should use a CLC instruction to clear the carry flag before the ADC instruction when doing eight-bit arithmetic.

Table 4-2. Definition of one-bit binary addition.

A + M = R	C
0 + 0 = 0	0
0 + 1 = 1	0
1 + 0 = 1	0
1 + 1 = 0	1

R = RESULT
C = CARRY

The ADC (add with carry) instruction may be defined as follows:

- The ADC instruction *adds* a number M in memory, the number A in the accumulator, and the value of the carry flag. The value of the carry flag is added to the least-significant bit. The sum is stored in the accumulator.
- Symbolically, the ADC instruction is expressed as

$$A + M + C \rightarrow A$$

where M and A are eight-bit numbers and C is the one-bit number in the carry flag.

- After the ADC instruction is executed, the carry flag will be *set* if the sum *exceeded \$FF*; otherwise, it will be *cleared*.

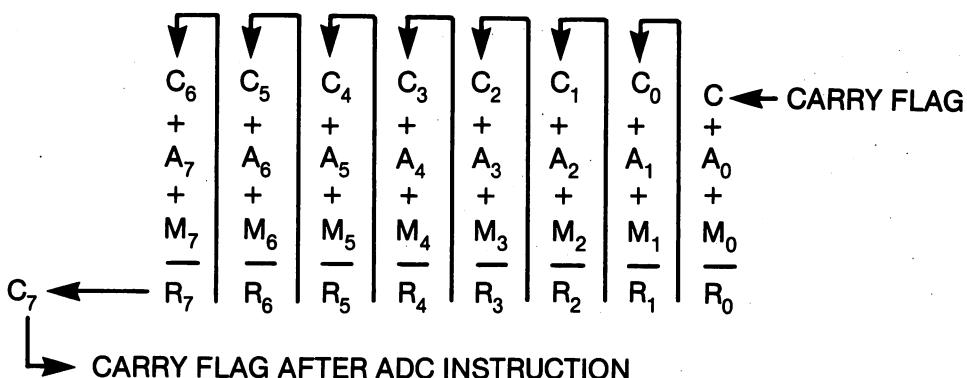


Figure 4-3. Diagram of the ADC instruction.

Example 4-1 illustrates the addition of two eight-bit numbers. Notice that the carry flag, C, is clear before the addition. Since the sum does not exceed \$FF, the carry flag is also clear after the addition.

Example 4-1. Adding Two Eight-Bit Binary Numbers

Assume \$21 is in the accumulator and \$8A is in the memory location referenced by the ADC instruction. Also assume the carry flag is clear before the ADC operation. Find the result of the ADC instruction.

Solution:

Hexadecimal	Binary	Symbolic
0	0	C
+	+	+
\$8A	10001010	M
+	+	+
<u>\$21</u>	<u>00100001</u>	A
= \$AB	= 10101011	- A
0	0	- C

Example 4-2 illustrates the use of the ADC instruction in a program. One of the addends is stored in NUM1, a symbol for the zero-page location \$00FB. The other addend is stored in location \$00FC, symbolized by NUM2. Both the LDA and ADC instructions are used in the zero-page addressing modes. The sum is stored in location \$00FD, symbolized by SUM. If the numbers \$21 and \$8A are stored in locations NUM1 and NUM2, then the answer \$AB given by Example 4-1 will be stored in SUM after the program is executed. Notice in example 4-1 that the CLC instruction was used before the ADC instruction.

Example 4-2. A Program to add two numbers

Object: Add the numbers stored in locations \$00FC and \$00FD. Store the sum in location \$00FE.

10 ADD	CLC	;CLEAR THE CARRY FLAG BEFORE ADDING.		C000 18
11 LDA	+NUM1	;FIND THE FIRST ADDEND IN NUM1.		C001 A5 FB
12 ADC	+NUM2	;ADD IT TO THE SECOND ADDEND.		C003 65 FC
13 STA	+SUM	;STORE THE ANSWER IN SUM.		C005 85 FD
14 RTS				C007 60

The exercises at the end of the chapter will provide you with additional practice in adding binary numbers. A computer-assisted instruction (CAI) program will allow you to input numbers in hexadecimal, display them in binary, find and display their sum in binary, and display the contents of the P register after the addition. If you need practice with adding binary numbers, you may wish to try some of the exercises now.

Example 4-3 illustrates a binary addition when the sum exceeds \$FF. The sum is \$11B. Observe that the number found in the accumulator will be \$1B. The carry flag is set after the addition, however, indicating that the sum exceeded \$FF. Taken together, the number in the accumulator and the value of the carry flag indicate that the sum is \$11B. This sum cannot be contained in one byte.

Example 4-3. Adding Numbers Whose Sum Exceeds \$FF

Find the sum of \$A5 and \$76. Assume the carry is clear before the addition.

Solution:

Hexadecimal	Binary	Symbolic
0	0	C
+	+	+
\$A5	10100101	M
+	+	+
<u>\$76</u>	<u>01110110</u>	A
= \$1B	= 00011011	→ A
1	1	→ C

If the sum of two numbers exceeds \$FF, how can you use the information provided in the carry flag to obtain the correct answer? The solution is shown in the program in Example 4-4. You use a second ADC instruction, add \$00, and store the result in another byte of memory. Notice in the program in Example 4-4 that the carry flag is *not cleared* before the second ADC instruction. Thus, if any carry occurs when the two numbers are added, the carry will be added to zero and it will appear in the second sum. If this program

were executed using the numbers in Example 4-3, then \$1B will be found in SUMLO and \$01 will be found in SUMHI, indicating that the complete answer is \$11B. If a sum is less than \$FF, then \$00 will be found in SUMHI.

Example 4-4. Adding Single-Byte Numbers and Saving the Carry

Object: Add two numbers and store the result. Store any carry from this sum in another location.

10	ORG	CLC	;CLEAR THE CARRY FLAG BEFORE ADDING.	C000	18
11		LDA +NUM1	;GET FIRST ADDEND.	C001	A5 FB
12		ADC +NUM2	;ADD IT TO THE SECOND ADDEND.	C003	65 FC
13		STA +SUMLO	;STORE THE EIGHT-BIT ANSWER.	C005	85 FD
14		LDA #00	;TO SAVE THE VALUE OF THE CARRY FLAG,	C007	A9 00
15		ADC #00	;FIND (0 + 0 + C) AND	C009	69 00
16		STA +SUMHI	;STORE THE RESULT IN SUMHI.	C00B	85 FE
17		RTS		C00D	60

V. Multiple-Byte Addition

It should be obvious that in many applications you will be dealing with numbers that are too large to be represented with only eight bits. These numbers are represented with two bytes. The largest whole number that can be represented with two bytes is 65,535. The number 12,345 is \$3039, so 12,345 would be represented by storing \$39 in the *least-significant byte* (LSB) and \$30 in the *most-significant byte* (MSB). If the application involves still larger numbers, then each number must be represented by three bytes. The largest three-byte number is 16,777,215 (\$FFFFFF).

When two or more bytes are used to represent numbers, then *multibyte* arithmetic is required. Two bytes are used to store each addend, and two bytes are used to store the sum. Example 4-5 illustrates such a sum, and a program to perform a double-byte addition is given in Example 4-6. It is important to

Example 4-5. Calculating a Double-Byte Sum

Find the sum of \$30D9 and \$1234. Assume the carry flag is clear.

Solution:

Hex	Binary	Symbolic	Hex	Binary	Symbolic
0	0	C	1	1	C
+	+	+	+	+	+
\$D9	11011001	A	\$30	00110000	A
+	+	+	+	+	+
\$34	00110100	M	\$12	00010010	M
= \$0D	= 00001101	- A	= \$43	= 01000011	- A
1	1	- C	0	0	- C

Therefore, the sum is \$430D.

observe that the carry flag is *not cleared* between the first and second addition. Since there is a carry from the sum of the least-significant bytes, clearing the carry flag would result in an incorrect sum. When performing multibyte arithmetic, the carry flag is cleared only before the first addition.

Example 4-6. A Program to Add Two-Byte Numbers

Object: Add the two-byte number stored at \$00FC and \$00FD to the two-byte number stored at \$00FE and \$00FF. Store the two-byte result in locations \$0002 and \$0003. For all numbers, the least-significant byte is stored in the location with the smallest address.

11	START	CLC	;CLEAR CARRY BEFORE FIRST ADD.	C000 18
12		LDA +NUM1LO	;GET LSB OF FIRST NUMBER.	C001 A5 FB
13		ADC +NUM2LO	;ADD LSB OF SECOND NUMBER.	C003 65 FD
14		STA +SUMLSB	;STORE IN LSB OF SUM.	C005 85 02
15		LDA +NUM1HI	;GET MSB OF FIRST NUMBER.	C007 A5 FC
16		ADC +NUM2HI	;ADD CARRY AND MSB OF NUM2.	C009 65 FE
17		STA +SUMMSB	;STORE SUM IN MSB OF SUM.	C00B 85 03
18		RTS		C00D 60

After studying Examples 4-5 and 4-6, you should see that the only point in having an *add with carry* (ADC) instruction is to provide for the possibility of multibyte addition. If you always restrict yourself to adding eight-bit numbers, then the carry flag is unnecessary, since you will never use it to provide a carry to a more significant byte.

VI. The SBC Instruction

The carry flag, C, is also used in the subtraction instruction, but in a more subtle way than you might expect. The carry flag can have two values, zero or one. The *complement* of the carry flag, designated by \bar{C} , has the binary value *opposite* to that found in the carry flag. Thus, if C = 1 then $\bar{C} = 0$, and if C = 0 then $\bar{C} = 1$. It may be helpful to think of the complement of the carry flag as a *borrow flag*, although you should be aware that there is not a borrow flag in the P register.

We are now ready to define the SBC instruction:

- The SBC instruction subtracts the number M in a memory location from the number A in the accumulator. The complement of the carry flag, \bar{C} , is also subtracted from the number in the accumulator. The result is stored in the accumulator.
- Symbolically, the SBC instruction is written

$$A - M - \bar{C} \rightarrow A$$

where M and A are eight-bit numbers and \bar{C} is a one-bit number.

- After execution of the SBC instruction, the carry flag will be *cleared* (indicating a borrow) if a larger number is subtracted from a smaller; otherwise, it will be set.

You can see from this definition that if the carry flag is set after a subtraction, no borrow was required. If the carry flag is cleared after a subtraction,

the need to borrow is indicated. You can also see that if you are subtracting two eight-bit numbers with the SBC instruction, the carry flag should be set before subtracting. Example 4-7 illustrates a subtraction.

Example 4-7. Subtracting Two Eight-Bit Numbers

Assume the number \$A5 is in the accumulator and the number \$5F is in the memory location referenced by the SBC instruction. Also assume the carry flag is set. Find the result of the SBC instruction.

Solution:

Hex	Binary	Symbolic
\$A5	10100101	A
-	-	-
\$5F	01011111	M
-	-	-
0	0	\bar{C}
= \$46	= 01000110	- A
1	1	- C

A program to perform a subtraction is given in Example 4-8. Notice that the carry flag is set before the SBC instruction. The number in location \$00FB is the minuend, the number in location \$00FC is the subtrahend, and location \$00FD stores the difference. We will return to this program in the exercises at the end of the chapter to obtain practice with binary subtraction.

Example 4-8. A Program to Subtract Two Numbers

Object: Subtract the number in zero-page location \$FC from the number in zero-page location \$FB. Store the result in zero-page location \$FD.

10	START	SEC	;SET CARRY BEFORE SUBTRACTING.	C000 38
11		LDA +MINUND	;FETCH THE MINUEND.	C001 A5 FB
12		SBC +SBTRND	;SUBTRACT THE SUBTRAHEND.	C003 E5 FC
13		STA +DIFF	;STORE THE DIFFERENCE.	C005 85 FD
14		RTS		C007 60

Example 4-9 illustrates a subtraction that results in a borrow. If you try binary subtraction, it is likely that you will have difficulty, especially in cases that require a borrow. It may be helpful to subtract in the same way the microprocessor subtracts. Begin by complementing the subtrahend (that is, change each of the bit values of the subtrahend), and add one to the number so obtained. The result obtained by complementing a number and adding one is called the *two's complement* of the number. Adding the two's complement of the subtrahend to the minuend will give the desired result for subtraction. Example 4-9 also illustrates how to find a difference by two's complement addition.

The two's-complement addition is, in fact, how the 6510 subtracts. Before the subtraction, the SEC instruction sets the carry flag. The subtrahend is fetched from memory, complemented, and the carry flag is added to form the two's complement. This result is added to the minuend to give the difference. In this way, circuitry that performs addition on the 6510 can also be used to perform subtraction. Two's-complement addition is also a good technique for human beings to use to subtract binary numbers. Observe in Example 4-9 that the carry flag is clear after the two's-complement addition (subtraction).

Example 4-9. Subtraction Resulting in a Borrow

Subtract \$7F from \$3A. Assume the carry flag is set before the subtraction.

Solution:

Ordinary Subtraction			Two's-complement Subtraction		
Hex	Binary	Symbolic	Hex	Binary	Symbolic
\$3A	00111010	A	\$3A	00111010	A
-	-	-	+	+	+
\$7F	01111111	M	\$80	10000000	M
-	-	-	+	+	+
0	0	C	1	1	C
= \$BB	= 10111011	→ A	= \$BB	= 10111011	→ A
0	0	→ C	0	0	→ C

Try the two's-complement approach for yourself, using the problem in Example 4-7.

VII. Multiple-Byte Subtraction

Numbers larger than \$FF must be represented with two or more bytes. When such numbers are subtracted, corresponding bytes of the minuend and the subtrahend are operated on by the SBC instruction starting with the least-significant byte (LSB) and ending with the most-significant byte (MSB). The carry flag is set with the SEC instruction before the LSBs are combined. After that, the carry flag is not modified with the SEC or CLC instruction; rather, it is the SBC instruction itself that sets or clears the carry corresponding to a "no borrow" or "borrow" situation, respectively. Example 4-10 illustrates a program that subtracts one two-byte number from another two-byte number. It should be compared with the two-byte addition program given in Example 4-6.

Example 4-10. A Two-Byte Subtraction Program

Object: Perform a two-byte subtraction and store the two-byte difference.

11	START	SEC	; SEC IS SAME AS CLEAR BORROW.	C000 38
12	LDA	MINLO	;GET LSB OF MINUEND.	C001 AD FA 02
13	SBC	SUBL0	;SUBTRACT LSB OF SUBTRAHEND.	C004 ED FC 02
14	STA	DIFFLO	;RESULT INTO LSB OF DIFFERENCE.	C007 8D FE 02
15	LDA	MINHI	;FETCH MSB OF MINUEND.	C00A AD FB 02
16	SBC	SUBHI	;SUBTRACT BORROW AND SUBTRAHEND.	C00D ED FD 02
17	STA	DIFFHI	;RESULT INTO MSB OF DIFFERENCE.	C010 8D FF 02
18		RTS		C013 60

VIII. Decimal Mode Arithmetic*

Instead of representing numbers with one or more eight-bit binary numbers, it is sometimes more convenient to use a slightly different representation, called *binary-coded decimal (BCD)*. In binary-coded decimal each decimal digit is represented by a four-bit binary number. Table 4-3 illustrates this concept. In binary-coded decimal, you will not see binary numbers such as 1010, 1011, and 1100 because they represent digits larger than nine, which do not exist in the decimal system.

The time-of-day clock on the 6526 CIA chip uses the BCD representation of numbers. This clock is organized into four registers for tenths of seconds, seconds, minutes, and hours, accessed at locations \$DD08 through \$DD0B, respectively. In each register, the time is stored as a BCD number. One reason for using the BCD representation is that it is easier to write a program to display a BCD number than a binary number. You cannot, however, PEEK at BCD numbers and expect to get the correct time because the PEEK instruction assumes it is dealing with an eight-bit binary number.

When BCD codes are being used, it is easier to think that each eight-bit location consists of two four-bit *nibbles*. In turn, each nibble represents one decimal digit, following the scheme described in Table 4-3. Thus, each memory location contains two decimal digits, represented with BCD of course.

Table 4-3. Representation of binary-coded decimal (BCD).

Decimal Number	Binary-Coded Decimal Number
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

If you are going to combine BCD numbers in arithmetic operations, then an important modification in the ADC and SBC instructions must occur. Why? Because when BCD numbers are added, you would like a carry from the low-order nibble to the high-order nibble when the sum of these two nibbles exceeds nine. Furthermore, you would like a carry from the sum of two bytes when the sum exceeds 99. Contrast that with ordinary addition in which a carry occurs when the sum exceeds \$FF (255).

The 6510 microprocessor is capable of adding and subtracting BCD numbers. Before BCD addition is used, the *decimal-mode flag* (D flag) in the processor status register must be set with the SED instruction. To return to binary addition and subtraction the decimal-mode flag is cleared with the CLD instruction.

To summarize decimal-mode arithmetic:

- The D flag must be set with the SED instruction before the ADC or SBC instruction.
- In decimal-mode addition, the carry flag will be set if the sum exceeds 99 in BCD; otherwise, it will be cleared.
- In decimal-mode subtraction, the carry flag will be *cleared* if a larger number is subtracted from a smaller number; otherwise, it will be set. Remember that the complement of the carry flag is also subtracted.

Example 4-11 illustrates the addition of two BCD numbers that results in a carry. Subtraction will be similar. Do not attempt to do two's-complement subtraction in the decimal mode. First perform the subtraction in decimal, then convert to BCD.

Example 4-11. Adding Two BCD Numbers

Add 34 and 87 in BCD. Assume the carry flag is cleared before the addition.

Solution:

Decimal	BCD	Symbolic
0	0	C
+	+	+
34	00110100	A
+	+	+
87	10000111	M
<u>—</u>	<u>=</u>	
21	00100001	— A
1	1	— C

A program to add two numbers in the decimal mode is given in Example 4-12. It is identical to the program in Example 4-2, except the decimal-mode flag is set before the ADC instruction and the decimal-mode flag has been cleared after the addition. Although it is not necessary to use the CLD instruction after a decimal-mode addition, it is good practice to do so. It should be clear that decimal-mode subtraction is similar to decimal-mode addition, with

the SBC instruction replacing the ADC instruction. It should also be clear that multibyte decimal-mode arithmetic is performed in exactly the same way as in Examples 4-6 and 4-1, except that the SED instruction must precede the arithmetic instructions, and it is good practice to conclude the program with a CLD instruction.

Example 4-12. A Program to Add Two Numbers in the Decimal Mode

Object: Add the BCD numbers stored in locations \$FB and \$FC. Store the result in location \$FD.

10 ADD	SED	;SET THE DECIMAL MODE FLAG.	C000 F8
11 CLC		;CLEAR THE CARRY FLAG BEFORE ADDING.	C001 18
12 LDA +NUM1		;FIND THE FIRST ADDEND IN NUM1.	C002 A5 FB
13 ADC +NUM2		;ADD IT TO THE SECOND ADDEND.	C004 65 FC
14 STA +SUM		;STORE THE ANSWER IN SUM.	C006 85 FD
15 CLD		;CLEAR THE DECIMAL MODE FLAG.	C008 D8
16 RTS			C009 60

IX. Signed Number Arithmetic*

Chapter 1 noted that the only information understood by the computer consists of binary ones and zeros. A minus sign is obviously not a one or a zero; it is a minus sign. Thus, the computer cannot recognize minus signs. How is it possible to represent negative numbers without minus signs?

Recall the previously described concept of the two's-complement of a number. The two's-complement of a number is obtained by changing each of the bit values of the number and adding one to the number. For example, consider the binary number 0101 1000 (\$58). Its complement is 1010 0111 (\$A7). Its two's-complement is \$A7 + 1 or 1010 1000 (\$A8). Now add the number \$58 to its two's-complement \$A8. Neglecting the carry you will get \$00.

The sum of a number and its two's-complement is zero. Let M represent an eight-bit number, and let \bar{M} represent its complement. Then $(\bar{M} + 1)$ is the two's-complement of M, and we may represent the italicized statement with the expression:

$$M + (\bar{M} + 1) = 0$$

If the sum of two numbers is zero, then one number is the *negative* of the other. Thus, the negative of M is $(\bar{M} + 1)$, the two's-complement of M.

Now we are ready to make some negative numbers for ourselves. The two's-complement of zero is zero, so zero is neither positive nor negative. The two's-complement of \$01 is \$FF, so \$FF = -1 = -\$01. The two's-complement of \$02 is \$FE, so \$FE = -2 = -\$02. We can continue this process until we get to \$7F (127). The two's-complement of \$7F is \$80, so \$80 = -128 = -\$7F. Now we have run out of eight-bit numbers.

The 256 eight-bit numbers have been divided into two groups:

- The 128 numbers from \$00 to \$7F (0 to 127) are positive.
- The 128 numbers from \$FF to \$80 (-1 to -128) are negative.

These facts are summarized in Table 4-4. (Although zero is neither positive nor negative, for programming purposes it is convenient to group it with the positive numbers.)

Table 4-4. Representation of signed numbers in binary and hexadecimal.

Decimal	Binary	Hexadecimal
-128	1000 0000	\$80
-127	1000 0001	\$81
-126	1000 0010	\$82
.	.	.
.	.	.
-5	1111 1011	\$FB
-4	1111 1100	\$FC
-3	1111 1101	\$FD
-2	1111 1110	\$FE
-1	1111 1111	\$FF
0	0000 0000	\$00
+1	0000 0001	\$01
+2	0000 0010	\$02
+3	0000 0011	\$03
+4	0000 0100	\$04
+5	0000 0101	\$05
.	.	.
.	.	.
+126	0111 1110	\$7E
+127	0111 1111	\$7F

Notice that the two groups of numbers have one significant difference. The numbers in the first group all have a zero in bit seven, while the numbers in the second group all have a one in bit seven. When working with signed numbers (integers), a *one in bit seven will indicate a negative number*. This is why bit seven is often called the *sign bit*, and this is why bit seven in the processor status register is called the *negative flag* or the N flag. It indicates that an arithmetic operation produced a negative number because it is set whenever an operation produces a one in bit seven, the sign bit.

How do you add and subtract signed numbers? Exactly as you would unsigned numbers. The programming examples given in this chapter work equally well for signed or unsigned numbers. The *interpretation* of a number as a *signed number* is made by the *programmer* rather than the program. The sign of the answer is always available in the N flag of the processor status register, where it can be tested with either of two branch instructions. This

will be explained in Chapter 6. (Decimal-mode arithmetic is an exception. It is more difficult to do signed-number arithmetic in the decimal mode, and we will not pursue this topic further.)

To handle numbers larger than 127 or smaller than -128, two or more bytes are used to represent the numbers. Bit seven in the most-significant byte of the number is interpreted as the sign of the number. It is possible to represent integers from -32,768 to +32767 with two bytes.

There is one complicating factor that remains to be discussed. Refer to Table 4-4 to follow the discussion. Suppose you add 5 to 125. Both of these are positive numbers. Adding 5 to 125 gives 130. Converting 130 to hexadecimal gives \$82. \$82 represents a binary number with a one in bit seven; hence, it is negative. In fact, \$82 represents -126. We have added two positive numbers and obtained a negative. It is clear that we have a problem.

If the sum of two positive numbers exceeds 127, there will be a carry from bit six to bit seven, the sign bit, producing an erroneous result. The sum *overflowed* into the sign bit, producing an *overflow error*. In this situation the *overflow flag*, V, in the processor status register will be set. Like the N flag, the V flag can be tested with branch instructions. The V flag is used to signal that a signed-number arithmetic operation has gone awry.

The overflow flag is also set when the sum of two negative numbers is less than -128. For example, add -2 (\$FE) to -128 (\$80). The sum really is -130, but you obtain \$7E, which is 126 (as you can see from Table 4-4). In summary,

- An overflow occurs and the V flag is set when a sum or difference is either larger than 127 or more negative than -128.
- Whenever an overflow occurs, the interpretation of the result as a signed number is no longer correct.
- The V flag can be cleared before an arithmetic operation with a CLV instruction.

For more information about signed-number arithmetic, consult the references at the end of Chapter 1.

X. The JSR and RTS Instructions

The concept of a subroutine is probably familiar to you from your work with the BASIC programming language. We will postpone a detailed discussion of subroutines until Chapter 9. The exercises at the end of this chapter will make use of subroutines, however, so it is useful to introduce the calling and return instructions at this point.

You have already been introduced to the RTS instruction. It is analogous to the RETURN command in BASIC. We have used the RTS instruction at the end of our machine-language subroutines to return control to the BASIC interpreter after a SYS command. Machine-language subroutines can also be called from machine-language programs. These subroutines must also end with the RTS instruction. The RTS instruction is a single-byte instruction (op code \$60) that uses only the implied addressing mode.

To call a machine-language subroutine from a machine-language program the JSR instruction is used. It is analogous to the GOSUB command in BASIC. JSR is a three-byte instruction that uses the absolute addressing mode. The first byte of the instruction is the op code (\$20), and the second and third bytes are the address of the location of the first program byte in the subroutine. Recall that an address consists of a low-order byte called the ADL and a high-order byte called the ADH. The JSR instruction requires that the address be ordered with the ADL first and the ADH last. Thus, if a subroutine is located at \$FEDC, the complete JSR instruction is

20 DC FE

Refer to Example 4-13 to see how the JSR instruction is written in assembly language. It has the form

JSR SUBPRO

where SUBPRO is the *label* of the starting address of the subroutine. This is the first illustration of the case where the operand field of the assembly-language program contains a label.

The subroutine must end with an RTS instruction. Execution of the RTS instruction causes the program to return to the instruction in the main program following the JSR instruction that called the subroutine. Refer to the exercises at the end of this chapter for illustrations of the JSR instruction, or refer ahead to Chapter 9 for a detailed description of subroutine calls and returns.

XI. Summary

The carry flag C is a bit in the processor status register (P register). C is set when the ADC instruction produces a sum that exceeds \$FF; otherwise, it is cleared. The carry flag is also used in subtraction. C is cleared when a larger number is subtracted from a smaller, indicating the necessity for a borrow; otherwise, it is set.

The ADC instruction adds an eight-bit number in memory, the eight-bit number in the accumulator, and the value of the carry flag and places the sum in the accumulator. The carry flag should be cleared with the CLC instruction before adding single-byte numbers. In multibyte addition, the carry flag is cleared before the LSBs are added.

The complement of the carry flag serves as a borrow flag when subtracting. The SBC instruction subtracts the borrow flag and a number in memory from the number in the accumulator and stores the difference in the accumulator. The carry flag should be set with the SEC instruction before subtracting single-byte numbers. In multibyte subtraction, the carry flag is set before the LSBs are subtracted.

The decimal mode (D) flag is set with the SED instruction before performing arithmetic operations on BCD numbers. D must be cleared with the CLD instruction before performing arithmetic operations on binary numbers.

The N and V flags are used when programmers choose to work with signed numbers. When working with signed numbers, bit seven is the sign bit; one means negative, zero means positive. The N flag may be used to indicate the sign of a sum or difference. The V flag may be used to signal an overflow into the sign bit, which destroys its validity.

Machine-language subroutines are called from a machine-language program with a JSR instruction. Control returns to the calling program upon execution of the RTS instruction.

XII. Exercises

We have developed a computer-assisted instruction (CAI) program to give you some practice with binary numbers and addition and subtraction operations. The hexadecimal codes of the computer-assisted instruction program are listed in Table 4-5. A complete assembly-language listing of this program is given in Appendix B. For the present, all you will need to do is load the hexadecimal codes using the program in Example 2-8 and the information in Table 4-5. Do this now, using the starting address of \$C100 when Example 2-8 is RUN.

Table 4-5. Hexadecimal codes for the computer-assisted instruction program.*

20	84	FF	A9	0D	20	D2	FF	20	13	C1	85	FB	20	13	C1
85	FC	60	20	E4	FF	C9	00	F0	F9	85	02	20	D2	FF	A5
02	20	61	C1	0A	0A	0A	0A	85	02	20	E4	FF	C9	00	F0
F9	48	20	D2	FF	68	20	61	C1	05	02	85	02	A2	06	A9
20	20	D2	FF	CA	D0	F8	A2	08	A5	02	85	97	06	97	A9
00	69	30	20	D2	FF	CA	D0	F4	A9	0D	20	D2	FF	A5	02
60	C9	40	B0	04	29	0F	10	02	E9	37	60	08	48	08	68
A2	08	85	FE	68	48	85	02	A9	20	20	D2	FF	CA	D0	F8
A2	08	06	02	A9	00	69	30	20	D2	FF	CA	D0	F4	A9	0D
20	D2	FF	A2	00	BD	CF	C1	20	D2	FF	E8	E0	11	90	F5
A2	08	A9	20	20	D2	FF	CA	D0	F8	A2	08	06	FE	A9	00
69	30	20	D2	FF	CA	D0	F4	A9	0D	20	D2	FF	A9	0D	20
D2	FF	68	28	60	08	48	20	E4	FF	F0	FB	68	28	60	20
20	20	20	20	20	20	20	4E	56	20	42	44	49	5A	43	0D

*The starting address for this machine-language program is \$C100. It can be loaded with the program in Example 2-8, saved on disk with the program in Example 2-10, and reloaded from disk with the program in Example 2-11.

What does the computer-assisted instruction program do? It allows you to input a byte of information in the form of two hexadecimal digits and it displays the byte as a binary number. Then you input another byte by typing

in two more hexadecimal digits. This byte is also displayed in binary. The bytes are stored in locations \$00FB and \$00FC. This much of the program is called subroutine GETTWO, and it starts at \$C100.

The computer-assisted program also contains a subroutine that will display, in binary, the eight-bit code found in location \$00FD and the flags in the processor status register. This subroutine is called DISPLAY, and it starts at \$C161.

We will use the computer-assisted instruction program in connection with the programs in Examples 4-2, 4-8, and 4-12. Refer to these programs and note that they obtain their data from locations \$00FB and \$00FC, the same locations that subroutine GETTWO in the computer-assisted instruction program places numbers. Thus, the programs in Examples 4-2, 4-8, and 4-12 can be called as subroutines to perform addition or subtraction using the numbers obtained from the keyboard using subroutine GETTWO.

The programs in Examples 4-2, 4-8, and 4-12 store their results in location \$00FD. This is the same location that the computer-assisted instruction program reads and displays. The DISPLAY subroutine allows you to check the result of the arithmetic operation and the status of the various flags after the ADC or SBC instruction is executed. Subroutine GETTWO and DISPLAY are both listed in Appendix B.

Example 4-13 illustrates how subroutines GETTWO and DISPLAY are used with the program in Example 4-2. The part of the program in Example 4-13 that is Example 4-2 is delineated by asterisks.

If you haven't already done so, load the computer-assisted instruction program using a starting address of \$C100, the program in Example 2-8, and the table of hexadecimal codes listed in Table 4-5. Next, load the program in Example 4-13 starting at location \$C000.

Example 4-13. A Program to Demonstrate Addition

Object: Input eight-bit addends, display them, display the sum, and display the P register.

```

10      ;*****
11 ADD    CLC          ;CLEAR THE CARRY FLAG BEFORE ADDING.   C000 18
12      LDA +NUM1      ;FIND THE FIRST ADDEND IN NUM1.        C001 A5 FB
13      ADC +NUM2      ;ADD IT TO THE SECOND ADDEND.       C003 65 FC
14      STA +SUM       ;STORE THE ANSWER IN SUM.         C005 85 FD
15      RTS          ;*****                                     C007 60
16
17 START  JSR GETTWO   ;GET TWO BYTES FROM KEYBOARD.      C008 20 00 C1
18      JSR ADD       ;ADD THE TWO BYTES.           C00B 20 00 C0
19      JSR DISPLAY    ;DISPLAY A AND THE P REGISTER.     C00E 20 6C C1
20      RTS          ;RETURN TO BASIC PROGRAM.      C011 60

```

Once the codes in Table 4-5 have been loaded into memory, and the entire program in Example 4-13 has been loaded into memory, then execute the

program by calling it as a subroutine from BASIC. Notice that the first instruction to be called is JSR GETTWO, located at \$C008 (49160). This instruction is labeled START in Example 4-13. Thus, you will use the

SYS 49160

command from BASIC to call the program in Example 4-13. Here is the BASIC program you use to access the machine-language program in Example 4-13:

```
10 SYS 49160
20 GOTO 10
```

RUN it when the machine-language programs are loaded.

Once the BASIC program is running, press any key that represents a hexadecimal digit. You will see the number displayed on the screen. You do not have to press RETURN. Press another key that represents a hexadecimal digit. You have now entered the information for one byte, and the byte will be displayed in binary. Enter another byte. After entering the second byte, the addition program is called by the JSR ADD instruction and the bytes are added. Finally, the result of the addition is displayed. The flags in the processor status register (P register) after execution of the arithmetic operation are also displayed.

Here are some problems to try with the computer-assisted instruction program. Working them will give you a deeper understanding of binary numbers, binary arithmetic, and the flags in the processor status register (P register).

1. Add the following pairs of numbers: (\$A5, \$5A), (\$7F, \$80), (\$11, \$EE) and (\$A1, \$5E). Do the sums produce a carry? Check the C flag. What is the status of the Z flag after each sum? The N flag? What do you conclude is the result of adding a number to its complement?
2. Add the following pairs of numbers: (\$A6, \$5A), (\$7F, \$81), (\$EF, \$11), and (\$A3, \$5D). Do these sums produce a carry? What is the status of the Z flag after each sum? The N flag? What do you conclude is the result of adding a number to its two's-complement? Under what circumstances is the Z flag set? The N flag?
3. Add the following pairs of numbers: (\$2F, \$51), (\$EE, \$92), (\$65, \$1A), and (\$7F, \$01). Can you tell under what circumstances the N flag is set?
4. Try some signed-number arithmetic. Refer to Table 4-4. Add \$05 to \$FB. Add \$05 to -3, which is \$FD. Do you get +2? Add -5 (\$FB) to +2 (\$02). Do you get -3? (Remember, $-3 = \$FD$ in two's-complement form.) Was the V flag set in any of these cases?
5. Add +126 (\$7E) and +3 (\$03). What is the value of the sign bit, bit seven (and the N flag). Is 126 + 3 negative? What is the status of the overflow flag (V flag) after this addition? Try adding -126 (\$82) and -3 (\$03). Note the sign of the answer and the status of the V flag. When is the V flag set?

Practice some subtraction problems. Replace the addition subroutine in Example 4-13 with the subtraction routine found in Example 4-8. What op codes must you change? Here are some problems to try.

1. Subtract the second number from the first using these pairs of numbers: (\$F0, \$6F), (\$77, \$3F), and (\$FF, \$7F). Predict the value of the carry flag before you perform the subtraction. Were you right? Under what circumstances will the Z flag be set in a subtraction? Try some problems to verify your hypothesis.
2. Repeat the problems given above but subtract the first number from the second. Predict the value of the carry flag before you perform the subtraction.
3. Refer to Table 4-4 and make up some subtraction problems involving signed numbers. Try them using the CAI program. Can you predict the values the C and Z flags will have after the subtraction?
4. Make up some subtraction programs that will set the V flag.

If you have been successful with the last two problems, you have an excellent grasp of the arithmetic operations and how they modify the flags in the processor status register.

Modify the program in Example 4-13 once more so that you can do the decimal-mode arithmetic program in Example 4-12. To do this you will have to move the instructions on lines 17-20 two bytes higher up in memory to make room for the op codes of the SED and CLD instructions. Try these problems: (88, 22), (66, 33), (99, 1), and (23, 32). When is the carry flag set? Is the Z flag set when the result is zero?

Invent some problems to *discover when the N, Z, and V flags are set*. Here are two additional programming problems to try.

1. Write a program to subtract, in the decimal mode, the number in location \$00FC from the number in location \$00FB. Store the result in \$00FD and clear the decimal mode.
2. Write a subroutine that increments a two-byte BCD number by one each time the subroutine is called. Routines similar to this are used to count external events, such as people moving through a turnstile, cartons passing on a conveyor belt, or pulses coming from a Geiger counter.

5

Logic Operations in Assembly Language

I. Introduction

Everyone understands that a computer must be able to perform arithmetic operations, but the use of logic operations is less intuitive, at least for the beginning assembly-language programmer. In this chapter, you will see that the three logic operations are used extensively in programming complex chips like the SID and VIC. If you have written graphics or music programs in Commodore 64 BASIC, then you may already be familiar with two logic operations, AND and OR. The purpose of this chapter is to introduce you to the three logic operations available to the assembly-language programmer, namely, AND, OR, and EXCLUSIVE OR. These logic operations correspond to the three 6510 assembly-language instruction mnemonics AND, ORA, and EOR, respectively.

The logical descriptions and the op codes of these instructions are given in Table 5-1. Notice the symbols that are used for the logic instructions:

AND = \wedge

ORA = \vee

EOR = ∇

Table 5-1. Op codes for the instructions used in Chapter 5.

Mnemonic	Logical Description	Op Codes for Each Addressing Mode		
		Immediate	Absolute	Zero Page
AND	$A \wedge M \rightarrow A$	\$29	\$2D	\$25
ORA	$A \vee M \rightarrow A$	09	0D	05
EOR	$A \nabla M \rightarrow A$	49	4D	45

(In some literature, a dot is used for AND, a plus sign for OR, and a circled plus sign for the EXCLUSIVE OR.) For simplicity, we will confine ourselves to the use of the three addressing modes listed in Table 5-1. Like arithmetic

operations, logic operations involve *two* numbers. One of the numbers involved in the logic operation is in the accumulator (A) and the other is in memory (M). We begin by defining each of the three operations; next we will show how the instructions are used in programs, and finally we will illustrate some applications.

II. The AND, ORA, and EOR Instructions

When *two* one-bit numbers are combined in an operation such as addition, subtraction, and a logical AND, there are *four* possible bit combinations that can occur and must be considered. The four unique bit combinations are

0	0
0	1
1	0
1	1

Refer to Table 5-2 to see that the result of the AND instruction for each of the four possible one-bit operations is defined in the third column. You can see that the result of an AND operation is one only if the bits being ANDed are both one; otherwise, the result is zero.

Table 5-2. Definitions of the logical operations.

AND		ORA		EOR	
A	M	A \wedge M	A	M	A \vee M
0	0	0	0	0	0
0	1	0	0	1	1
1	0	0	1	0	1
1	1	1	1	1	0

This fact is related to the way humans think, and it may be demonstrated with an English-language statement. An AND statement such as

"She owns a car *and* she owns a computer."

is true (one) only if both "owns a car" and "owns a computer" are true (one). Otherwise, it is false (zero).

Of course, the 6510 microprocessor in the Commodore 64 operates on eight-bit numbers rather than one-bit numbers. Corresponding bits of the two numbers combined in the AND operation are treated exactly as defined in Table 5-2. This is illustrated in Table 5-3, where the AND operation is illustrated for two eight-bit numbers. Notice once again that corresponding bits of the number in the accumulator (A) and a number in memory (M) are combined using the definition in Table 5-2.

Tables 5-2 and 5-3 also define the ORA and EOR instructions. Carefully study Tables 5-2 and 5-3 and you will observe that:

- The result of an AND instruction is one only if both bits are one; otherwise, the result is zero.

- The result of an ORA instruction is zero only if both bits are zero; otherwise, the result is one.
- The result of an EOR instruction is one only if the bits have different values; otherwise, the result is zero.

The latter two facts may also be demonstrated with English-language statements. The statement

"I have a VIC 20 or a Commodore 64."

is false (zero) only if both "have a VIC 20" and "have a Commodore 64" are false (zero.) Is the statement false if I own both machines? Also, the statement

"It is Tuesday or it is Wednesday."

is true (one) if "it is Tuesday" is true (one) and "it is Wednesday" is false (zero), or vice versa. The statement is false (zero) if both components are false (it is neither Tuesday nor Wednesday) or both are true (an impossible situation). "Or" is used both *inclusively* and *exclusively* in our language.

Table 5-3. Examples of eight-bit logical operations.

AND		ORA		EOR	
\wedge	A 11001010 M <u>10101100</u>	\vee	A 11001010 M <u>10101100</u>	\vee	A 11001010 M <u>10101100</u>
\neg	A 10001000	\neg	A 11101110	\neg	A 01100110

Just as in addition and subtraction, the *result* of an AND, ORA, or EOR instruction is placed in the accumulator.

Complementation is another logical operation. To *complement* a number, all of its bit values are *changed*. In the last chapter, you complemented a number and then added one to form the two's-complement of the number in order to do signed-number arithmetic. You should understand that the complement of a number and the two's-complement of a number are different numbers: they differ by one.

The 6510 instruction set does not include a complement instruction (called NOT in some instruction sets). How can the 6510 complement a number? Notice that if a bit is EXCLUSIVE ORed with one, then the result is the complement of the bit. Refer to Table 5-2 and observe that

$$\begin{aligned} 0 \text{ EOR } 1 &= 1 \\ 1 \text{ EOR } 1 &= 0 \end{aligned}$$

Thus, an EOR with 1 complements the bit. *To complement an eight-bit number, it is EORed with \$FF.* That is,

$$M \text{ EOR } \$FF = \bar{M}$$

where we have used ' \bar{M} ' to symbolize the *complement* of ' M '.

The exercises at the end of the chapter will provide you with experience using the logic instructions AND, ORA, and EOR. You must be well-acquainted with these instructions before you can see how to use them. We turn next to several illustrations of these instructions in programs.

III. Simple Programs to Illustrate the Logic Instructions

The program in Example 5-1 illustrates the use of an AND instruction in a program. The program ANDs the number in location \$00FB with the number in \$00FC and stores the result in location \$00FD. These are the same locations used by the computer-assisted instruction (CAI) program introduced in Chapter 4. The exercises at the end of this chapter are designed to use the programs in Examples 5-1 to 5-3 and the CAI program to allow you to gain experience with the logical instructions.

Example 5-1. A Program to Demonstrate the AND Instruction

Object: AND the numbers in locations \$00FB and \$00FC.

10 BEGIN LDA +NUM1 ;GET THE FIRST NUMBER.	C000 A5 FB
11 AND +NUM2 ;AND IT WITH THE SECOND.	C002 25 FC
12 STA +RESULT ;STORE THE RESULT.	C004 85 FD
13 RTS	C006 60

The programs in Examples 5-2 and 5-3 are identical to the program in Example 5-1, except that they substitute the ORA and EOR instructions, respectively, for the AND instruction.

Example 5-2. A Program to Demonstrate the ORA Instruction

Object: ORA the numbers in locations \$00FB and \$00FC.

10 BEGIN LDA +NUM1 ;GET THE FIRST NUMBER.	C000 A5 FB
11 ORA +NUM2 ;OR IT WITH THE SECOND.	C002 05 FC
12 STA +RESULT ;STORE THE RESULT.	C004 85 FD
13 RTS	C006 60

Example 5-3. A Program to Demonstrate the EOR Instruction

Object: EOR the numbers in locations \$00FB and \$00FC.

10 BEGIN LDA +NUM1 ;GET THE FIRST NUMBER.	C000 A5 FB
11 EOR +NUM2 ;EXCLUSIVE OR IT WITH THE SECOND.	C002 45 FC
12 STA +RESULT ;STORE THE RESULT.	C004 85 FD
13 RTS	C006 60

How do the logic instructions modify the flags in the processor status register? Recall that the ADC and SBC instructions made extensive use of and modified the carry flag. Also remember that when doing signed-number

arithmetic, the V flag was of considerable importance. The logic instructions (AND, ORA, and EOR) *do not* modify the carry flag (C) or the overflow flag (V). On the other hand,

- If the result of a logic instruction is *zero*, then the Z flag will be set; otherwise, it will be cleared.
- If the result of a logic instruction has a *one* in bit seven, the most-significant bit, then the N flag will be set; otherwise, it will be cleared.

If an instruction modifies a flag in the P register, this fact is indicated in the last column of the instruction set summarized in Table 2-1. Refer to Table 2-1 to confirm the fact that the logic instructions modify the N and Z flags. You can also verify these facts using the CAI program and the exercises at the end of this chapter.

IV. Using the Logic Instructions

The ORA and AND instructions are used to *set* or *clear* specific bits in a memory location *without* affecting the other bits in the same memory location.

For example, suppose we wish to select the bit-mapped mode (BMM) of the VIC chip. In this mode, each bit in an 8K block of memory corresponds to a pixel (picture element) on the screen. In the bit-mapped mode, the screen consists of 320 (horizontal) by 200 (vertical) pixels, giving a total of 64,000 picture elements. Since one byte of memory can control eight pixels, we require 8,000 bytes of memory to control the 64,000 pixels. In this mode, the Commodore 64 has the highest resolution (most pixels) and the bit-mapped mode is ideal for detailed graphics. See Chapter 11 for more details.

To select the bit-mapped mode, bit five in VIC register \$11 must be *set*. To return to the character display mode, the same bit must be *cleared*. The program in Example 5-4 illustrates how an ORA instruction sets a bit, bit five in location \$D011. Setting this bit in register \$11 of the VIC chip selects the bit-mapped mode. When the operand of the ORA instruction is written in binary, it is easy to see which bit we wish to set and which bits we do not want to modify. If the program is executed, you will see "garbage" because you have not put any useful information in the memory locations mapped onto the screen.

Example 5-4. Setting a Bit with an ORA Instruction

Object: Switch on the bit-mapped mode.

10	ORIGIN	LDA #\\$20	;	\$20 = 00100000, BIT 5 IS SET.	C000	A9	20
11		ORA VIC11	;	OR \\$20 WITH REGISTER 11 IN VIC.	C002	0D	11 D0
12		STA VIC11	;	STORE RESULT IN VIC CHIP.	C005	8D	11 D0
13		RTS			C008	60	

The program in Example 5-5 turns off the bit-mapped mode by *clearing* bit five in register \$11 of the VIC chip. Notice that an AND instruction was used.

When the operand of the AND instruction is written in binary, it is easy to see which bit we wish to clear and which bits we do not wish to modify.

Example 5-5. Clearing a Bit with the AND Instruction

Object: Switch off the bit-mapped mode.

10 ORIGIN LDA #\$DF	;\$DF = 11011111, BIT 5 IS CLEAR.	C009 A9 DF
11 AND VIC11	;AND \$FD WITH REGISTER 11 IN VIC.	C00B 2D 11 D0
12 STA VIC11	;STORE RESULT IN VIC CHIP.	C00E 8D 11 D0
13 RTS		C011 60

First execute the program in Example 5-4 using the SYS 49152 command in BASIC. The screen will change. Then run the program in Example 5-5 using the SYS 49161 command. You will not be able to see this command as it is entered, but enter it anyway. After the program executes, you will be back in the normal screen mode, and you can see that you entered the correct SYS 49161 command.

It is possible to accomplish the objectives given in Examples 5-4 and 5-5 with an EOR instruction. Refer to Example 5-6 where the value of bit five of register \$11 in the VIC chip is *switched* using an EOR instruction, changing the mode. Execute this program with a SYS 49170 command. Execute it again with the SYS 49170 command. Each time this routine is called, bit five in the VIC chip register \$11 is switched. This would be a useful routine if you are switching back and forth between the bit-mapped mode and a text-filled screen. Notice that successive applications of the EOR instruction cause a bit to *toggle* back and forth between zero and one.

Example 5-6. Switching a Bit with the EOR Instruction

Object: Toggle the bit-mapped mode.

10 ORIGIN LDA #\$20	;\$20 = 00100000, BIT 5 IS SET.	C012 A9 20
11 EOR VIC11	;EOR \$20 WITH REGISTER 11 IN VIC.	C014 4D 11 D0
12 STA VIC11	;STORE RESULT IN VIC CHIP.	C017 8D 11 D0
13 RTS		C01A 60

We can summarize our results after making this definition:

- A bit is *set* if it has a value of one.
- A bit is *clear* if it has a value of zero.

To summarize:

- A specific bit (or bits) in a memory location may be *set* by *ORAing* the number in the memory location with a number with a one (or ones) in the bit (or bits) to be set.
- A specific bit (or bits) in a memory location may be *cleared* by *ANDing* the number in the memory location with a number with a zero (or zeros) in the bit (or bits) to be cleared.

- A specific bit (or bits) in a memory location may be *switched* by EORing the number in the memory location with a number with one (or ones) in the bit (or bits) to be switched.

Of course, after the logic operation is performed, the number must be stored (STA) in the memory location to be modified.

These various forms of bit manipulation work because ORAing a bit with one changes the bit to one, but ORAing a bit with zero leaves the bit unchanged. Likewise, ANDing a bit with zero changes the bit to zero, but ANDing a bit with one leaves the bit unchanged. Finally, EORing a bit with one switches the bit, but EORing a bit with zero leaves the bit unchanged.

Example 5-7 illustrates a case where two bits are cleared with an AND instruction. One of them is subsequently set with an ORA instruction. The VIC chip can be made to look at various 16K banks of memory. This is useful if you wish to switch between character sets or pictures displayed on the screen. The switching is accomplished by setting and clearing bits zero and one in an I/O port whose address is \$DD00. For more details, refer to either the *Programmer's Reference Guide*, page 102, or Chapter 11 in this book. Our purpose here is simply to illustrate the *use* of the logic instructions.

Refer to Example 5-7 and observe that the AND \$FC instruction clears bits zero and one. Next the ORA \$02 instruction sets bit one. The result is stored in the I/O port. Only bits zero and one of this port are modified; all the other bit values remain the same.

Example 5-7. Clearing and Setting Several Bits

Object: Select bank one (\$4000 through \$7FFF) for the VIC.

10	ORIGIN	LDA PORTA	;READ THE I/O PORT.	C000	AD	00	DD
11		AND #\$FC	;CLEAR THE TWO LOW BITS.	C003	29	FC	
12		ORA #\$02	;SELECT BANK 1 FOR VIC TO	C005	09	02	
13		STA PORTA	;LOOK AT LOCATIONS \$4000-\$7FFF.	C007	8D	00	DD
14		RTS		C00A	60		

The process of clearing one or more bits of a given number is called *masking*. The cleared bits are said to be *masked*. To mask the high-order nibble (most-significant four bits) of a byte, it is ANDed with \$0F (00001111). The number \$0F is called the *mask*. How would you mask the low-order nibble of a byte? Choose \$F0 (11110000) for the mask. How would you mask the odd-numbered bits? Choose \$55 (01010101) for the mask.

The program in Example 5-8 illustrates a number of important concepts introduced in this and the preceding two chapters. Study this program carefully. The concept of a mask is illustrated. Data transfer instructions are found on the first three lines. A logic instruction and arithmetic instructions follow. The program has a subroutine call, and it concludes with a return from subroutine instruction.

The function of the program is to read and display one register of the time-of-day (TOD) clock on the 6526 CIA chip. The TOD clock keeps time in tenths of seconds, seconds, minutes, and hours, requiring four memory locations,

\$DC08 through \$DC0B. The time is kept in *binary-coded decimal* (BCD), described in Chapter 4. Thus, the four least-significant bits of location \$DC09 contains the *ones* of seconds. The next three more-significant bits in the same memory location represent the *tens* of seconds. Of course, the tens of seconds will only advance to five, with nine in the ones of seconds, after which the minutes counter increases and the seconds register returns to zero.

Example 5-8. A Program to Count Seconds

Object: Read and display the ones of seconds in the TOD clock.

10 ORG	LDA #00	;START THE CLOCK.	C000 A9 00
11	STA CLKTHS		C002 8D 08 DC
12 HERE	LDA CLKSEC	;GET SECONDS.	C005 AD 09 DC
13	AND #\$0F	;MASK HIGH ORDER NIBBLE.	C008 29 0F
14	CLC	;ADD \$30 TO CONVERT TO	C00A 18
15	ADC #\$30	;A C-64 CHARACTER CODE.	C00B 69 30
16	JSR CHROUT	;KERNEL OUTPUT ROUTINE.	C00D 20 D2 FF
17	RTS		C010 60

The first two lines of the program in Example 5-8 start the clock. Writing to the tenths-of-seconds location, CLKTHS, starts the clock. We chose to start it at zero. The clock starts at the completion of the STA CLKTHS instruction. In line 12, the seconds location of the clock is read with the LDA CLKSEC instruction. We now have tens of seconds and units of seconds, in BCD, in the accumulator. To display a BCD number, you handle one digit at a time. Thus, we *mask* the high-order nibble, leaving only ones-of-seconds in the accumulator. The masking is accomplished with the AND #\$0F instruction.

You cannot just send a number to an output routine or a screen memory location and have it displayed. The number must be represented by its screen code or ASCII code (see Appendices E and F in the *Commodore 64 User's Guide* that came with your computer). If you study the codes for the digits 0 through 9, you will see that they are 48 through 57 (\$30 through \$39). Thus, a single-digit number, such as we now have in the accumulator, can be converted either to a screen code or ASCII by adding 48 (\$30). This is the purpose of the CLC and the ADC #\$30 instructions. Finally, we make use of a subroutine in the Commodore 64 operating system to output the character to the screen.

Here is a short BASIC routine to call the program in Example 5-8:

```

1 SYS 49152 : REM START AND READ CLOCK.
2 PRINT "<HOM>" : REM TIME APPEARS UPPER LEFT.
3 SYS 49157 : REM READ THE CLOCK.
4 GO TO 2 : REM READ AND DISPLAY FOREVER.

```

Load both programs and RUN the BASIC program. You should see the seconds ticking in the upper left-hand corner of the screen. Modify the program to read and display the tenths-of-seconds register whose address is

\$DC08. Displaying all four registers simultaneously is beyond the scope of the material you have learned in the first five chapters, but we will return to this topic later.

We conclude this section with a hypothetical problem that provides another illustration of the use of the logic instructions. Assume we have several devices connected to the user port on the Commodore 64. This port is accessed at location \$DD01, and it will be symbolized by PORT. The hypothetical devices attached to this port are listed in Table 5-4. We would like to be able to detect (1) whether a device has changed its state, and if so, (2) did it change from one to zero, or (3) did it change from zero to one? Location PSTATS (\$00FB) will be used to store the *prior-status nibble*, that is, the binary values of the four bits at an earlier time. After testing for items (1), (2), and (3), the prior-status nibble should be updated. Location FLIP (\$00FC) will contain ones in each bit that changed its status. Location OTOZ (\$00FD) will contain a one in each bit that changed from one to zero, and location ZTOO (\$00FE) will contain a one in each bit that changed from zero to one.

Table 5-4. Status information for four input devices.

Device	Bit Number	Status Information
Smoke detector	0	0=No smoke 1=Smoke
Intrusion detector	1	0=Intruder detected 1=No intruder
Line voltage detector	2	0=Power failure 1=Power on
Touch-sensitive detector on the safe	3	0=No touch 1=Touch

Example 5-9 is the solution to this problem. It is important to realize that when a solution appears in a book, it probably did not simply emerge full blown from the author's mind. A considerable amount of trial and error and pencil-and-paper calculations by many hands take place before a final version is reached. Space does not permit us to show how much work is involved in finding the solution; only the solution is shown. Likewise, your understanding of the solution will require some pencil-and-paper work. See the instructions below that suggest how to verify that the program works.

Example 5-9. A Program to Detect State Changes

Object: Detect state changes on pins 0 – 4 of the user input port.

10 ORG	LDX PORT	;READ THE PORT FOR FRESH DATA.	C000 AE 01 DD
11 TXA		;HOLD DATA IN X, USE DATA IN A.	C003 8A
12 AND #\$0F		;MASK HIGH ORDER NIBBLE.	C004 29 0F
13 EOR +PSTATS		;EOR WITH PREVIOUSLYOBTAINED DATA.	C006 45 FB

```

14      STA +FLIP      ;SET ALL BITS THAT FLIPPED.          C008 85 FC
15      AND +PSTATS   ;SET BITS FLIPPED FROM 1 TO 0.      C00A 25 FB
16      STA +OT0Z     ;STORE RESULT.                  C00C 85 FD
17      EOR +FLIP      ;SET BITS FLIPPED FROM 0 TO 1.      C00E 45 FC
18      STA +ZTOO     ;STORE RESULT.                  C010 85 FE
19      STX +PSTATS   ;UPDATE PREVIOUS STATUS BYTE.    C012 86 FB
20      RTS

```

Note in Example 5-9 the use of the EOR instruction to detect changes in bit values and the use of the AND instruction to mask the bits of interest. How can you verify that the program works as advertised? Start by assuming the number in the port was 00000110, corresponding to a normal state of affairs. Only the low-order nibble is of interest, so work with the number 0110, and assume it is in PSTATS. Next, assume that bit two changes to zero, indicating a power failure (except for the computer), and that bit zero changes to one, indicating the presence of smoke. Thus, the low-order nibble of the number obtained from PORT is 0011. Starting at the beginning of the program and using these two numbers, make the same calculations that the program makes and verify that it works. Next to each instruction write the number in the accumulator at the completion of the instruction. An alternative is to use the debug or single-step mode of your assembler package and step through the program, observing the contents of the accumulator after each step.

Table 5-5. Memory assignments for the program in Example 5-9.

Location	Symbol	Function
\$DD01	PORT	Input port for the devices in Table 5-4.
\$00FB	PSTATS	Contains bit values previously read.
\$00FC	FLIP	Contains ones for each bit that flipped.
\$00FD	OT0Z	Contains ones for bits flipped from 1 to 0.
\$00FE	ZTOO	Contains ones for bits flipped from 0 to 1.

V. Summary

The AND instruction combines corresponding bits in two eight-bit numbers according to the following rule: if both of the bits are one, then the result is one; otherwise, the result is zero. The ORA instruction operates on numbers with this rule: if both of the bits are zero, then the result is zero; otherwise, it is one. The EOR instruction combines bits with this rule: if the bits have different values, then the result is one; otherwise, it is zero. A number may be COMPLEMENTED by combining it with \$FF in an EOR operation.

The ORA instruction is used to set bits, the AND instruction is used to mask (clear) bits, and the EOR instruction is used to complement (switch) bits. When executed, these instructions will set the Z flag if all eight bits of the result are zero; otherwise, the Z flag will be cleared. The instructions also set the N flag if they produce a result of one in bit seven; otherwise, the N flag will be cleared.

VI. Exercises

It is important to have a strong intuitive feeling for the three logic instructions. They should become as familiar to you as adding and subtracting. To help familiarize you with these three operations we, once again, suggest the use of the computer-assisted instruction (CAI) program. This program was used to give you practice adding and subtracting at the end of the last chapter, and it can also be used in conjunction with Examples 5-1 to 5-3 to provide practice with the AND, ORA, and EOR instructions, respectively. The CAI program is listed in Appendix B, and a listing of the hexadecimal codes was given in Table 4-5. Begin by loading the CAI program into memory from either the tape or disk where you saved it. Otherwise, load it from the hexadecimal codes in Table 4-5, using the program in Example 2-8 and a starting address of \$C100.

The program in Example 5-10 illustrates how the CAI subroutines are used. The part of the program delineated by asterisks is Example 5-1. This part of the program may be replaced by Example 5-2 or Example 5-3 to demonstrate the ORA and EOR instructions, respectively. In fact, you have to change only one op code to move from one logic instruction to the other, so only the code at location \$C002 need be changed to demonstrate all three logic instructions.

Example 5-10. A Program to Demonstrate the AND Instruction

Object: Display the operands and the result of the AND instruction. Also display the flags in the processor status register.

10	*****			
11	LOGIC	LDA +NUM1	;GET THE FIRST NUMBER.	C000 A5 FB
12		AND +NUM2	;AND IT WITH THE SECOND NUMBER.	C002 25 FC
13		STA +RESULT	;STORE THE RESULT.	C004 85 FD
14		RTS		C006 60
15	*****			
16	START	JSR GETTWO	;GET TWO BYTES FROM KEYBOARD.	C007 20 00 C1
17		JSR LOGIC	;AND THE TWO BYTES.	C00A 20 00 C0
18		JSR DISPLAY	;DISPLAY A AND THE P REGISTER.	C00D 20 6C C1
19		RTS	;RETURN TO BASIC PROGRAM.	C010 60

Load the CAI program and the program in Example 5-10. Call the program in Example 5-10 as a subroutine with this BASIC program:

```
1 SYS 49159
2 GO TO 1
```

Enter two hexadecimal digits to make one byte. The eight-bit number will be displayed on the screen. Enter another two hexadecimal digits to make the second eight-bit operand of the AND instruction. It too will appear on the screen, followed by the result of the AND instruction. The contents of the processor flag register will also be displayed. Pay particular attention to the Z and N flags, since these are the flags that are modified by the logic instructions.

Now that you have the program working, here are some exercises to try using the AND instruction.

1. AND these pairs of numbers and note the result: (\$00, \$FF), (\$01, \$FF), (\$02, \$FF), (\$80, \$FF). What is the result of ANDing a number with \$FF? Which of the previous problems set the Z flag? Why?
2. AND these pairs of numbers and note the result: (\$A5, \$5A), (\$7F, \$80). What do you obtain when you AND a number with its complement? Why was the Z flag set in both of these cases? What was the status of the N flag after each of these operations?
3. The number \$FF has a one in each bit. What number should you choose to AND with \$FF to clear bit seven? Bit three? Bits seven and three? Try your answers with the program in Example 5-10 to see if they are correct.
4. Which of these two problems will cause the N flag to be set after the AND operation: (\$7F, \$80) or (\$C6, \$91)?

In the program in Example 5-10, replace the AND op code \$25 with the op code for the ORA instruction. Refer to Table 5-1 to choose the op code. Now try these problems.

1. ORA the numbers in these pairs: (\$00, \$F5), (\$00, \$7F), (\$00, \$11). What is the result of ORAing a number with zero? Which of the preceding problems set the N flag? Why?
2. Combine these pairs of numbers with the ORA instruction: (\$7F, \$80), (\$EE, \$11), (\$A1, \$5E). What is the result of ORAing a number with its complement?
3. The number \$00 has zeros in all its bits. What number should you ORA with \$00 to set bit seven? Will the N flag be set or clear after this operation? What number should you ORA with \$00 to set bit six? Bit five? Bit three and bit zero? Try these problems using Example 5-10 modified to use the ORA instruction.

Replace the ORA op code with an EOR op code from Table 5-1. Try these problems.

1. Combine these pairs of numbers in an EOR operation: (\$00, \$FF), (\$A5, \$FF), (\$FF, \$E3). Do you get the complement of a number when you EOR it with \$FF?
2. Use the answers to the problems above to EOR a number with its complement. What do you conclude?
3. What number should you EOR with \$A7 to flip bit six? Bit five? Bits zero and seven? Try it with the program in Example 5-10 modified to work with the EOR instruction.

Experiment with these programs until you can anticipate the result obtained by combining any two numbers with an AND, ORA, or EOR instruction, and until you can anticipate the N and Z flag values after the operation.

Here are some additional programming problems.

1. Assemble this program:

```
LDA CRA
AND # $FE
STA CRA
```

where CRA has the address \$DC0E. Locate the program starting at address \$C000. The effect of the program is to clear bit zero in the 6526 CIA control register A. Clearing this bit will disable the keyboard by stopping a counter/timer on the 6526 CIA. Execute the program from BASIC with a SYS 49152 command. Type some letters on the keyboard. What happens? Your computer is now running out of your control. To regain control you must turn off the power and then turn it on again.

2. Write a program to clear bit four in register \$11 of the VIC chip, location \$D011. Also write a program to set this bit. What are the effects of running these programs?
3. Write a program to enable sprites 5 and 1 and disable sprites 2 and 3. The sprite-enable register is at location \$D015. Each sprite is enabled by placing a one in the corresponding position in the sprite-enable register. A sprite is disabled by placing a zero in the corresponding position in the sprite-enable register. Refer to the *Commodore 64 User's Manual* or the *Programmer's Reference Guide* for additional details about creating sprites. Also see Examples 3-6 and 3-7.

6

Branches and Loops

I. Introduction

Without decision-making instructions, the computer would be a far less versatile tool than it is. The "if...then..." type of decision is an important part of almost all programs. For example, in a game program you might have a decision of this type: *if* two sprites collide, *then* make an exploding noise. In applications involving the computer as a controller we find decisions of this type: *if* there are no cars in the left-turn lane, *then* do not turn on the left-turn arrow. Decisions are at the heart of any control application, and the decision-making instructions are of central importance to the material in this chapter. In assembly language, the decision-making instructions are the *branch* instructions, BCC, BCS, BEQ, BNE, BPL, BMI, BVC, and BVS. Refer to Table 6-1 for a brief description and the op code of each branch instruction.

Table 6-1. Op codes for the branch instructions.

<i>Mnemonic</i>	<i>Description</i>	<i>Relative Addressing</i>	<i>Mode Op Code</i>
BCC	Branch on carry clear ($C=0$)		\$90
BCS	Branch on carry set ($C=1$)		B0
BNE	Branch on result not equal zero ($Z=0$)		D0
BEQ	Branch on result equal zero ($Z=1$)		F0
BPL	Branch on plus ($N=0$)		10
BMI	Branch on minus ($N=1$)		30
BVC	Branch on no overflow ($V=0$)		50
BVS	Branch on overflow ($V=1$)		70

Branch instructions in assembly language are very similar to the

IF...THEN GO TO...

command in BASIC. (Although the "GO TO" may be omitted in BASIC, its use here illustrates the point we are trying to make.) This is how the IF...THEN

GO TO... command works: *If the condition is met, then execution continues at the line number that follows the "GO TO."* If the condition is not met, execution continues with the command on the *next line*.

Likewise, in assembly language, a branch instruction tests a certain condition, namely, the status of a specific flag in the processor status register. *If the condition is met, then the program branches (jumps) to another address to continue execution.* If the condition is not met, then execution continues with the instruction following the branch instruction. The flags in the processor status register will, once again, be of considerable interest to us in this chapter.

Instructions or commands in a BASIC program are identified by *line numbers*. Consequently the object of the

IF...THEN GO TO...

command is a line number, and the program branches to this line number. Recall from our discussion in Chapter 1 that machine-language instructions are identified by their location in memory, that is, their address. The program counter, a 16-bit register in the 6510 microprocessor, stores the address of the next program byte to be fetched from memory. A branch is effected by *adding a signed number to the program counter*.

If you have previously used either BASIC or another programming language, then you have probably worked with a programming structure known as a *loop*. A loop allows a specific task to be repeated. Loop counter variables keep track of the number of times the task has been accomplished. In BASIC, the instructions that perform the task are frequently placed between a

FOR I = 1 TO N

instruction and a

NEXT

instruction. In this case, the variable I serves as the loop counter, and N is the number of times the task is to be performed.

A common practice in assembly language is to use the number in either the X or Y register as a *loop counter*. Either the INX or INY instruction is used to *increment* the loop counter. These instructions are analogous to the NEXT instruction in BASIC. On the other hand, sometimes it is advantageous to *decrement* the loop counter using either a DEX or DEY instruction. Refer to Table 6-2 for a description of these instructions and their op codes.

It is also possible to increment or decrement the number in a *memory location* using the INC or DEC instructions, respectively. Thus, a loop counter can be stored in memory as well as in a register in the microprocessor.

How does the loop end? One member of a group of instructions known as a *test* instruction is often used in conjunction with a branch instruction to terminate the loop. The test instructions include the CPX, CPY, CMP, and BIT instructions. Refer to Table 6-2 for their op codes for three of the simplest addressing modes.

Table 6-2. Increment, decrement, and test instruction op codes.

<i>Mnemonic</i>	<i>Description</i>	<i>Op Codes for Each Addressing Mode</i>			
		<i>Immediate</i>	<i>Absolute</i>	<i>Zero-Page</i>	<i>Implied</i>
INX	X+1→X				\$E8
DEX	X-1→X				CA
INY	Y+1→Y				C8
DEY	Y-1→Y				88
INC	M+1→M		\$EE	\$E6	
DEC	M-1→M		CE	C6	
CPX	X-M	\$E0	EC	E4	
CPY	Y-M	C0	CC	C4	
CMP	A-M	C9	CD	C5	
BIT	(See text)		2C	24	

You have many new instructions to learn in this chapter. The complexity inherent in the loop structure demands greater programming skill than straight line programming. The challenge of mastering these new instructions will be rewarded with more interesting and powerful programs. We turn next to a detailed and orderly description of these new instructions.

II. Reviewing the Flags

A review of the flags in the processor status register may be useful before introducing the branch instructions. Only the C, Z, N, and V flags are relevant to this chapter. Refer to Chapter 4, Figure 4-1 in particular, and study the diagram of the processor status register.

- The C (carry) flag is set whenever the result of an arithmetic operation generates a carry. The C flag is cleared if the arithmetic operation does not generate a carry. The C flag is also set or cleared with the shift and rotate instructions that will be described in Chapter 7.
- The Z (zero) flag is set whenever execution of an instruction gives a result of zero. This flag is cleared in the event of a nonzero result.
- The N (negative) flag is set whenever execution of an instruction generates a one in bit seven; otherwise, it is cleared.
- The V (overflow) flag is used only when doing signed-number arithmetic. It is set whenever an arithmetic operation produces a result greater than +127 or less than -128, indicating an overflow into the sign bit; otherwise, it is cleared.

A flag is *modified* by an instruction if it is either set or cleared when the instruction is executed. Refer to the last column in the instruction set summary in Table 2-1 to determine which flags a particular instruction modifies. Examples 6-1 and 6-2 will clarify these ideas.

Example 6-1. Modifying Flags with the LDA Instruction

How does the LDA instruction modify the flags in the processor status register?

Solution: Refer to the instruction set summary in Table 2-1 and observe that the LDA instruction modifies both the Z and N flags. If the operand of the LDA instruction is \$00, then the Z flag will be *set*; otherwise, it will be *cleared*. If the operand of the LDA instruction has a one in bit seven (codes \$80 through \$FF) then the N flag will be *set*; otherwise, it will be *cleared*.

Example 6-2. Modifying Flags with the DEX Instruction

How does the DEX instruction modify the flags in the processor status register?

Solution: Refer to Table 2-1 and observe that the DEX instruction modifies the Z and N flags. If the number in the X register is decremented to zero, then the Z flag will be set; otherwise, it will be cleared. If, after it is decremented, the number in the X register has a one in bit seven (numbers \$80 through \$FF), then the N flag will be set; otherwise, it will be cleared.

This concludes our brief review of the flags.

III. The Branch Instructions

A branch instruction tests the value of a specific bit in the processor status register. Table 6-1 indicates the condition that is tested by each of the eight branch instructions. If the condition tested is *met*, then the program counter is altered, and the branch is said to be *taken*. If the condition tested is *not met*, then the program counter is not altered, and the branch is *not taken*. In the latter case, the program continues by executing the instruction immediately following the branch instruction.

A branch instruction is a two-byte instruction. The first byte is the op code given in Table 6-1. The second byte of a branch instruction is interpreted by the 6510 microprocessor as a *signed number to be added to the program counter if the branch condition is met*. The second byte of a branch instruction is called the *offset*. In other words, if the branch condition is met, the offset is added to the program counter to determine the address of the next program byte. Since the offset is regarded by the microprocessor as a signed number, a *forward branch* is limited to 127 bytes, and a *backward branch* is limited to -128 bytes. Refer to Chapter 4 for a description of signed numbers. Because the change in the program counter is always *relative* to the location of the branch instruction, the addressing mode used by the branch instruction is called *relative addressing*.

Recall from Chapter 1 that the program counter is incremented immediately after the microprocessor fetches a program byte. Thus, by the time the processor is ready to add the offset to the program counter, the value of the program counter corresponds to the address of the op code of the first instruction *after* the branch instruction. It is very important to remember this! This sequence of events is diagrammed in Figure 6-1.

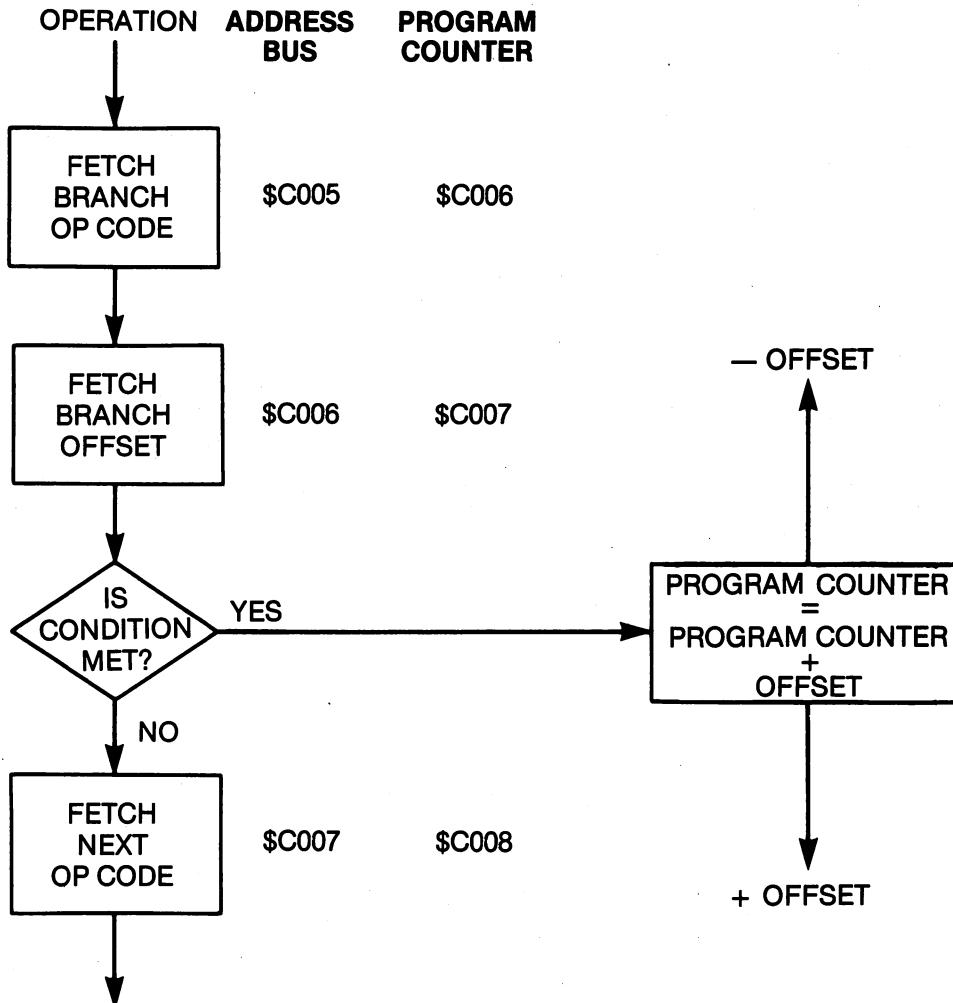


Figure 6-1. Flowchart of a branch instruction.

Assume the branch op code is in the memory location whose address is \$C005 (we will provide such an example in a moment). While the processor fetches the branch op code, the program counter is already being incremented to fetch the next byte of the instruction. The second byte of the branch instruction is the offset. When the offset is fetched, the program counter is incremented again. The program counter now holds the address of the op code

of the instruction *following* the branch instruction. Thus, the program counter will be \$C007. If the branch is to be taken, the processor now adds the offset. Note that it is added to \$C007, *not* to \$C005 or \$C006 where the branch instruction is stored in memory. An offset of \$05, for example, will cause the microprocessor to fetch the next program byte at \$C00C; an offset of \$F9 (-7) will cause the microprocessor to fetch the next program byte at location \$C000.

It is time to present some concrete examples. Refer to the program in Example 6-3 and its flowchart in Figure 6-2. The alarm on the time-of-day (TOD) clock "sounds" by setting bit two in the 6526 CIA interrupt control register (ICR2). This register is located at \$DC0D. Suppose our task is to wait until the alarm sounds. The program in Example 6-3 accomplishes this task. Line 10 sets up the mask to isolate bit two. Line 11 ANDs the mask with the number in the ICR. If the alarm bit is still zero, the result of the AND IRCREG instruction will be zero. The BEQ OFFSET instruction will test the Z flag, find that it is set, and the branch will be taken. The branch offset is \$F9 (-7), so the program branches backward to the address labeled OFFSET. If the alarm bit is set, then the result of the AND IRCREG instruction will be one, the Z flag is clear, the branch will not be taken, and the program will continue with the RTS instruction following the branch instruction. Example 6-3 uses the same addresses as the diagram in Figure 6-1. A flowchart of the program in Example 6-3 is shown in Figure 6-2.

Example 6-3. A Program to Demonstrate a Branch Instruction

Object: Wait in a loop until the TOD clock alarm bit is set.

10	OFFSET	LDA #\$04	; MASK TO ISOLATE BIT 2.	C000	A9	04
11		AND IRCREG	; IS THE ALARM BIT SET?	C002	2D	0D DC
12		BEQ OFFSET	; NO, Z = 1, BRANCH BACK TO OFFSET.	C005	F0	F9
13		RTS	; YES, Z = 0, RETURN.	C007	60	

There are three other points worth mentioning about Example 6-3 before moving on to other examples. First, the *label* of the instruction that is the *destination* of the branch is found in the operand field of the assembly-language program. Thus, in Example 6-3, the label OFFSET is in the operand field of the BEQ instruction. This is standard assembly-language practice. The assembler will calculate the correct offset when given this information.

Second, if you are assembling programs by hand, the offset is most easily calculated by starting with the program byte immediately following the branch instruction and counting \$00, \$01, \$02, ... forward until the destination of the forward branch is reached. For a backward branch, start with the byte following the branch instruction and count \$00, \$FF, \$FE, ... backward until the destination of the backward branch is reached. Thus, in Example 6-3, start counting backward with the byte located at \$C007, and end with the byte at \$C000. Examples 6-4 and 6-5 illustrate the calculation of offsets.

Finally, in the program in Example 6-3, we have encountered our first program *loop*. The program waits in the loop, composed of the LDA #\$04, the AND IRCREG, and the BEQ OFFSET instructions, until the alarm bit is set.

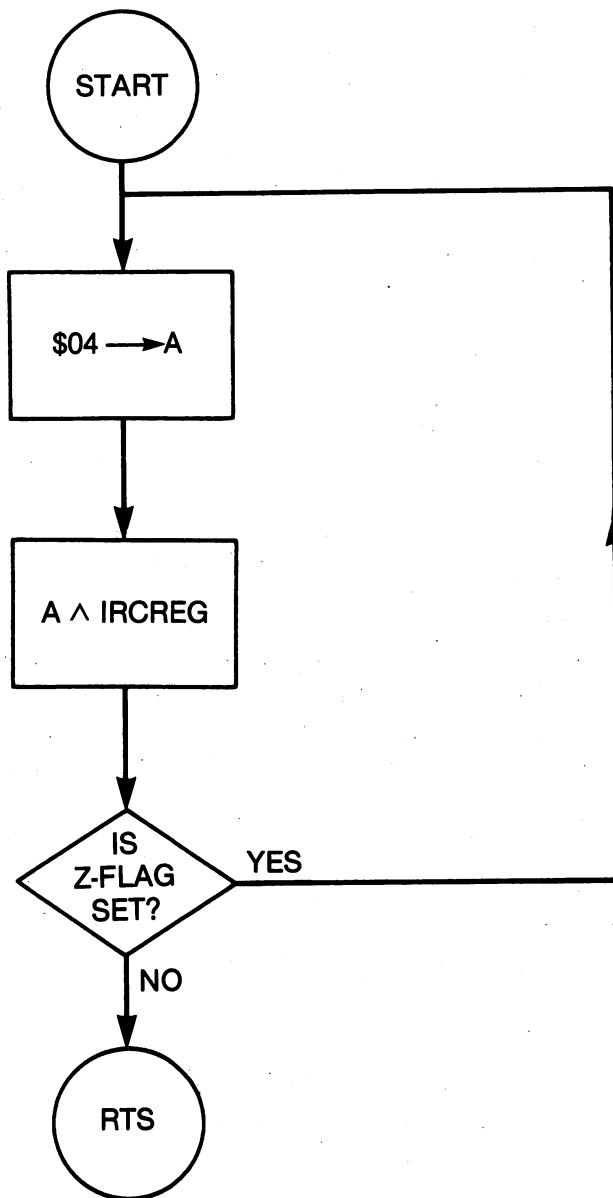


Figure 6-2. Flowchart of the program in Example 6-3.

Example 6-4. Calculation of a Forward Branch

When a program reaches \$C123, it must branch to \$C18F if the N flag is set. What instruction should be located at \$C123 and \$C124?

Solution: A branch-on-minus, BMI, instruction is required. The offset is \$C18F - \$C125 = \$6A. The BMI op code \$30 should be located at \$C123 and the offset \$6A should be located at \$C124.

Example 6-5. Calculation of a Backward Branch

A BCS instruction is located at \$C2FB. If the carry flag is set, the program must branch backward to the op code located at \$C2F0. What is the correct offset for this backward branch?

Solution: The first op code after the BCS instruction is located at \$C2FD. $\$C2FD - \$C2F0 = \$0D$. We need to branch -13 ($\$0D$) bytes. The two's-complement of $\$0D$ is $\$F3$. $\$F3$ is the correct offset. Recall that the two's-complement is found by complementing the number and adding one.

We conclude this section with one more program listed in Example 6-6. The subroutine GETIN is in the Commodore 64 operating system. Subroutines in the operating system do not need to be written, they already exist in ROM, and they can be called by any program. This subroutine reads the keyboard buffer. If a key has been pressed, the subroutine returns with the character in the accumulator. If a key has not been pressed, the subroutine returns with the Z flag set. If the Z flag is set, then the program branches backward to wait for a key to be pressed. If a key is pressed, the program in Example 6-6 outputs the character to the screen using the Commodore 64 operating system subroutine CHROUT.

Example 6-6. A Program to Read the Keyboard and Output the Character

Object: Output the characters typed on the keyboard.

10	GET	JSR GETIN	;GET A CHARACTER FROM THE BUFFER.	C000 20 E4 FF
11		BEQ GET	;WAIT FOR A NON-ZERO RESULT.	C003 F0 FB
12		JSR CHROUT	;OUTPUT THE CHARACTER TO THE SCREEN.	C005 20 D2 FF
13		CLV	;CLEAR THE OVERFLOW FLAG TO FORCE	C008 B8
14		BVC GET	;A BRANCH TO LOOP FOREVER.	C009 50 F5

The really interesting feature of the program in Example 6-6 is the function of the last two instructions. The 6510 instruction set does not include an unconditional branch; however, by first clearing a flag and then branching on the condition that the flag is clear, you can produce an unconditional branch. Notice in Example 6-6 that we clear the V flag with a CLV instruction and then we force a branch with the BVC GET instruction. Also note that we have introduced an *infinite* loop, that is, there is no way to escape from the loop created by the BVC GET instruction. The only way to escape is by pressing the RUN/STOP and the RESTORE keys simultaneously, producing a NMI-type interrupt (see Chapter 9). You can also pull the plug and start over.

IV. Loop Counters

The increment and decrement instructions in Table 6-2 are easily understood. Consider the increment instructions INX, INY, and INC. The number in the X register, Y register, or a memory location M is *incremented by one* when the corresponding instruction is executed. When the number being incremented reaches \$FF, the next increment instruction will produce a zero result. Thus, the counting proceeds as follows:

... \$FD, \$FE, \$FF, \$00, \$01, \$02, ...

In the case of the decrement instructions DEX, DEY, and DEC, the number in the X register, Y register, or a memory location M is *decremented by one* when the corresponding instruction is executed. In this case, the counting proceeds as follows:

... \$02, \$01, \$00, \$FF, \$FE, \$FD, ...

The INX, INY, DEX, and DEY instructions use the implied addressing mode because the location of the operand is implied by the instruction itself. These are, therefore, single-byte instructions. The INC and DEC instructions, on the other hand, modify a number in memory. The program must, therefore, supply information about the address of the memory location where the operand is located. These instructions use, among other addressing modes to be described in Chapter 8, zero-page and absolute addressing.

The INC and DEC instructions do not use the immediate addressing mode because that would imply that a number *in the program* is to be modified, a practice that is strongly discouraged because programs should be capable of being stored in ROM where they cannot be modified. If you study the instruction set summarized in Table 2-1, you will also notice that the immediate addressing mode is not used by the various store (STA, STX, STY) instructions.

To illustrate the use of a loop counter, consider the program in Example 6-7 and its flowchart in Figure 6-3. The program does nothing but waste time, which is why it is called a *delay loop*. Machine-language programs execute so quickly that it is sometimes necessary to introduce delays. The program in Example 6-7 can be modified to give various delays. Notice the use of the DEX instruction and the BNE branch instruction.

To calculate the delay, the number of clock cycles required by each instruction is found from the instruction set in Table 2-1. These are added together to find the total delay. Calculating the delay in clock cycles is frequently tedious and requires great care. For the program in Example 6-7 we find that the delay, expressed in clock cycles N is given by the formula

$$N = 5*X + 13$$

where X is the number placed in the X register before the loop starts. In obtaining this formula, we have taken into account the JSR DELAY instruction (six cycles) required to get to this subroutine and the RTS instruction (six cycles) required to return to the calling program. With X = \$CA, as shown

in Example 6-7, we find $N = 1,023$ clock cycles. Since a clock cycle on the Commodore 64 is approximately 0.977778 microseconds, the delay loop in Example 6-7 gives a delay of approximately 1.000 millisecond. Try to verify these calculations yourself.

Example 6-7. A Simple Delay Loop

Object: Delay approximately 1.000 millisecond.

10	DELAY	LDX #\$CA	;SET UP DELAY FOR 1023 CYCLES.	C200 A2 CA
11	LOOP	DEX	;DECREMENT X. X = X - 1.	C202 CA
12		BNE LOOP	;BRANCH IF X IS NOT ZERO.	C203 D0 FD
13		RTS		C205 60

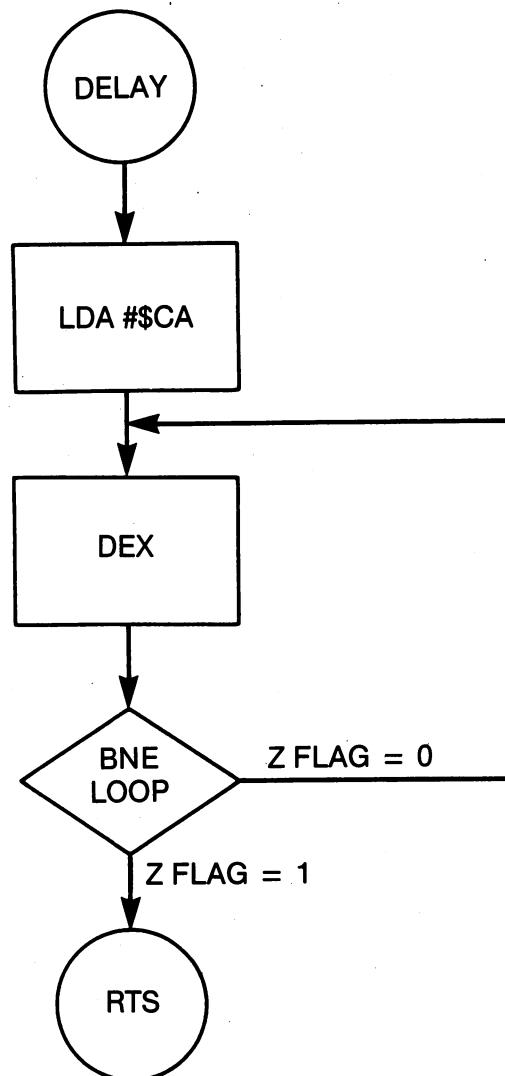


Figure 6-3. Flowchart of the delay loop in Example 6-7.

We turn next to an explanation of the test instructions, after which we will provide programming examples that illustrate both the loop counters and the test instructions.

V. Test Instructions

The *test* instructions are so named because they do not produce results in quite the same way as previously described instructions. In fact, after a test instruction is executed, the *only* code that is modified is the one in the processor status register. In other words, the test instructions are used to set flags, and that is all.

Three of the test instructions identified in Table 6-2 are used to *compare* two numbers. The numbers are compared by subtracting one from the other. The minuend of the subtraction operation is in A, X, or Y, depending, respectively, on whether a CMP, CPX, or CPY instruction is used. The subtrahend of the subtraction operation is in memory, M. In other words, a number in memory is subtracted *from* a number in a 6510 register. *The difference is not stored!*

In contrast to the SBC instruction, where the carry flag enters into the calculation, *the carry flag is not involved* in the subtraction when compare instructions are executed. The carry flag, however, is modified by the *outcome* of the subtraction.

Consider the CMP instruction as an example. Its logical description is A – M, meaning that a number in memory is subtracted from the number in the accumulator.

- If $A > M$, then no borrow occurred so the C flag is *set*. Also, the answer is not zero so the Z flag is *clear*.
- If $A = M$, then no borrow occurred so the C flag is *set*. Also, the answer is zero so the Z flag is *set*.
- If $A < M$, then a borrow was generated so the C flag is *clear*. Also, the answer is not zero so the Z flag is *clear*.

If, in the previous definition of the CMP instruction, you replace A with X or Y, then you have a description of the CPX and CPY instructions, respectively. Notice once again that the logical description of the CMP instruction ($A - M$) *does not involve* the carry flag. The carry flag is modified by the compare instructions, but it is not used by the compare instructions.

Although the N flag is also modified by the compare instructions, in almost all programming problems it is preferable to overlook this fact. Subtracting a smaller unsigned number from a larger unsigned number, for example, does not always produce a positive number (bit seven and the N flag clear). For example, $\$FF - \$7F = \$80$, which has a one in bit seven, signifying a negative result. The negative result, in turn, suggests that we subtracted a larger number from a smaller. You can see that the results can be confusing. A good rule of thumb is to use compare instructions and the carry flag to indicate "greater than" or "less than" relationships between numbers.

Examples 6-8 and 6-9 illustrate typical comparison calculations.

Example 6-8. Illustration of a CMP Calculation

If the number \$BE is in the accumulator and \$2D is the operand of the CMP instruction, how will the C, Z, and N flags be modified by the CMP instruction?

Solution: Since \$BE > \$2D, C will be set and Z will be cleared. Also, since \$BE - \$2D = \$91 and \$91 has a one in bit seven, the N flag will also be set by the CMP instruction.

Example 6-9. Illustration of a CPY Calculation

Assume \$7F is in the Y register and the location referenced by the CPY instruction contains \$F7. Describe the condition of the C, Z, and N flags after the CPY instruction is executed.

Solution: Since \$7F < \$F7, C will be clear and Z will be clear. Since \$7F - \$F7 = \$88 and \$88 has a one in bit seven, N will be set.

Table 6-3 illustrates which branch instructions are required for the various relational possibilities. The branch instruction listed will produce a branch if the relational situation is true.

Table 6-3. Branch instructions for number relationships.

Number	Relationship	Carry Flag (C)	Zero Flag (Z)	Branch Instruction(s)
A = M,	A - M = 0	1	1	BEQ
A ≠ M,	A - M ≠ 0	X	0	BNE
A > M,	A - M > 0	1	0	BCS
A < M,	A - M < 0	0	0	BCC
A = > M,	A - M = > 0	1	X	BCS
A < = M,	A - M < = 0	X	X	BCC BEQ

Notes: 1. An X means the flag could be either clear or set.
 2. If the number relationship is true, the branch instruction(s) will cause the branch to be taken.

The last test instruction to be described is the BIT instruction. The BIT instruction forms the logical AND with the number in the accumulator and a number in memory. It also transfers bit seven of the memory location it tests to the N flag and bit six of the memory location it tests to the V flag in the processor status register. To summarize the BIT instruction:

- If A AND M = 0, then the Z flag will be set; otherwise, it will be cleared.
- M → N; that is, bit seven transfers into the N flag.
- M → V; that is, bit six transfers into the V flag.

It is often true that an LDA instruction will set the same flags as the BIT test. Why use the BIT test? The BIT test does not modify the code in the

accumulator, whereas an LDA instruction will modify that code. The BIT test is used when you want to test specific bits without modifying the code in the accumulator. The calculation in Example 6-10 illustrates a BIT calculation.

Example 6-10. Illustration of a Bit Calculation

If the accumulator contains \$04 and the operand of the BIT instruction is \$42, how will the C, Z, N, and V flags be modified by the execution of the BIT instruction?

Solution: The C flag is not modified by the BIT instruction. Since \$04 AND \$42 = \$00, the Z flag will be set. Since the operand (\$42) of the bit instruction has a zero in bit seven, the N flag will be cleared. Since the operand has a one in bit six, the V flag will be set.

This completes our explanation of the test instructions. Programming examples that illustrate how these instructions are used will follow.

VI. Programming Examples for Branches and Loops

The program in Example 6-11 illustrates INY and CPY instructions. The loop starts with Y = \$21 and ends with Y = \$80. The Y register serves as a loop counter and the CPY instruction is used to terminate the loop. The object of the program is to output part of the Commodore 64 character set to the screen. In this case, the program starts with the character code for the exclamation point, \$21, and ends with the character code \$7F, one of the graphics characters. Determining the values taken by a loop counter is sometimes difficult and worth detailed attention. Try using this program to output a different set of characters. Remember that certain codes may home the cursor or clear the screen.

Example 6-11. A Program to Demonstrate the INY and CPY Instructions

Object: Print a subset of the Commodore 64 character set on the screen.

```
10 ORG      LDY #21          ;START CHARACTER CODES AT $21.      C000 A0 21
11 LOOP     TYA             ;TRANSFER CODE TO THE ACCUMULATOR. C002 98
12         JSR CHRROUT      ;OUTPUT IT WITH COMMODORE 64 ROUTINE. C003 20 D2 FF
13         INY             ;INCREMENT Y. Y = Y + 1.        C006 C8
14         CPY #$80          ;CALCULATE Y - $80 AND SET FLAGS. C007 C0 80
15         BCC LOOP         ;IF Y < $80 THEN GO TO LOOP.   C009 90 F7
16         RTS             ;IF Y = > $80, THEN RETURN.    C00B 60
```

The function of the program in the next example is identical to the program in Example 6-3, but the program makes use of the BIT instruction to accomplish the same task. Refer to Example 6-12 to see how the BIT instruction

is used to wait for the alarm bit to be set on the TOD clock on the CIA. Compare Examples 6-3 and 6-12. Which requires less time? Notice that when the BIT instruction is used, the mask that isolates bit two is loaded only once, whereas in Example 6-3 the mask is loaded each time through the loop. The reason for this is that the BIT instruction does not modify the number in the accumulator. The loop time in Example 6-3 is nine clock cycles, and the loop time in Example 6-12 is seven clock cycles.

Example 6-12. Demonstration of the Bit Instruction

Object: Wait in a loop until the alarm bit is set.

1	;EXAMPLE 6-12		
2	IRCREG	EQU	\$DC0D
3	CHECK	EQU	\$C000
4	;		
5	;		
6	;		
7	;		
8	;		
9	;		
10	CHECK	LDA	#\$04
11	OFFSET	BIT	IRCREG
12		BEQ	OFFSET
13		RTS	
			MASK IS 00000100.
			;IS THE ALARM BIT, IRC3, SET?
			;NO, Z = 1, BRANCH BACK TO OFFSET.
			;YES, Z = 0, RETURN.
			DC0D
			C000
			C000 A9 04
			C002 2C 0D DC
			C005 F0 FB
			C007 60

To test the programs in Examples 6-3 and 6-12, we provide the program in Example 6-13. It sets the alarm for one minute, sets the clock to start at zero, and starts the clock. After starting the clock, the program jumps to the subroutine in Example 6-12 to wait for the alarm bit to be set. After the alarm bit is set, control returns to the program in Example 6-13 and then back to BASIC. Load both Examples 6-12 and 6-13 into memory, then execute Example 6-13 using a SYS 49152 command from BASIC. One minute later you should see BASIC's READY prompt appear. Replace Example 6-12 with Example 6-3 and repeat the experiment.

The only unusual feature in Example 6-13 is the use of the SEI and CLI instructions. Please accept these as necessary to test the example programs: these two instructions will be fully explained in Chapter 9.

Example 6-13. A Program to Set the Alarm and Start the Clock

Object: Set the alarm for one minute after the clock is started.

11	ORG	SEI	;STOP OPERATING SYSTEM INTERRUPTS.	
12		LDA CRB	;SET BIT SEVEN OF CONTROL REGISTER.	
13		ORA #\$80	;	
14		STA CRB	;	
15		LDA #00	;SET ALARM.	
16		STA HRS	;INITIALIZE HOURS REGISTER.	
17		STA SEC	;SET SECONDS.	
18		STA THSSEC	;SET TENTHS OF SECONDS.	
19		LDA #\$01	;SET ALARM FOR 1 MINUTE.	
20		STA MIN	;	
21		LDA #\$7F	;NOW SET TIME.	
			C100	78
			C101	AD 0F DC
			C104	09 80
			C106	8D 0F DC
			C109	A9 00
			C10B	8D 0B DC
			C10E	8D 09 DC
			C111	8D 08 DC
			C114	A9 01
			C116	8D 0A DC
			C119	A9 7F

22	AND CRB		C11B 2D 0F DC
23	STA CRB	;CLEAR BIT SEVEN TO SET TIME.	C11E 8D 0F DC
24	LDA #00	;CLEAR ALL TIME REGISTERS.	C121 A9 00
25	STA HRS		C123 8D 0B DC
26	STA MIN		C126 8D 0A DC
27	STA SEC		C129 8D 09 DC
28	STA THSSEC		C12C 8D 08 DC
29	JSR CHECK	;CHECK THE ALARM.	C12F 20 00 C0
30	CLI	;ENABLE OPERATING SYSTEM INTERRUPTS.	C132 58
31	RTS		C133 60

The CMP, INC, and DEC instructions have not yet been illustrated. The program in Example 6-14 illustrates these instructions and two branch instructions in a graphics application. The program illustrates how a sprite can be moved: in particular, sprite #0 is moved up and down on the screen. It is difficult to analyze a long assembly-language program, so we have divided the program in Example 6-14 into three parts separated by spaces. The first section of the program is identical to the initialization sequence in Example 3-6, and it will not be discussed further.

Example 6-14. A Program to Bounce a Sprite Up and Down

Objective: Move sprite #0 up and down on the screen.

11	ORG	LDA #\$0D	;SPRITE #0 MEMORY POINTER.	C000 A9 0D
12		STA \$07F8		C002 8D F8 07
13		LDA #\$01	;TURN ON SPRITE #0 ON THE VIC.	C005 A9 01
14		STA VIC15		C007 8D 15 D0
15		LDA #\$07	;SELECT A YELLOW BACKGROUND.	C00A A9 07
16		STA VIC21		C00C 8D 21 D0
17		LDA #\$06	;SELECT BLUE FOR THE SPRITE.	C00F A9 06
18		STA VIC27		C011 8D 27 D0
19		STA VIC20	;SELECT A BLUE BORDER COLOR.	C014 8D 20 D0
20		LDX #\$80	;\$80 = 128 WILL BE THE X COORDINATE	C017 A2 80
21		STX VIC00	OF SPRITE #0.	C019 8E 00 D0
22		LDY #\$80	;ALSO CHOOSE 128 FOR THE Y COORDINATE	C01C A0 80
23		STY VIC01		C01E 8C 01 D0
24		;		
25		;		
26	DOWN	INC VIC01	;MOVE SPRITE DOWN.	C021 EE 01 D0
27		JSR DELAY	;SLOW DOWN SPRITE MOVEMENT.	C024 20 3D C0
28		LDA VIC01	;FETCH THE Y COORDINATE.	C027 AD 01 D0
29		CMP #\$E5	;IS SPRITE AT BOUNDARY?	C02A C9 E5
30		BCC DOWN	;NO. BRANCH BACKWARD TO MOVE DOWN.	C02C 90 F3
31	UP	DEC VIC01	;YES, START MOVING UP.	C02E CE 01 D0
32		JSR DELAY	;SLOW DOWN THE SPRITE.	C031 20 3D C0
33		LDA VIC01	;FETCH THE Y COORDINATE.	C034 AD 01 D0
34		CMP #\$32	;IS SPRITE AT THE TOP BOUNDARY?	C037 C9 32
35		BCS UP	;NO. BRANCH BACKWARD TO MOVE UP.	C039 B0 F3
36		BCC DOWN	;YES. START DOWN AGAIN.	C03B 90 E4
37		;		
38		;		
39	DELAY	LDX #00	;DELAY LOOP. START X AT ZERO.	C03D A2 00
40	LOOP	DEX	;X = X - 1.	C03F CA
41		BNE LOOP	;LOOP UNTIL X = 0 AGAIN.	C040 D0 FD
42		RTS		C042 60

The second part of the program in Example 6-14 is where the sprite is moved. The y-coordinate of sprite #0 is controlled by the number in register VIC01. INCrementing this number moves the sprite down, DECrementing this number moves the sprite up. As long as the y-coordinate remains between

\$32 and \$E5, the entire sprite remains on the screen. CMP instructions are used to keep the y-coordinate between these two values. Be sure you understand how the CMP instructions affect the carry flag, because it is the status of the carry flag that is tested in the program in Example 6-14 in order to decide whether to move the sprite up or down. Refer to the previous section and study the program and the comments until you understand how the program works. Be sure to create a sprite with the program in Example 3-7 before you try to execute the program in Example 6-14, or you will not see the sprite at all.

The last part of the program in Example 6-14 is the delay loop previously described in Example 6-7 and Figure 6-3. Without a delay, the sprite moves so quickly that it is very difficult to observe. It is educational to write a BASIC program that moves the sprite up and down and compare the speed of the BASIC sprite with our machine-language sprite.

The program in Example 6-15 illustrates at least two new concepts in addition to reinforcing concepts already learned. Once again, the program has been divided into three parts. The first part is the initialization sequence for the SID chip, Voice #1, given in Example 3-5. It will not be discussed further. The last part of the program is a delay loop that slows the middle part of the program.

It is the middle section of the program that interests us. This section causes the SID chip to output its entire range of frequencies. The frequency of Voice #1 is controlled by the numbers in two eight-bit, write-only registers located at \$D400 and \$D401. \$D400 is the least-significant byte (LSB) of the frequency and \$D401 is the most-significant byte (MSB) of the frequency. A register that cannot be read (with an LDA instruction for example) is called a *write-only* register. All but four of the SID registers are write-only registers. Write-only registers preclude the use of the INC and DEC instructions, among others, because these instructions require that the microprocessor first *read* the number stored in a register or memory location, then *modify* the number, and then *write* the new number back into the register. The INC and DEC instructions are part of a group of instructions called *read-modify-write* instructions, and they cannot be used in conjunction with write-only registers. Any instruction that uses a read operation will not work with a write-only register.

To increment the frequency, we choose two other read/write memory locations to hold the LSB and the MSB of the frequency. These locations, \$00FB and \$00FC in Example 6-15, can be incremented or decremented, and their contents can be transferred to the corresponding frequency registers in the SID chip with an STA instruction. That is the approach we use in Example 6-15. You now know how to deal with write-only registers.

Example 6-15. Incrementing a Two-Byte Counter

Object: Sample all the frequencies available on the SID chip by incrementing the two-byte frequency register.

13	INITLZ	LDA #\$0F	; \$0F CORRESPONDS TO MAXIMUM VOLUME.	C000 A9 OF
14		STA SID18	; SET THE VOLUME ON SID.	C002 8D 18 D4
15		LDA #\$00	; SET THE ATTACK/DECAY RATE	C005 A9 00
16		STA SID05	; FOR VOICE #1 ON SID.	C007 8D 05 D4
17		LDA #\$F0	; SET THE SUSTAIN LEVEL FOR VOICE #1	C00A A9 F0

```

18      STA SID06      ;ON SID.                                C00C 8D 06 D4
19      LDA #$21       ;SELECT A RAMP WAVEFORM FOR VOICE #1   C00F A9 21
20      STA SID04      ;AND START THE SOUND.                  C011 8D 04 D4
21      LDA #00         ;START FREQUENCY AT ZERO.                 C014 A9 00
22      STA +LOFREQ    ;BYTE TO HOLD LOW FREQUENCY.             C016 85 FB
23      STA +HIFREQ    ;BYTE TO HOLD HIGH FREQUENCY.            C018 85 FC
24      ;
25      ;
26  LOOP  LDA +T.OFREQ  ;GET LOW FREQUENCY BYTE.                C01A A5 FB
27      STA SID00      ;STORE IN LOW FREQUENCY REGISTER.          C01C 8D 00 D4
28      LDA +HIFREQ    ;GET HIGH FREQUENCY BYTE.                 C01F A5 FC
29      STA SID01      ;STORE IN HIGH FREQUENCY REGISTER.          C021 8D 01 D4
30      JSR DELAY     ;DELAY LONG ENOUGH TO HEAR TONE.           C024 20 30 C0
31      INC +LOFREQ    ;INCREMENT TWO-BYTE COUNTER.               C027 E6 FB
32      BNE PAST       ;BRANCH PAST UNLESS LOFREQ = 0.            C029 D0 02
33      INC +HIFREQ    ;IF LOFREQ = 0, THEN INC HIFREQ.            C02B E6 FC
34  PAST  BNE LOOP    ;STAY IN LOOP UNTIL ALL POSSIBLE          C02D D0 EB
35      RTS           ;FREQUENCIES HAVE BEEN GENERATED.           C02F 60
36      ;
37      ;
38      ;DELAY SUBROUTINE
39  DELAY  LDX #$FF    ;MAKE A DELAY LOOP.                   C030 A2 FF
40  LOAF   DEX         ;X = X - 1.                      C032 CA
41      BNE LOAF      ;LOOP UNTIL X = 0.                  C033 D0 FD
42      RTS           C035 60

```

The second new concept illustrated by Example 6-15 is how to increment a two-byte counter. The locations LOFREQ at \$00FB and HIFREQ at \$00FC are employed to store a single 16-bit number, namely, the frequency of the SID chip. Program lines 31 through 34 increment this 16-bit number. Notice that the MSB is incremented only when the LSB passes through zero. That is because counting proceeds in this manner:

1, ..., \$FE, \$FF, \$100, \$101, ..., \$1FF, \$200, \$201, ..., \$FFFF

Notice that the only time the MSB is incremented is when the LSB is \$00. After the two-byte number in LOFREQ and HIFREQ is incremented, it is stored in the SID registers by the instructions on lines 26 through 29.

Be sure to clear all of the SID registers (see Example 3-5) before executing the program in Example 6-15. Experiment with different values for the delay loop constant \$FF on line 39 in Example 6-15.

The next programming example shows how a loop structure can be used to measure time intervals, in this case the time between a visual stimulus and your response to this stimulus. The programs in Examples 6-16 and 6-17 also illustrate how machine-language and BASIC programs can be used to complement each other. Again, the assembly-language program in Example 6-17 has been divided into three parts, an initialization sequence, a timing loop, and a sequence of instructions that concludes the program and puts the Commodore 64 operating system in operation once again. We are principally interested in the timing sequence in the middle of the program in Example 6-16, and a flowchart for this part of the program is shown in Figure 6-4. Refer to Examples 6-16 and 6-17. The machine-language program is called from the BASIC program by the SYS 49152 instruction on line 25 of the BASIC program. Here is what happens. The machine-language program blanks the screen by clearing bit four in the VIC control register at \$D011. As soon as the screen is blanked, the timing loop starts. The timing loop is terminated when

you press the button on paddle Y of your game paddle. The loop counters are stored and, after putting the operating system back together, program control returns to the BASIC program where the loop counters are used to calculate the time it took you to respond to the screen going blank. In other words, these two programs allow you to measure your reaction time to a visual stimulus.

The BASIC program provides written instructions to the user, and it and calculates the time. It also waits a random length of time before calling the machine-language subroutine. Refer to lines 5 through 20 in Example 6-17. The time is calculated and printed with lines 30 to 50 in the BASIC program. Line 30 calculates the number of clock cycles that elapsed between the screen going blank and the paddle button being pressed. Line 40 converts this number to seconds by dividing by the clock frequency, 1,022,727 Hz. If you wait longer than about 0.7 seconds, the loop counter overflows and returns to zero. This contingency is handled by lines 35 and 70 in the BASIC program.

Example 6-16. A Program to Measure a Time Interval

Object: Measure the time interval between the events "screen is blanked" and "paddle fire button is pressed." Use Control Port 2.

```

10 ORG      SEI          ;DISABLE OPERATING SYSTEM INTERRUPTS. C000 78
11          LDA DDRA      ;SAVE CONTENTS OF DDRA TO RETURN      C001 AD 02 DC
12          STA +$02      ;OPERATING SYSTEM TO NORMAL.           C004 85 02
13          LDA #$00      ;MAKE PORT A, PAD, AN INPUT PORT BY   C006 A9 00
14          STA DDRA      ;CLEARING DDR FOR PORT A.          C008 8D 02 DC
15          LDX #00      ;INITIALIZE LOOP COUNTERS.           C00B A2 00
16          LDY #00      ;                           C00D A0 00
17          LDA #$EF      ;MASK BIT 4.                         C00F A9 EF
18          AND VICTRL    ;CLEAR BIT 4 TO BLANK SCREEN        C011 2D 11 D0
19          STA VICTRL    ;TO SIGNAL START OF TIMING.         C014 8D 11 D0
20          ;
21          ;
22          LDA #$08      ;ISOLATE PADDLE Y BUTTON.          C017 A9 08
23 LOOP      BIT PAD      ;IS BUTTON PRESSED? BIT 3 = 0?       C019 2C 00 DC
24          BEQ OUT      ;IF SO, BRANCH OUT OF LOOP.        C01C F0 06
25          INX          ;IF NOT, GO THROUGH THE TIMING LOOP. C01E E8
26          BNE LOOP      ;RETURN TO CHECK BUTTON.          C01F D0 F8
27          INY          ;THEN GO THROUGH THE Y LOOP.       C021 C8
28          BNE LOOP      ;RETURN TO CHECK BUTTON.          C022 D0 F5
29 OUT       STX +$FB      ;STORE THE X COUNTER.            C024 86 FB
30          STY +$FC      ;STORE THE Y COUNTER.            C026 84 FC
31          ;
32          ;
33          LDA #$10      ;TURN ON THE SCREEN AGAIN.        C028 A9 10
34          ORA VICTRL    ;                           C02A 0D 11 D0
35          STA VICTRL    ;GET DDRA CONTENTS BACK.          C02D 8D 11 D0
36          LDA +$02      ;PUT OPERATING SYSTEM BACK TO NORMAL. C030 A5 02
37          STA DDRA      ;ENABLE INTERRUPTS AGAIN.         C032 8D 02 DC
38          CLI          ;RETURN TO BASIC TO CALCULATE TIME. C035 58
39          RTS          ;                           C036 60

```

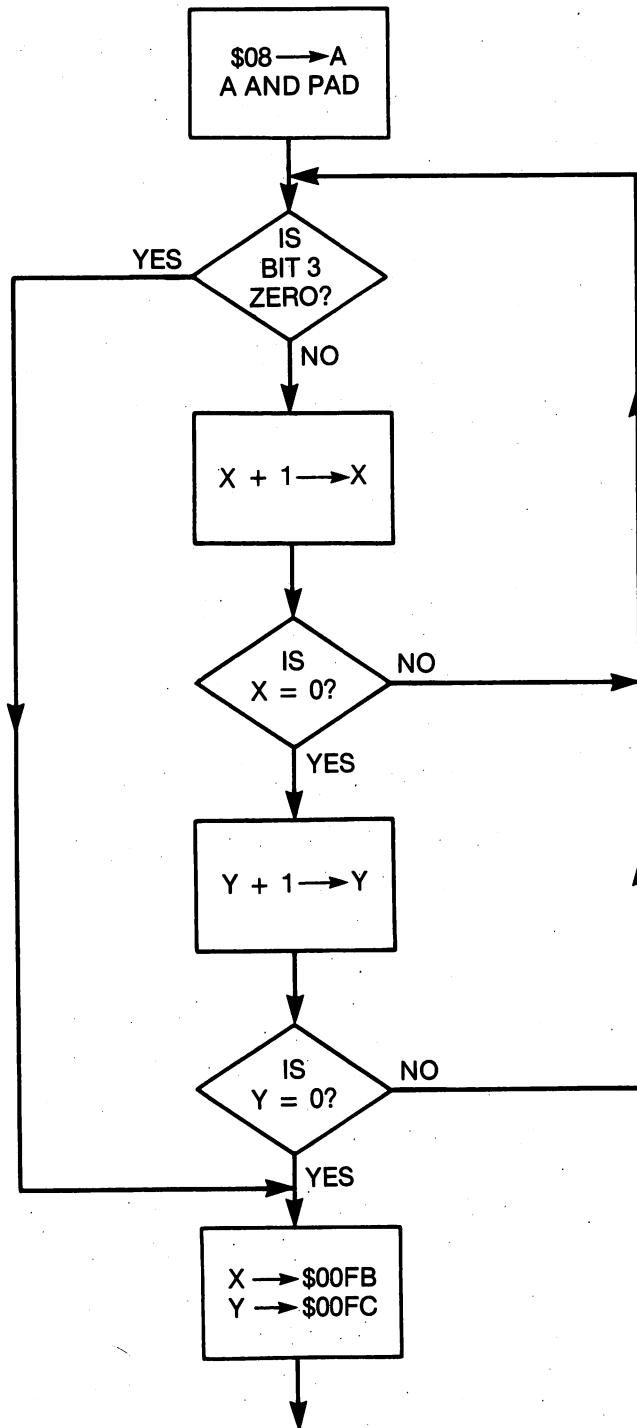


Figure 6-4. Flowchart of the timing loop in Example 6-16.

Example 6-17. A BASIC Program to Call the Machine-Language Program in Example 6-16

READY.

```

4 REM EXAMPLE 6-17
5 PRINT "<CLR>"
10 PRINT "PRESS THE PADDLE BUTTON WHEN"
11 PRINT "THE SCREEN GOES BLANK."
15 Y=1000+6000*RND(0)
20 FOR I=0 TO Y:NEXT
25 SYS 49152
30 T=11*PEEK(251)+2820*PEEK(252)
35 IF T = 0 THEN 70
40 T=T/1022727:T=INT(T*1000+.5)/1000
45 PRINT "<HOM><DWN><DWN><DWN><RHT><RHT><RHT><RHT>"; 
50 PRINT "REACTION TIME = ";T; "SECONDS.      "
60 GO TO 15
70 PRINT "YOU'RE TOO SLOW, TRY AGAIN."
80 GO TO 10

```

READY.

READY.

How does the machine-language loop measure time? Refer to the flowchart in Figure 6-4. The LDA #\$08 and BIT PAD instructions on lines 22 and 23 of the program in Example 6-16 isolate bit three of CIA input port A. The bit value of bit three is zero if the paddle button is pressed; otherwise, it is one. As soon as the button is pressed, therefore, the program will branch out of the timing loop that follows the BEQ OUT instruction. The timing loop increments both the X and Y registers. The X register is incremented in the *inside* loop while the Y register is incremented in the *outside* loop. Each inside loop requires 11 clock cycles. This includes the BIT PAD, BEQ OUT, INX, and BNE LOOP instructions. Each outside loop means the inside loop has executed 256 times. Thus, each outside loop amounts to 256*11 cycles, plus four more cycles required by the INY and the second BNE LOOP instructions. The total number T of clock cycles is, therefore,

$$T = 11*X + 2820*Y$$

This result explains the constants on line 30 of the BASIC program.

Load both programs into memory and RUN the BASIC program. Connect your game paddles to Control Port 2. Only one of the two paddle fire buttons will function, and you can determine which it is by trial and error. Now you are ready to measure your reaction time. Try it, then study the assembly-language program to understand as much of it as possible. The ability to make timing measurements is a powerful tool, and we shall return to this topic again in Chapter 12.

We conclude our sample programs with the two short but valuable routines given in Examples 6-18 and 6-19. The flowcharts for these programs are shown in Figures 6-5 and 6-6, respectively. To output information to an output device, for example, the screen of the video monitor, disk drive, or modem, the information is usually *coded* in a particular way. The code most frequently used to represent numbers, letters, and punctuation marks is the *American Standard Code for Information Interchange*, identified by its acronym, ASCII. These codes are identified in your *Commodore 64 User's Guide*, page 135. In this discussion we are concerned only with the codes for the hexadecimal digits

0, 1, 2, ..., 9, A, B, ..., E, F

These are summarized in Table 6-4.

Table 6-4. ASCII codes for the hexadecimal digits.

<i>Hexadecimal Number</i>	<i>ASCII Representation</i>
\$0	\$30
1	31
2	32
3	33
4	34
5	35
6	36
7	37
8	38
9	39
A	41
B	42
C	43
D	44
E	45
F	46

Example 6-18. Hex-to-ASCII Conversion Routine

Object: Convert a one-digit hexadecimal number in the accumulator to its ASCII representation.

10 HEXCII	CMP #\$0A	; IS THE HEX DIGIT => 9?	C800 C9 0A
11	BCC AROUND	;NO.	C802 90 02
12	ADC #\$06	;YES. ADD \$07 (CARRY + \$06).	C804 69 06
13 AROUND	ADC #\$30	;ADD \$30 TO CONVERT TO ASCII.	C806 69 30
14	RTS		C808 60

Example 6-18 converts a one-digit hexadecimal number in the accumulator to its ASCII representation and returns with the ASCII code in the accumulator. When studying Table 6-4 you will see that a number from \$0 to \$9 can be converted to its ASCII code by adding \$30. However, a number from \$A to \$F is converted to ASCII by adding \$37. Thus, a routine to convert a

one-digit hexadecimal number to ASCII must first distinguish between the numbers represented by ordinary numerals and the numbers represented by letter numerals A to F. This is the function of the CMP #\$0A instruction on

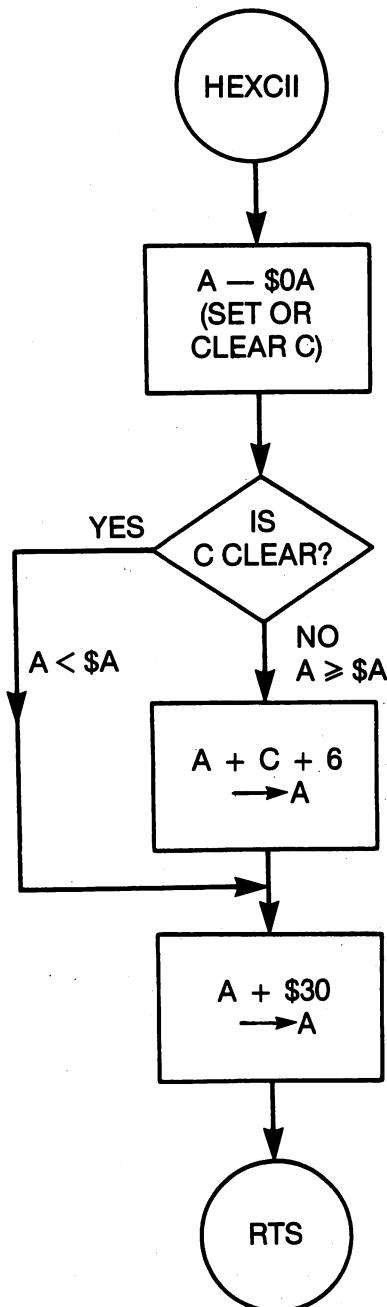


Figure 6-5. Flowchart of the hexadecimal-to-ASCII conversion routine.

line 10 of Example 6-18. If the number is less than ten (\$0A), \$30 is added. If the number is between 10 and 15, \$37 is added. Observe how \$37 is added. The \$30 must be added in either case, so this instruction is put last. If the

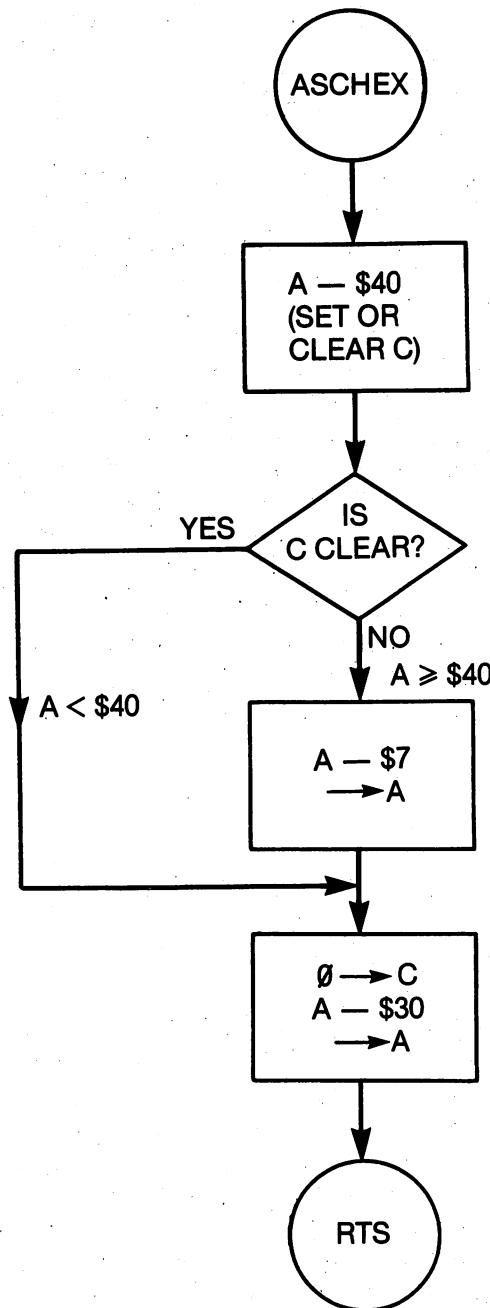


Figure 6-6. Flowchart of the ASCII-to-hexadecimal conversion routine.

number to be converted is greater than nine, then the BCC AROUND branch will not be taken. To add seven, we add \$06 plus the carry flag, set by the CMP #\$0A instruction.

Example 6-19 converts an ASCII code for a hexadecimal digit into the digit itself, and then returns with the single-digit number in the accumulator. In studying Table 6-4 you will see that this process can be accomplished for the numbers \$0 to \$9 by subtracting \$30 from the ASCII representation. You will also observe that the numbers \$A to \$F can be derived from their ASCII representations by subtracting \$37. Can you understand how the program in Example 6-19 does this? The CMP instruction is used to set or clear the carry flag, depending on whether the ASCII code is greater than or equal to \$39, or less than \$39. Also notice the use of the branch instruction to bypass the SBC #\$07 instruction. Study the flowchart in Figure 6-6 for additional details.

Example 6-19. ASCII-to-Hex Conversion Routine

Object: Convert the ASCII representation of a hexadecimal digit to the corresponding number, and place it in the accumulator.

10	ASCHEX	CMP #\\$40	; IS THE CODE < \\$40?	C809 C9 40
11		BCC SKIP	;YES.	C80B 90 02
12		SBC #\\$07	;NO, SUBTRACT \\$07 AND THEN	C80D E9 07
13	SKIP	SEC	;SUBTRACT \\$30 TO CONVERT TO A	C80F 38
14		SBC #\\$30	;HEXADECIMAL DIGIT FROM \\$0 - \\$F.	C810 E9 30
15		RTS		C812 60

VII. Summary

A branch instruction tests the value of a specific flag in the processor status register. If the flag has the value for which it was tested, then the branch instruction adds a signed number to the program counter to produce either a forward or backward branch. If the flag does not have the value for which it was tested, then execution continues with the instruction following the branch instruction. The N, V, Z, and C flags can be tested by branch instructions for values of either zero or one, requiring a total of eight branch instructions. A branch instruction is always represented in a flowchart by a diamond-shaped decision box.

Branch instructions are frequently used with increment or decrement instructions to implement loops, a program structure that allows repetitive execution of a task. Loops are also used to cause a delay, or to make timing measurements. The increment or decrement instructions serve as loop counters.

Test instructions are used to modify flags in the processor status register. The compare instructions set flags with a subtraction operation, and are used to indicate "greater than," "equal to," or "less than" conditions. The BIT instruction is used to test the value of a specific bit in a memory location. When followed by a branch instruction, the BIT instruction is used to determine

whether or not to take a particular course of action, depending on whether a specific bit is set or clear. None of the test instructions modify either the number in the accumulator or numbers in memory. The test instructions are used in conjunction with branch instructions to make decisions that affect the flow of the program.

VIII. Exercises

1. A BPL \$FB instruction is located at \$102D. At what address will the processor obtain the next byte of the program if the N flag is clear? If the N flag is set?
2. What will be the status of the Z flag, the N flag, and the C flag after executing a CMP #\$6F instruction if the number \$FF is in the accumulator?
3. What should the branch offset be if a program is to branch to location \$AF5B, when the branch instruction is located at addresses \$AF25 and \$AF26?
4. Write a program segment that waits in a loop until a sprite-data collision involving sprite #0 occurs. If such a collision occurs, bit zero in VIC register \$1F is set?
5. Modify the program in Example 6-16 to use the button on paddle X rather than paddle Y. Paddle X uses bit two rather than bit three.
6. Modify the program in Example 6-14 to move the sprite horizontally between x-coordinates \$18 and \$FF. Then modify the program so the sprite moves in both the x- and y-coordinates at once.
7. Modify the program in Example 6-16 to use a joystick button rather than a paddle button.
8. Make a flowchart for lines 26 to 35 in Example 6-15.
9. Modify the delay loop in Example 6-7 to use an INX instruction instead of a DEX instruction; then find an expression for the number of clock cycles required to execute the loop.
10. Rewrite the program in Example 6-15 so that the SID outputs a tone whose frequency decreases rather than increases.

Shift and Rotate Routines

I. Introduction

There are two shift instructions, ASL and LSR, and two rotate instructions, ROL and ROR. The uses of these instructions are not immediately obvious, but they turn up in a surprisingly large number of routines. The instructions are easy to understand, but the routines in which they are found are generally complex, and in this chapter you will find that the complexity of the programs increases as we proceed.

The op codes of these instructions for the addressing modes that we will use in this chapter are given in Table 7-1. The shift instructions are diagrammed in Figure 7-1 and the rotate instructions are diagrammed in Figure 7-2.

Table 7-1. Op codes for the instructions in Chapter 7.

<i>Mnemonic</i>	<i>Op Codes for Each Addressing Mode</i>		
	<i>Absolute</i>	<i>Zero Page</i>	<i>Accumulator</i>
ASL	\$0E	\$06	\$0A
ROL	2E	26	2A
LSR	4E	46	4A
ROR	6E	66	6A

II. Definitions of the Shift and Rotate Instructions

We begin our study of the shift and rotate instructions with their definitions:

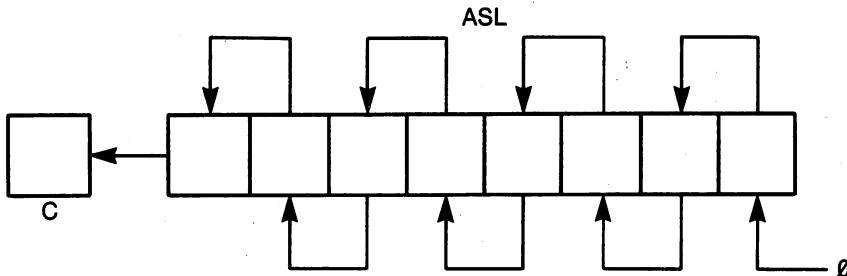
- **ASL—Arithmetic Shift Left:** Each bit in the operand is moved one bit position to the *left*. Zero is shifted into bit zero, and bit seven is shifted into the carry flag. Refer to Figure 7-1.
- **LSR—Logical Shift Right:** Each bit in the operand is moved one bit position to the *right*. Zero is shifted into bit seven, and bit zero is shifted into the carry flag. Refer to Figure 7-1.

- ROL—Rotate One Bit Left: This instruction is identical to the ASL instruction, *except that* the carry flag, rather than zero, is shifted into bit zero. Refer to Figure 7-2.
- ROR—Rotate One Bit Right: This instruction is identical to the LSR instruction, *except that* the carry flag, rather than zero, is shifted into bit seven. Refer to Figure 7-2.

It is obvious that these instructions modify the carry flag because a bit from the operand is always shifted or rotated into the carry flag. These instructions also affect the N and Z flags. If a one is shifted into bit seven, the N flag is set; otherwise, it is cleared. If, as a result of a shift or rotate instruction, the operand becomes zero, then the Z flag will be set; otherwise, it will be cleared. The exercises at the end of the chapter will provide you with a program to demonstrate each of these instructions so that you can see what happens to both the operand and the flags when the shift and rotate instructions are executed.

Refer again to Table 7-1 and you will see that these instructions have a unique addressing mode available to them. This is the *accumulator addressing mode*. In this mode, a code in the *accumulator* can be shifted or rotated. The notation used to indicate the accumulator addressing mode when writing assembly-language programs is

ASL A



ASL

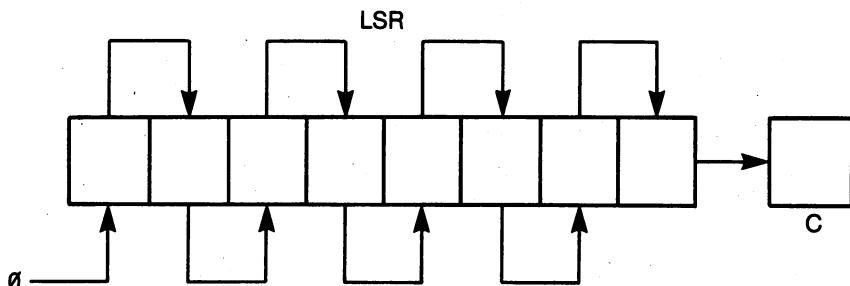


Figure 7-1. Diagram of the ASL and LSR instructions.

where we have used the ASL instructions as an example. Of course, these instructions also work when the operand is in memory, which is to say that they also have absolute and zero-page addressing modes.

It is interesting to study the similarities and differences in the op codes in Table 7-1. For example, it should be clear that the 6510 microprocessor can identify the addressing mode of these instructions by the least-significant nibble of the instruction, while the most-significant nibble is 0, 2, 4, or 6 for the ASL, ROL, LSR, and ROR instructions, respectively.

III. Programs That Use the Shift and Rotate Instructions

With the exception of the exercises, the remainder of this chapter will be devoted to illustrating the use of the shift and rotate instructions in assembly-language programs.

A. Code shift

The program in Example 7-1 shifts a code from one memory location to another using shift and rotate instructions. Although this is an unusual way

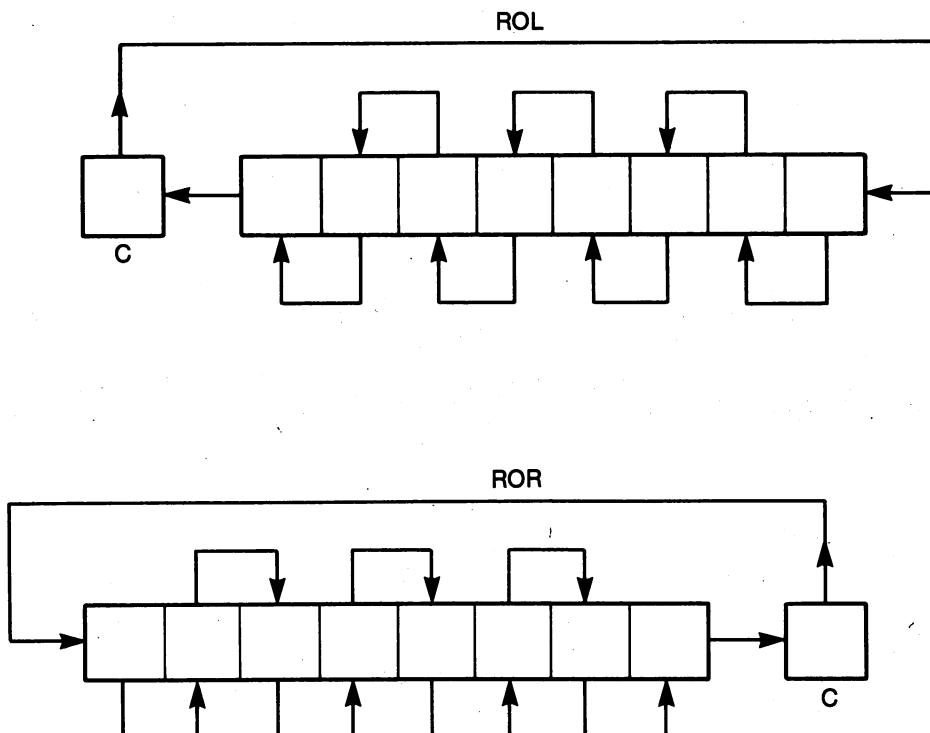


Figure 7-2. Diagram of the ROL and ROR instructions.

to move a code from one location to another if your objective is to move codes, the technique is used within more complex routines, and you will see it again in our multiplication program in Example 7-9.

Example 7-1. Using the ASL and ROL Instructions

Object: Shift the code in location \$00FB one bit at a time into location \$C000.

10	START	LDX #08	; X CONTAINS LOOP COUNTER.	C001 A2 08
11	LOOP	ASL +NUM1	; SHIFT CODE ONE BIT LEFT.	C003 06 FB
12		ROL NUM2	; ROTATE LEFT INTO NUM2 LOCATION.	C005 2E 00 C0
13		DEX	; REPEAT EIGHT TIMES.	C008 CA
14		BNE LOOP		C009 D0 F8
15		RTS		C00B 60

It is important to understand exactly how this program works. On line 10, a counter to keep track of the number of shifts is initialized to count eight shifts. The ASL + NUM1 instruction on line 11 shifts bit seven of the code in location NUM1 into the carry flag. A zero is shifted into bit zero of NUM1, but this is not important for our purposes. Next, the ROL NUM2 instruction rotates the carry flag into the least-significant bit, bit zero, of location NUM2. Thus, the ASL + NUM1 instruction places *one bit* of NUM1 in the carry flag. The ROL NUM2 instruction moves the one-bit carry flag into NUM2. After looping through these instructions eight times, the code that was in NUM1 will now be in NUM2, and the routine is finished.

Place a number in NUM1 with a POKE 251 instruction, then use the SYS 49153 command to execute the program in Example 7-1. Now use a PEEK 49152 instruction to see if the code was moved from location 251 (\$00FB) to location 49152 (\$C000). What do you expect to find in location \$00FB after the program has executed? Why? PEEK in this location to see if your hypothesis is correct. Would the program work if you replaced the ASL and ROL instructions with LSR and ROR instructions, respectively?

B. Displaying the code in a memory location

The program in Example 7-2 displays, in binary, the code found in memory location \$0002 by printing on the screen the bit value of each bit, starting with the most-significant bit on the left and ending with the least-significant bit on the right. Recall from Chapter 6 that numbers are printed on the screen by sending the ASCII code for the number to subroutine CHROUT, an operating system subroutine. The ASCII code must be in the accumulator when the JSR CHROUT instruction is executed. Since, in the present case, we are going to display eight bits of a binary number, we are only interested in the ASCII codes for zero and one, \$30 and \$31, respectively. If a bit in the memory location is one, then we must send the \$31 code to subroutine CHROUT. If a bit in the memory location is zero, then we must send the \$30 code to subroutine CHROUT. A code must be output for each of the eight bits in the memory location, starting with the most-significant bit.

Example 7-2. Displaying the Code in a Memory Location in Binary

Object: Display the number in location \$0002 in binary.

10	DISPLA	LDX #08	;COUNT EIGHT BITS.	C000 A2 08
11	LOOP	ROL +NUMBER	;ROTATE NUMBER INTO CARRY.	C002 26 02
12		LDA #00	;CLEAR ACCUMULATOR.	C004 A9 00
13		ADC #\$30	;CONVERT TO ASCII.	C006 69 30
14		JSR CHROUT	;OUTPUT THE CODE FOR 1 OR 0.	C008 20 D2 FF
15		DEX		C00B CA
16		BNE LOOP	;LOOP TO GET ALL 8 BITS.	C00C D0 F4
17		RTS		C00E 60

Notice how the program in Example 7-2 accomplishes the objective. The most-significant bit in NUMBER is shifted into the carry flag and then added to \$30. If a one was shifted into the carry flag, then after addition the ASCII code for one, \$31, is in the accumulator. If a zero was shifted into the carry flag, then after addition the ASCII code for zero, \$30, is in the accumulator. Subroutine CHROUT changes the ASCII code to a character on the screen. The program then loops to get the remaining bits in location \$0002 and display their values.

C. Printing a byte on the screen

To test and debug programs it is useful to have a routine that will print the number in the accumulator. What must be accomplished? An eight-bit number consists of two four-bit nibbles. The least-significant nibble occupies bits zero through three, while the most-significant nibble occupies bits four through seven. Each nibble is represented by one hexadecimal digit. We would like to output the most-significant hexadecimal digit followed by the least-significant hexadecimal digit. For example, suppose the number \$F3 is in the accumulator. There will be a \$3 (0011) in the least-significant nibble, and there will be a \$F (1111) in the most-significant nibble. We would like our routine to print an "F" followed by a "3."

If necessary, refer to the previous chapter or recall that in Example 6-18 we have a routine to convert a hexadecimal digit to its ASCII representation. To output "F3" we must first shift \$F in the most-significant nibble to the least-significant nibble. Then this number must be converted to its ASCII representation by calling the subroutine in Example 6-18, called HEXCII. After converting the nibble to ASCII, we can output it with subroutine CHROUT. Next we must get the \$3, convert it to ASCII, and send it to subroutine CHROUT. Example 7-3 accomplishes this objective.

Notice that the program in Example 7-3 has been divided into three parts. The first part of the program stores the number in the accumulator, shifts the most-significant nibble to the least-significant nibble with four LSR instructions, uses subroutine HEXCII in Example 6-18 to convert this number to ASCII, and then uses subroutine CHROUT to print the digit on the screen. Notice that the LSR instruction is used in its accumulator addressing mode.

Example 7-3. Print the Byte in the Accumulator

Object: Convert the number in the accumulator to two hexadecimal digits and output these digits to the screen.

10	PRBYTE	STA +TEMP	;TEMPORARILY STORE NUMBER.	C813 85 02
11		LSR A	;SHIFT THE NUMBER IN THE	C815 4A
12		LSR A	;ACCUMULATOR FOUR BITS TO THE RIGHT.	C816 4A
13		LSR A	;MOST SIGNIFICANT NIBBLE IS ZERO.	C817 4A
14		LSR A	;NUMBER IS NOW IN LOW NIBBLE.	C818 4A
15		JSR HEXCII	;CONVERT NUMBER TO ASCII.	C819 20 00 C8
16		JSR CHROUT	;OUTPUT ITS CODE TO THE SCREEN.	C81C 20 D2 FF
17	;			
18	;			
19		LDA +TEMP	;GET ORIGINAL NUMBER BACK IN A.	C81F A5 02
20		AND #\$0F	;MASK MOST SIGNIFICANT NIBBLE.	C821 29 0F
21		JSR HEXCII	;CONVERT SECOND DIGIT TO ASCII.	C823 20 00 C8
22		JSR CHROUT	;OUTPUT ITS CODE.	C826 20 D2 FF
23	;			
24	;			
25		LDA #\$20	;OUTPUT ASCII SPACE BY SENDING	C829 A9 20
26		JSR CHROUT	;SPACE CODE TO OUTPUT ROUTINE.	C82B 20 D2 FF
27		LDA +TEMP	;RESTORE THE ACCUMULATOR.	C82E A5 02
28		RTS		C830 60

The second part of the program retrieves the number in the accumulator, masks the most-significant nibble, converts the number in the least-significant nibble to ASCII using subroutine HEXCII, and then uses CHROUT to output the code to the screen. The last part of the program sends a space to CHROUT to provide a space on the screen between successive numbers, the original number in the accumulator is recovered, and control returns to the calling program.

Thus, if the number \$F3 were in the accumulator, lines 10 through 16 of the program in Example 7-3 would output the "F." Lines 19 through 22 would output the "3." Lines 25 and 26 output a space, and lines 27 and 28 restore the accumulator and return to the calling program.

Because the programs in Examples 6-18, 6-19, 7-3, and 7-4 are so useful, we have written them so they all can be stored in page \$C8 of memory. Then they are out of the way of other programs, and they can easily be stored on disk and recalled. We suggest that you do the same thing. In subsequent examples we will call these routines. These examples will also serve to test the HEXCII, ASCHEX, PRBYTE, and GETBYT routines. We turn next to a description of the GETBYT routine.

D. Getting a byte from the keyboard

We will now describe a subroutine called GETBYT, which is listed in Example 7-4. This subroutine is used to input one byte of information from the keyboard and return with this byte in the accumulator. The byte of information is typed on the keyboard as two hexadecimal digits. Subroutine GETBYT calls subroutine ASCHEX in Example 6-19 to convert the ASCII representation of a hexadecimal digit into a hexadecimal number.

We have divided the program in Example 7-4 into four parts. The first part, consisting of lines 10 through 14, fetches a character from the keyboard and prints the character on the screen. This part uses the operating system subroutine GETIN to read the keyboard and the operating system subroutine CHROUT to echo the keyboard character on the screen. When a key is pressed, subroutine GETIN returns with the ASCII code for the character on the key. In this case, we are assuming that a key representing a hexadecimal digit from 0 to F was pressed. At the end of the first part of the program, this code is in the accumulator.

The second part of the program calls subroutine ASCHEX in Example 6-19 to convert the ASCII code to a hexadecimal number. The most-significant digit is normally entered first, so this is the most-significant digit of the two-digit number. However, ASCHEX returns with this number in the least-significant nibble, so four ASL instructions are used to shift this number to the most-significant nibble. This result is temporarily stored in location TEMP, completing the second part of the program in Example 7-4.

Example 7-4. A Program to Get a Byte of Data from the Keyboard

Object: Fetch two hexadecimal digits from the keyboard and return with the number in the accumulator.

10	GETBYT	JSR GETIN	;READ THE KEYBOARD.	C831 20 E4 FF
11		BEQ GETBYT	;WAIT FOR A NON-ZERO RESULT.	C834 F0 FB
12		TAX	;SAVE CHARACTER IN X FOR A MOMENT.	C836 AA
13		JSR CHROUT	;OUTPUT THE KEYBOARD CHARACTR.	C837 20 D2 FF
14		TXA	;GET THE CHARACTER BACK FROM X.	C83A 8A
15	;			
16	;			
17		JSR ASCHEX	;CONVERT ASCII TO HEXADECIMAL.	C83B 20 09 C8
18		ASL A	;SHIFT DIGIT TO THE MOST	C83E 0A
19		ASL A	SIGNIFICANT NIBBLE USING FOUR	C83F 0A
20		ASL A	;SHIFTS LEFT IN THE	C840 0A
21		ASL A	ACCUMULATOR ADDRESSING MODE.	C841 0A
22		STA +TEMP	;TEMPORARILY STORE THIS RESULT.	C842 85 02
23	;			
24	;			
25	LOAF	JSR GETIN	;GET ANOTHER CHARACTER CODE.	C844 20 E4 FF
26		BEQ LOAF		C847 F0 FB
27		TAX	;SAVE IT IN X FOR A MOMENT.	C849 AA
28		JSR CHROUT	;OUTPUT IT TO SEE IT.	C84A 20 D2 FF
29		TXA	;PUT CODE BACK IN ACCUMULATOR.	C84D 8A
30	;			
31	;			
32		JSR ASCHEX	;CONVERT IT TO HEXADECIMAL.	C84E 20 09 C8
33		ORA +TEMP	;COMBINE IT WITH THE FIRST DIGIT.	C851 05 02
34		RTS		C853 60

The third part of the program in Example 7-4 is exactly like the first part. Its function is to obtain the second character code from the keyboard and print the character on the screen. The character code is in the accumulator when the program moves to its final part.

The last part of the program, lines 32 through 34, converts the last character entered into a hexadecimal number. This number will occupy

the least-significant nibble. This nibble is ORAed with the first nibble to give the final result. The binary equivalent of the two hexadecimal digits entered on the keyboard is now in the accumulator, completing the program.

A program that tests the subroutines in Examples 6-18, 6-19, 7-3, and 7-4 is given in Example 7-5. This program calls GETBYT to get a byte of data from the keyboard and calls PRBYTE to print the byte on the screen. A space is also printed to separate the characters printed on the screen by GETBYT from those printed on the screen by PRBYTE. Load all of the routines just mentioned into the locations from \$C800 to \$C853. Then load the program in Example 7-5 into memory and execute it with a SYS 49152 command. If all of your subroutines have been assembled and loaded correctly, you should be able to type a two-digit (hexadecimal) number on the keyboard and see it displayed on the screen. Do not wait for a flashing cursor because there will be none. Directly after the SYS 49152 command, type a two-digit number on the keyboard. You now have some valuable routines that provide keyboard input and screen output for your machine-language programs. Store them on tape or disk.

Example 7-5. A Program to Test Examples 7-3 and 7-4

Object: Input a byte with subroutine GETBYT and output the same byte with subroutine PRBYTE.

10 TEST	JSR GETBYT	;GET A NUMBER TO PUT IN A.	C000 20 31 C8
11 STA +TEMP		;SAVE IT HERE.	C003 85 02
12 LDA #\$20		;OUTPUT A SPACE.	C005 A9 20
13 JSR \$FFD2			C007 20 D2 FF
14 LDA +TEMP		;GET THE NUMBER.	C00A A5 02
15 JSR PRBYTE		;OUTPUT IT AS TWO HEX DIGITS.	C00C 20 13 C8
16 CLV		;FORCE A BRANCH.	C00F B8
17 BVC TEST		;STAY IN THIS LOOP FOREVER.	C010 50 EE

E. Simple arithmetic with shifts

It is possible to do some relatively simple multiplications and divisions with the shift instructions. To illustrate, suppose we have the number \$84 = 10000100 in the accumulator. After one shift right (LSR), the binary number is clearly 01000010, or \$42. After another shift right, the binary number is 00100001, or \$21. You can see that one shift right is equivalent to division by two, two shifts right is equivalent to division by four, and so on. Of course, if we had started with an odd number, then the first shift right would have moved a bit into the carry, where it is lost. In other words, you can divide by 2, 4, 8, 16, ..., but the remainders are lost as they disappear into the carry flag or disappear altogether in that great bit bucket in the sky.

Suppose we start with the number \$5 (00000101) in the accumulator and try some left shifts. The first shift left gives 00001010 = \$0A, or 10. The second shift left gives 00010100 = \$14, or 20. The third shift left gives 00101000 = \$28,

or 40. In other words, successive shifts left multiply the number in the accumulator by two, provided no significant bits shift out of the accumulator into the carry flag or beyond.

To summarize:

- Multiplication by 2, 4, 8, 16, ... can be accomplished with successive ASL instructions, provided no significant bits are shifted into oblivion.
- Division by 2, 4, 8, 16, ... can be accomplished with successive LSR instructions, with the realization that any remainder is lost.
- To avoid losing significant bits or remainders, ROL or ROR instructions can be used to move the bits lost by the ASL or LSR instructions into new locations.

It is also possible to perform certain other multiplications with the ASL instruction. For example, since $10 = 2 + 8$, you can multiply by 10 by first multiplying by 2, then by 8, and then add these two results. How would you multiply by five? By 40?

The program in Example 7-6 demonstrates these ideas. It uses subroutine GETBYT in Example 7-4 to get an eight-bit number in the accumulator. It prints this result, then it goes through a succession of eight shifts, printing the number each time. Obviously, after eight shifts the number in the accumulator will be zero. Load GETBYT and the program in Example 7-4. Execute them with a SYS 49152 command. Enter a number like \$84 from the keyboard and observe the results. Try an odd number such as \$7F. What happens to it during successive divisions by two? In Example 7-6 replace the LSR op code with an ASL op code using the accumulator addressing mode. Run the program again and enter a number such as \$41. What do you observe? Finally, experiment with ROL and ROR instructions in place of the LSR instruction in Example 7-6. Can you understand and explain your results?

Example 7-6. A Program to Illustrate Successive Divisions by 2

Object: Demonstrate the effect of successive LSR instructions on a number.

10	START	JSR	GETBYT	;GET A NUMBER IN A.	C000	20	31	C8
11	LOOP	JSR	PRBYTE	;PRINT IT.	C003	20	13	C8
12		LSR	A	;SHIFT IT RIGHT.	C006	4A		
13		BNE	LOOP	;CONTINUE TO OUTPUT IT UNTIL	C007	D0	FA	
14		RTS		;IT IS ZERO.	C009	60		

F. Watching the clock

The next program illustrates an application for the PRBYTE subroutine. The program in Example 7-7 illustrates how the time-of-day clock on the CIA can be read using the PRBYTE subroutine in Example 7-3. The clock is started with the STA TENTHS instruction on line 10. After that the program simply reads and prints the hours, minutes, seconds, and tenths of seconds, using PRBYTE to display the information. Since the TOD registers keep time in BCD, the numbers displayed are decimal numbers. Finally, on lines 20 through 24 the

cursor is backspaced 12 spaces so that the new times are written directly over the old times. The instruction on line 25 keeps the program in an infinite loop. Be sure subroutine PRBYTE is in memory before trying to execute the program in Example 7-7.

Example 7-7. A Program to Read the TOD Clock

Object: Read and output each register of the TOD clock on the CIA.

10	TOD	STA TENTHS	; START THE TOD CLOCK.	C000 8D 08 DC
11	MORE	LDA HOURS	; READ THE HOURS.	C003 AD 0B DC
12		AND #\$1F	; MASK SOME HIGH BITS.	C006 29 1F
13		JSR PRBYTE	; OUTPUT THE HOURS.	C008 20 13 C8
14		LDA MINUTS	; PETCH MINUTES REGISTER CONTENTS.	C00B AD 0A DC
15		JSR PRBYTE	; OUTPUT TO THE SCREEN.	C00E 20 13 C8
16		LDA SECNDS	; PETCH THE SECONDS.	C011 AD 09 DC
17		JSR PRBYTE	; OUTPUT TO THE SCREEN.	C014 20 13 C8
18		LDA TENTHS	; GET TENTHS OF SECONDS.	C017 AD 08 DC
19		JSR PRBYTE		C01A 20 13 C8
20		LDY #\$0C	; MOVE CURSOR BACK 12 SPACES.	C01D A0 0C
21	LOOP	LDA #\$9D	; \$9D IS CODE FOR CURSOR LEFT.	C01F A9 9D
22		JSR \$FFD2		C021 20 D2 FF
23		DEY		C024 88
24		BNE LOOP		C025 D0 F8
25		BEQ MORE	; LOOP FOREVER.	C027 F0 DA

IV. More Advanced Arithmetic Operations*

The 6510 microprocessor cannot multiply or divide, so it is up to the programmer to write routines to accomplish these operations. When writing a program to perform a task as complex as multiplication or division, it is frequently useful to perform the task with pencil and paper and then try to write, in English, an algorithm that will accomplish the task. The more closely you think like a microprocessor, the more easily your algorithm will translate into assembly-language code.

Consider multiplication, for example. We know that the product of two four-bit numbers cannot exceed $15 \times 15 = 225$, so an eight-bit location can hold the product. On the other hand, the product of two eight-bit numbers may be as large as 16 bits. Therefore, to keep the original problem simple, we will consider the problem of multiplying two four-bit numbers. Our pencil-and-paper calculation is shown in Table 7-2. It uses the algorithm you used in elementary school. This may not be the most efficient algorithm for the 6510, but it is a start.

If you closely examine the work in Table 7-2, you will see that the *multiplicand* appears once in the set of *partial products* for each binary one in the *multiplier*. Corresponding to each binary zero in the multiplier there is a zero in the set of partial products. Notice, also, that the multiplicand is *shifted left* in successive partial products.

Table 7-2. Multiplying two four-bit numbers.

<i>Hexadecimal</i>	<i>Binary</i>	
\$C	1100	Multiplicand
×	×	
\$B	1011	Multiplier
= \$84	1100	
	1100	
	0000	
	1100	
=	10000100	Product

The algorithm suggested by the calculation illustrated in Table 7-2 and our analysis in the previous paragraph is:

1. Shift the multiplier right into the carry flag.
2. If the carry is set by this shift, add the partial product; otherwise, skip to step 3.
3. Shift the multiplicand left to give a new partial product.
4. If this shift produces a zero, the multiplication is finished; otherwise, return to step 1.

This algorithm is implemented in 6510 assembly language in Example 7-8. The accumulator holds the sum of the partial sums and, therefore, the product when the calculation is finished. Step 1 in our algorithm is accomplished on line 11 in the program in Example 7-8. Step 2 in our algorithm is accomplished with lines 12 through 14, while step 3 is accomplished with line 15. Finally, step 4 in the algorithm is accomplished with line 16. The product is in the accumulator at this point, and it is saved by storing it in memory with line 17.

Example 7-8. A Four-Bit Multiplication Routine

Object: Multiply the number in location \$00FD by the number in location \$00FC. Store the product in location \$00FE.

10	MULTPL	LDA #00	;CLEAR A TO HOLD PRODUCT.	C000	A9	00
11	UP	LSR +MLTP	;SHIFT MULTIPLIER INTO C.	C002	46	FC
12		BCC DOWN	;DO NOT ADD IF C IS CLEAR.	C004	90	03
13		CLC	;ADD (SHIFTED) MULTPLICAND TO	C006	18	
14		ADC +MCND	;FORM PARTIAL SUM.	C007	65	FD
15	DOWN	ASL +MCND	;SHIFT MULTPLICAND LEFT.	C009	06	FD
16		BNE UP	;STOP WHEN MULTPLICAND SHIFTS TO 0.	C00B	D0	F5
17		STA +PROD	;STORE THE PRODUCT.	C00D	85	FE
18		RTS		C00F	60	

Having succeeded in finding a multiplication algorithm, we would like to extend it to handle multiplication of eight-bit numbers. In this case, the product can be as large as a 16-bit number, so we will need two bytes of memory to hold the product. The program in Example 7-9 accomplishes our objective. It differs from the program in Example 7-8 in two details. First, the product is contained in a two-byte number called PRODLO and PRODHII. PRODHII

holds the most-significant byte of the product. Second, rather than shifting the multiplicand left, the partial sum is shifted and rotated right. An analysis of the program will show that this avoids the necessity of doing a two-byte sum.

Example 7-9. An Eight-Bit Multiplication Routine

Object: Multiply the eight-bit number in \$00FC by the eight-bit number in \$00FD. Store the product in locations \$00FE (LSB) and \$00FF (MSB).

10	MLTPLY	LDA #00	;CLEAR A TO HOLD HIGH BYTE.	C000 A9 00
11		STA +PRODLO	;CLEAR LOW BYTE OF PRODUCT.	C002 85 FE
12		LDY #8	;Y IS USED TO COUNT EIGHT LOOPS.	C004 A0 08
13	UP	LSR +MLTP	;SHIFT MULTIPLIER INTO CARRY.	C006 46 FC
14		BCC DOWN	;DO NOT ADD IF C = 0.	C008 90 03
15		CLC	;ADD MULTIPLICAND TO	C00A 18
16		ADC +MCND	;FORM PARTIAL SUM.	C00B 65 FD
17	DOWN	LSR A	;SHIFT THE PARTIAL SUM RIGHT.	C00D 4A
18		ROR +PRODLO	;ROTATE LOW BYTE RIGHT.	C00E 66 FE
19		DEY		C010 88
20		BNE UP	;REPEAT EIGHT TIMES.	C011 D0 F3
21		STA +PRODHI	;FINALLY SAVE THE HIGH BYTE.	C013 85 FF
22		RTS		C015 60

To digress momentarily, it is a common fallacy that everything has a simple explanation. If programming is so simple, why is the demand for good programmers so large? As a matter of fact, many worthwhile programs are extremely difficult to understand, having evolved through many stages of pencil-and-paper calculations, with several programmers contributing improvements. Programs that fall in the "very difficult" category include the eight-bit multiplication program just described and the remaining programs in this chapter.

What follows is the best and most serious advice I can give you if your aim is to understand these programs. Purchase a thick pad of legal-size paper, several pencils, and a large eraser. When you have several hours of uninterrupted time, perform a number of the calculations by hand, using either four-bit or eight-bit numbers. Do not worry about how the computer program is going to accomplish the same task. Next, use pencil and paper to follow the steps in the program you are trying to understand. On your pad of paper, duplicate the effect of each instruction in the program.

Do not be frustrated if your progress is slow. Your understanding will increase slowly until you feel confident that you understand the program. Two days later you will have forgotten everything, but with some additional study it will come back to you. Do not either overestimate the ease with which others may learn or underestimate the effort you must put forth. A college-level assembly-language course occupies a full semester, and the end result is a beginning programmer rather than an expert.

Returning to our main discussion, our binary division program is a result of a procedure similar to the one described for multiplication. That is, we started by performing some binary divisions with pencil and paper, then we

tried to express our calculations as an algorithm that could be performed by the 6510 microprocessor, and finally we translated this algorithm into assembly language. Follow the advice given above if you wish to understand the algorithm and the program.

Example 7-10. A Program to Divide Two Eight-Bit Numbers

Object: If the divisor is in \$00FD and the dividend is in \$00FC, find the quotient and place it in \$00FE.

10	DIVIDE	LDA #00	C000 A9 00
11		LDY #8	C002 A0 08
12	UP	ASL +DIVDND	C004 06 FC
13		ROL A	C006 2A
14		CMP +DIVSOR	C007 C5 FD
15		BCC DOWN	C009 90 02
16		SBC +DIVSOR	C00B E5 FD
17	DOWN	ROL +QUOTNT	C00D 26 FE
18		DEY	C00F 88
19		BNE UP	C010 D0 F2
20		RTS	C012 60

We conclude this section with two programs that we provide, without explanation, because of their usefulness. These are difficult programs to understand, but they are valuable to have, so we include them here for reference. The algorithms are described by John B. Peatman in his book *Microcomputer-Based Design* (McGraw-Hill, New York, 1977), Chapter 7.

The first of these programs, listed in Example 7-11, converts an eight-bit binary number to a binary-coded decimal (BCD) number. The binary number is in location \$00FC, and the program returns with the least-significant byte of the BCD number in location \$00FD and the most-significant byte of the BCD number in location \$00FE. Since an eight-bit binary number can be as large as 255, and each memory location can hold only two BCD digits, we need one location to hold the two least-significant BCD digits and another memory location to hold the single most-significant digit.

Example 7-11. A Binary-to-BCD Conversion Routine

Object: Convert the binary number in location \$00FC to a BCD number in locations \$00FD (LSB) and \$00FE (MSB).

1	;EXAMPLE 7-11		
2	BINUM	EQU \$FC	00FC
3	BCDLO	EQU \$FD	00FD
4	BCDHI	EQU \$FE	00FE
5	BCD	EQU \$C000	C000
6	;		
7	;		
8	;		
9	;		
10	BCD	SED	;PERFORM ADDITION IN DECIMAL MODE.
11		LDA #00	C000 F8
12		STA +BCDLO	C001 A9 00
13		STA +BCDHI	C003 85 FD
14		LDY #08	C005 85 FE
			C007 A0 08

15 WHERE	ASL +BINUM	;SHIFT BINUM INTO CARRY.	C009 06 FC
16 LDA +BCDLO		;ADD BCD NUMBER TO ITSELF.	C00B A5 FD
17 ADC +BCDLO			C00D 65 FD
18 STA +BCDLO			C00F 85 FD
19 LDA +BCDHI		;ADD HI BYTE TO ITSELF.	C011 A5 FE
20 ADC +BCDHI			C013 65 FE
21 STA +BCDHI			C015 85 FE
22 DEY			C017 88
23 BNE WHERE			C018 D0 EF
24 CLD		;CLEAR DECIMAL MODE.	C01A D8
25 RTS			C01B 60

As an example of an application of the program in Example 7-11, suppose you wish to output to the screen the numbers obtained from the game paddles. These numbers are obtained from the POTX and POTY registers on the SID chip, and they are binary numbers. Human beings like their numbers in decimal. The program in Example 7-11 could be used to output the numbers in decimal.

The last program in this section reverses the process in Example 7-11. It takes a two-digit BCD number and changes it to a binary number. The BCD number is located in \$00FD. At the completion of the program, the binary number is in location \$00FC. Human beings like to input numbers in decimal, but the computer likes to work with numbers in binary. The routine in Example 7-12 provides a way to input a BCD number and have it converted into a binary number.

Note the use of the shift and rotate instructions in Examples 7-11 and 7-12.

Example 7-12. A BCD-to-Binary Conversion Routine

Object: Convert the two-digit BCD number in location \$00FD into a binary number in location \$00FC.

1	;EXAMPLE 7-12		
2	BINUM	EQU	\$FC
3	BCD	EQU	\$FD
4	BINARY	EQU	\$C000
5	;		
6	;		
7	;		
8	;		
9	;		
10	BINARY	LDA	#\$80
			;START BINUM WITH 1 IN BIT 7.
11		STA	+BINUM
12	UP	LSR	+BCD
13			;DIVIDE BCD NUMBER BY TWO.
14		ROR	+BINUM
15			;QUOTIENT INTO BINUM.
16		BCS	FINISH
17			;TASK IS FINISHED.
18	FIX	LDA	+BCD
19			;DO WE NEED A FIX?
20		AND	#\$08
21			;ONE IN BIT THREE?
22		BEQ	UP
23			;NO, SO RETURN TO DIVIDE AGAIN.
		LDA	+BCD
			;YES, FIX THE BCD NUMBER.
		SEC	
		SBC	#03
		STA	+BCD
		BCS	UP
23	FINISH	RTS	

V. Summary

The shift and rotate instructions are used to examine a code in a memory location or the accumulator bit by bit. The ASL instruction moves a code left one bit, placing a zero in bit zero, and placing bit seven in the carry flag. Eight ASL instructions operating on the same register or memory location will clear it. The LSR instruction is identical to the ASL instruction, except that it moves a code one bit to the right. Bit seven receives a zero, and bit zero is transferred into the carry flag. The ROL and ROR instructions are similar, except the number in the carry flag is moved back to bit zero (ROL) or bit seven (ROR). Nine successive operations of a ROL or ROR instruction on the same register or memory location will restore the code in that location to its original value.

A ROL instruction is frequently used in conjunction with an ASL instruction. The ASL instruction moves a bit into the carry flag and the ROL instruction moves the carry flag into another register or memory location. Similar statements can be made for the ROR and LSR instructions. Shift and rotate left instructions can be used to perform multiplications by powers of two, while shift and rotate right instructions can be used to divide by powers of two.

Shift and rotate instructions are frequently found in advanced arithmetic and code-conversion routines: multiplication and division, for example. Trial and error, patience, and perseverance are required to write routines such as these.

VI. Exercises

1. To help you understand the effect of the shift and rotate instructions, you should run the demonstration program in Example 7-13. It makes use of the CAI routines described in connection with Example 4-13, and you must load the codes in Table 4-5. By now you should have these codes stored on tape or disk so that you can simply read the CAI program into memory.

Example 7-13. A Program to Demonstrate the Shift and Rotate Instructions

Object: Display a code and the contents of the P register as the code is shifted or rotated.

10	DEMO	JSR GETONE	;GET A BYTE OF DATA.	C000 20 0D C1
11	LOOP	JSR DISPLAY	;DISPLAY THE NUMBER.	C003 20 6C C1
12	;			
13	;			
14		ASL A	;SHIFT THE NUMBER.	C006 0A
15	;			
16	;			
17		JSR WAITKY	;WAIT FOR KEY DEPRESSION.	C007 20 C5 C1
18		CLV	;FORCE A BRANCH	C00A B8
19		BVC LOOP	;BACK TO LOOP.	C00B 50 F6

Notice in Example 7-13 that the most important instruction has been delineated by spaces.

Load this program and the CAI program, then call this program from BASIC with a SYS 49152 command. Type in the number \$41. You will see the number expressed in binary. Now press any key. This causes the shifted number to be displayed. The flags in the P register will also be displayed. Successive key depressions result in successive shifts. You should study what happens to the code after each shift, and you should note how the C, N, and Z flags change.

2. Replace the ASL A instruction in Example 7-13 with a ROL A instruction. Only one op code needs to be changed (refer to Table 7-1). Run the program again and enter any number, such as \$37. Observe and be able to explain what happens with successive key presses.
3. Repeat the previous two exercises with LSR A and ROR A instructions. With sufficient experimentation, you should have a good intuitive feeling for the shift and rotate instructions.
4. Modify the program in Example 7-1 to work with shift and rotate *right* instructions. Test it with POKEs and PEEKs.
5. Write a program that uses subroutine GETBYT in Example 7-4 to place numbers into zero-page locations \$FC and \$FD. Continue the program with a subroutine call to the program in Example 7-8, the four-bit multiplication program. End the program by using subroutine PRBYTE in Example 7-3 to output the number in zero-page location \$FE. You now have a program that inputs two numbers, multiplies them, and outputs the answer. Use this program to input two four-bit hexadecimal numbers (\$03 and \$02, for example) and output the product. What do you get when you multiply \$05 by \$02? You should get \$0A. Perhaps you expected to get 10.
6. Modify the program you just wrote to call the eight-bit multiplication routine in Example 7-9. You should also modify the program to output both bytes of the product with the most-significant byte first. Practice some hexadecimal multiplications.
7. Modify the programs you just wrote to test the division routine in Example 7-10. What happens when you divide a smaller number by a larger? What happens when you divide by zero?
8. Use GETBYT, PRBYTE, and the routine in Example 7-11 to input a binary number and output its decimal equivalent.
9. Use GETBYT, PRBYTE, and the routine in Example 7-12 to input a BCD number and output its binary equivalent.
10. Draw a flowchart for the multiplication and division routines in Examples 7-9 and 7-10, respectively.

8

Indexed Addressing Modes

I. Introduction

At this point in your study of assembly language, you are acquainted with most of the instructions in the 6510 instruction set. You have also made use of at least six of the 13 addressing modes, namely, *immediate*, *absolute*, *zero page*, *accumulator*, *implied*, and *relative*. Examine the instruction set summarized in Table 2-1 and observe the set of *indexed* addressing modes with these column headings: (IND,X); (IND),Y; Z PAGE,X; ABS,X; ABS,Y; and Z PAGE,Y. These are the addressing modes we will study in this chapter. As you will see, these addressing modes enable the 6510 to handle enormous amounts of data in short periods of time.

To provide some motivation, consider the simple task of clearing the first \$19 (25) write-only registers of the 6581 SID chip, a task that should be performed before programming the chip to make sounds. In BASIC, this task may be accomplished with this program:

```
10 FOR X = 0 TO 24  
20 POKE 54272 + X, 0  
30 NEXT
```

If you restrict yourself to the addressing modes you have learned, an assembly-language program to clear these registers would appear as follows:

```
LDA #00  
STA $D400 ;Clear register zero.  
STA $D401 ;Clear register one.  
.  
.  
.  
STA $D418 ;Clear register $18.
```

This program requires 25 STA instructions and 77 bytes of memory. It should be obvious that there must be a better way, and there is. Here it is:

```
LDX #00 ;Start X at zero.  
LDA #00 ;A holds number to be placed in registers.  
LOOP STA $D400,X ;Store 0 in location $D400 + X
```

```

INX      ;Next X. Replace X with X + 1.
CPX #$19 ;If X => $19 then C = 1; otherwise C = 0.
BCC LOOP ;Branch if C = 0 (X < $19).

```

You should carefully compare this program with the BASIC program listed above. Note that \$D400 = 54272. Like the BASIC program, it will go through the loop \$19 (25) times, starting with X = 0 and ending with X = \$18 (24). When X in either program reaches \$19, the programs exit from their loops. The *most important comparison you can make* with the BASIC program is between the

`POKE (54272 + X), 0`

and the

`STA $D400,X`

instructions because *they perform identical functions.*

The addressing mode indicated by the

`STA $D400,X`

notation is called the *absolute indexed by X* addressing mode, and it is abbreviated to "ABS,X" in the column headings in Table 2-1. Knowing that the STA \$D400,X instruction performs the same function as the POKE instruction in our BASIC example, you must realize that the STA instruction in this addressing mode stores the number in the accumulator at the address found by *adding the number in the X register to the address \$D400.*

With some motivation and a brief and intuitive introduction to indexed addressing, we turn to a more formal presentation.

II. The Absolute Indexed Addressing Modes

The two absolute indexed addressing modes, ABS,X and ABS,Y, are identical except that in the first case the number in the X register is used while in the second case the number in the Y register is used. We will discuss the ABS,X addressing mode.

Although there are many instructions that make use of the ABS,X addressing mode, we will use the STA instruction in our explanation. The *form* of an instruction using the ABS,X addressing mode in an assembly language program is

`STA ADDR,X`

In this case, the symbol ADDR stands for a 16-bit address that is said to be *indexed by X*, the number in the X register. Symbols other than ADDR may be used. When executed, the STA ADDR,X instruction will cause the number in the accumulator to be stored in the memory location *whose address is the sum of the 16-bit number symbolized by ADDR and the 8-bit index in the X register.*

Suppose ADDR symbolizes \$D400. Then the assembled version of the instruction is

STA ADDR,X 9D 00 D4

where the op code \$9D is obtained from the instruction set in Table 2-1, or it is provided by your assembler. Notice that ADDR symbolizes a 16-bit address, called the *base address*, that consists of a low-order byte called BAL for base address low and a high-order byte called BAH for base address high. In this example, BAL = \$00 and BAH = \$D4. Observe that the op code is followed by the base address, with BAL first and BAH second, forming an instruction consisting of three bytes.

The address of the operand of the STA instruction is found by adding the number in the X register to the address BAH:BAL. Recall that we identify the address of the operand of an instruction with two bytes, ADH and ADL. Now we can diagram the ABS,X addressing mode in Figure 8-1, and we can neatly summarize these concepts as follows:

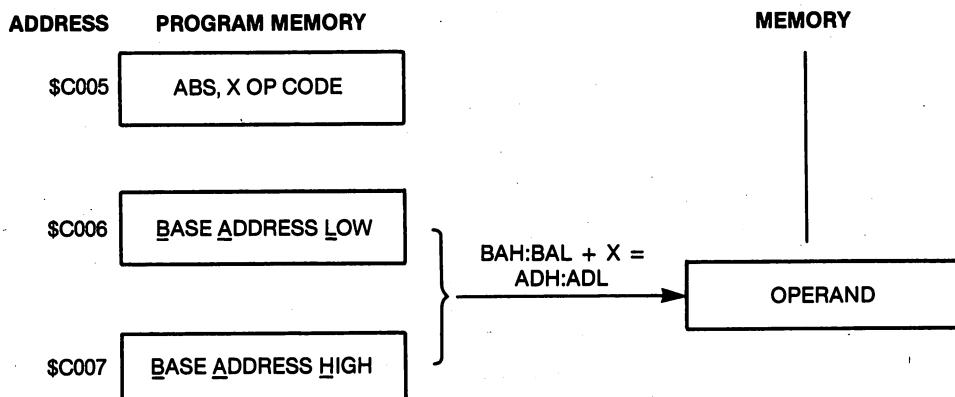


Figure 8-1. Diagram of the absolute indexed by X addressing mode.

- The ABS,X addressing mode is a three-byte instruction. The first byte is the op code. The next two bytes, BAL and BAH, form a 16-bit address.
- The address of the operand, ADH:ADL, is found by adding X to the address BAH:BAL. That is,

$$\text{ADH:ADL} = \text{BAH:BAL} + \text{X}$$

- In other words, the ADL of the operand is the sum of the number in the X register and the second byte of the instruction.
- The ADH of the operand is the third byte of the instruction plus any carry from the sum of X and the second byte.

Example 8-1 illustrates some of these ideas for the LDA instruction when it uses the ABS,X addressing mode.

Example 8-1. Calculating the Address of the Operand with the ABS,X Addressing Mode

Assemble an LDA ADDR,X instruction if ADDR is \$C020. At what address is the operand located if X = \$7F? If X = \$E5?

Solution: Refer to Table 2-1 to find the op code. The assembled instruction is

LDA ADDR,X BD 20 C0

If X = \$7F, the operand is located at \$C020 + \$7F = \$C09F. If X = \$E5, the operand is located at \$C020 + \$E5 = \$C115. Notice in this last case that there is a carry from the sum of X and BAL.

Example 8-2 illustrates a program to clear the first \$19 registers of the SID chip. Notice how the loop structure in Example 8-2 varies from the assembly-language program described in the introduction. In this case, the INX instruction is placed *before* the STA ADDR,X instruction, so X must be initialized to -1 (\$FF). The advantage, if any, to this type of loop structure is that the loop ends when X is exactly equal to the ADL of the last register to be cleared.

Example 8-2. A Program to Illustrate the ABS,X Addressing Mode

Object: Clear the write-only registers of the SID chip.

10 START	LDX #\$FF	;START X AT -1.	C000 A2 FF
11	LDA #00	;CLEAR A.	C002 A9 00
12 LOOP	INX	;NEXT X. (THE FIRST X IS ZERO.)	C004 E8
13	STA ADDR,X	;ADH:ADL = BAH:BAL + X = \$D400+X.	C005 9D 00 D4
14	CPX #\$18	;IS X>>\$18? YES, C=1. NO, C=0.	C008 E0 18
15	BCC LOOP	;C = 0, CLEAR MORE REGISTERS.	C00A 90 F8
16	RTS	;C = 1, TASK IS FINISHED.	C00C 60

Example 8-3 illustrates the use of the ABS,Y addressing mode to accomplish the same objective as the program in Example 8-2. The ABS,Y addressing mode works in exactly the same way as the ABS,X addressing mode, except that the number in the Y register is used rather than the number in the X register. Example 8-3 illustrates yet a third way of constructing the loop. In this program, the registers are cleared starting with the register with the largest address, \$D418, and ending with the register whose address is the same as BAH:BAL, \$D400. Choose the loop structure with which you feel most comfortable.

A *page* of memory consists of 256 memory locations that all have the same ADH. For example, page \$C0 of memory consists of the locations \$C000 to \$C0FF. An instruction such as

AND ADDR,X

that uses the absolute indexed addressing mode will reference one page of memory if ADDR symbolizes a *page boundary*. A page boundary is any address

whose ADL is \$00; for example, \$C800. Of course, if the base address of an instruction using the ABS,X addressing mode is not a page boundary, then, for a certain value of X, a *page crossing* will occur. A page crossing was illustrated in Example 8-1, when the base address was \$C020 and X was \$E5. In this case, the base address is in page \$C0 and the location referenced by the instruction is in page \$C1. Any time a page crossing occurs, the execution time of the instruction is one clock cycle more than if no page crossing occurs.

Example 8-3. A Program to Illustrate the ABS,Y Addressing Mode and a Modified Loop Structure

Object: Clear the write-only registers of the SID chip.

10	START	LDY #\$19	;\$19 IS NUMBER OF REGISTERS CLEARED.	C000 A0 19
11		LDA #00	;CLEAR A.	C002 A9 00
12	LOOP	STA ADDR-1,Y	;CLEAR REGISTER AT \$D3FF+Y.	C004 99 FF D3
13		DEY	;NEXT Y. STEP = -1.	C007 88
14		BNE LOOP	;IS Y = 0? NO, RETURN TO LOOP.	C008 D0 FA
15		RTS	;YES, TASK IS FINISHED.	C00A 60

Regardless of whether a page crossing occurs or not, the maximum number of locations that can be referenced with either the ABS,X or ABS,Y addressing modes with a *single* base address is 256 locations.

Suppose your task is to clear the Commodore 64 screen memory. Screen memory consists of 1,000 memory locations starting at \$0400. The 1,000 comes from the fact that the screen consists of 25 rows and 40 columns, and one byte of memory is used to identify the character in each of these 1,000 screen locations. It is impossible to reference 1,000 memory locations with the ABS,X addressing mode with a single base address. The program in Example 8-4 illustrates how this problem may be solved. We have used four different base addresses. Each base address is used to reference 250 locations. Of course, if many pages of memory must be modified, for example, when using bit-mapped graphics, the technique illustrated in Example 8-4 becomes awkward, and another addressing mode will be more efficient.

Example 8-4. Clearing the C-64 Screen Memory

Object: Clear the 1,000 memory locations starting at \$0400.

10	CLRSCN	LDA #\$20	;\$20 IS SCREEN CODE FOR A SPACE.	C400 A9 20
11		LDX #250	;START X AT 250.	C402 A2 FA
12	LOOP	DEX	;NEXT X. STEP -1.	C404 CA
13		STA SCN,X	;ADH-ADL = SCN + X.	C405 9D 00 04
14		STA SCN+250,X	;ADH-ADL = SCN + 250 + X.	C408 9D FA 04
15		STA SCN+500,X	;ADH-ADL = SCN + 500 + X.	C40B 9D F4 05
16		STA SCN+750,X	;ADH-ADL = SCN + 750 + X.	C40E 9D EE 06
17		BNE LOOP	;IF X <> 0 THEN GO TO LOOP.	C411 D0 F1
18		RTS		C413 60

Example 8-4 also illustrates some tricks that your assembler can perform for you. On line 11 we have 250 in decimal in the operand field. The assembler will convert this to the hexadecimal \$FA found in the machine-language program. Another trick is illustrated in lines 13 through 15. The operand field in line 13 contains SCN + 250. Since the assembler has been told what SCN is, namely, \$0400, the assembler will convert 250 to hexadecimal and add it to \$0400; the base address for the STA instruction on line 13 then becomes \$04FA. In other words, the assembler can handle a certain amount of arithmetic in the operand field. Remember, it is the assembler that makes the necessary calculations.

We will provide additional examples of this addressing mode at the end of this chapter. We turn next to a study of the zero-page indexed addressing mode.

III. The Zero-Page Indexed Addressing Mode

The zero-page indexed addressing mode is almost the same as the absolute indexed addressing mode, *except* that the operand is always in page zero of memory. There are two such modes, depending on which index register, X or Y, is used as an index. The two modes are designated Z-PAGE,X and Z-PAGE,Y. Only two instructions, LDX and STX, use the Z-PAGE,Y mode. The Z-PAGE,Y mode is rarely used, but it is worth keeping in mind that it is available. It works in the same way as the Z-PAGE,X addressing mode that we will now describe.

Consider the LDA instruction in its Z-PAGE,X addressing mode. Its assembly-language form is

LDA BAL,X

which is exactly the same as the form of the ABS,X addressing mode. To distinguish the ABS,X and the Z-PAGE,X modes, some assemblers require a left-arrow symbol before the BAL symbol. Other assemblers will automatically use the Z-PAGE,X mode if BAL is an address in page zero of memory. The Z-PAGE,X addressing mode may be summarized as follows:

- The first byte of an instruction using the Z-PAGE,X addressing mode is the op code.
- The second byte of the instruction is the low byte of a zero-page address, called BAL. The high byte, BAH, is, of course, zero.
- The operand is found in the zero-page address whose ADL is BAL + X. The high byte of the address of the operand is, of course, zero.

This mode is diagrammed in Figure 8-2. In symbols,

```
ADL = BAL + X
ADH = $00
operand = (ADH:ADL)
```

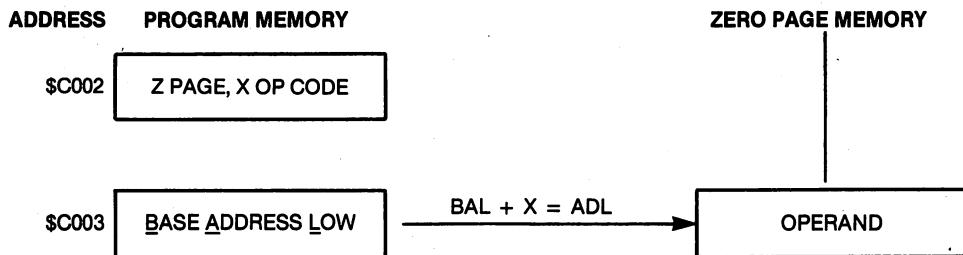


Figure 8-2. Diagram of the zero-page indexed by X addressing mode.

where BAL is the second byte of the instruction, X is the number in the X register, ADL is the low-order byte of the zero-page address where the operand is found, and ADH is understood to be \$00. We have used parentheses "()" to mean "the contents of." This is the traditional use of parentheses in assembly-language programming. Thus, the symbolism "operand = (ADH:ADL)" is read, "the operand is *the contents of* the memory location with address ADH:ADL."

It is important to realize that any carry from the sum BAL + X is *disregarded*. In other words, *only* memory locations in page zero will be referenced. Example 8-5 illustrates some of these ideas.

Example 8-5. Locating the Operand in the Z-PAGE,X Addressing Mode

Assemble the instruction

LDA BAL,X

if the Z-PAGE,X addressing mode is desired and BAL is \$80. Where will the operand be located if X = \$1D? If X = \$81?

Solution: The assembled version of the instruction is

LDA BAL,X B5 80

If BAL is \$80 and X is \$1D, then

$$\text{ADL} = \$80 + \$1D = \$9D$$

so the operand is in location \$009D. If X is \$81, then

$$\text{ADL} = \$80 + \$81 = \$01$$

so the operand is in location \$0001. Note that this case illustrates the situation in which the carry from the sum of BAL and X is ignored.

The program in Example 8-6 illustrates a use for Z-PAGE,X addressing. The addresses in Figure 8-2 correspond to the program in this example. This program outputs the ASCII codes stored in a table in page zero of memory starting at \$00F8 and ending at \$00FF. The codes are sent to the operating system

subroutine CHROUT, which prints the characters on the screen. The particular characters we have chosen to print on the screen are the symbols for the flags in the processor status register, N, V, D, B, I, Z, and C. We chose to output a space for the unused flag bit.

Example 8-6. A Program to Illustrate the Z-PAGE,X Addressing Mode

Object: Output the letter symbols for the flags in the P register.

10 ZPAGE	LDX #0	;START X AT ZERO.	C000 A2 00
11 LOOP	LDA +BAL,X	;GET CHARACTER FROM LIST.	C002 B5 F8
12	JSR CHROUT	;OUTPUT IT TO SCREEN.	C004 20 D2 FF
13	INX	;NEXT X.	C007 E8
14	CPX #8	;IF X=>8, THEN C=1; OTHERWISE C=0.	C008 E0 08
15	BCC LOOP	;BRANCH TO LOOP IF C=0.	C00A 90 F6
16	RTS		C00C 60
17 ;			
18 ;			
19 EQU \$F8		00F8	
20 BYT 'NV		00F8 4E 56	
21 BYT \$20		00FA 20	
22 BYT 'BDI		00FB 42 44 49	
23 BYT 'ZC		00FE 5A 43	

Notice that our assembler was used to define the data to be placed in the table that the program sends to subroutine CHROUT. The EQU directive at the end of the listing in Example 8-6 is *not* a new instruction. A *directive* is a command that the assembler understands: it will not be recognized by the 6510. In Example 8-6, the EQU directive informs the assembler that the starting address of the table of codes is \$00F8. Also observe that we used the assembler's BYT directive. This directive is used to define the codes that are placed in the table. Study the specifications for these directives that appear in your assembler manual.

To test the program in Example 8-6 begin by loading the codes in the table. You can use your assembler or POKE them into memory, starting at location \$00F8 (248). In decimal, the codes are 78, 86, 32, 66, 68, 73, 90, and 67 for N, V, space, B, D, I, Z, and C, respectively. Then, when the program is also in memory, enter SYS 49152 to execute it.

IV. The Indirect Indexed Addressing Mode

The indirect indexed addressing mode is the most powerful addressing mode in the 6510 instruction set. It may be difficult to grasp, but it is worth the effort to understand this addressing mode. At the very least, do not give up trying to understand it until you have studied the sample programs. Sometimes programs are more easily understood than their explanations. The indirect indexed addressing mode is used in applications that require the

manipulation of large amounts of data and it is used in a variety of graphics applications. This mode is identified in the instruction set in Table 2-1 by the notation "(IND),Y."

The essential idea of *indirect addressing* is that the instruction *does not* specify the address of the operand, but the instruction *does* contain information about *where* the address of the operand is stored.

In other words, the address of the operand is not stored in the program, but it is stored elsewhere in memory, specifically in page zero, where it may be modified. *Indexing* this stored address means that the number in the Y register will be added to it, just as in the case of the ABS,Y addressing mode. Pay close attention to what follows.

Consider, for example, the CMP instruction in its indirect indexed addressing mode. The assembly-language form of this instruction is

CMP (IAL),Y

Instructions using the (IND),Y mode are two-byte instructions. The first byte is the op code. The second byte, symbolized by IAL, is the *low byte* of an address in page zero of memory. Of course, the *high byte* of this address is \$00 because it is in page zero. IAL is an acronym for *indirect address low*. The number stored in IAL is the base address low, BAL, of a location in memory. The base address high, BAH, is stored in the zero-page memory location just above IAL, namely, at IAL + 1.

Thus, the zero-page memory locations IAL and IAL + 1 contain an address that we call BAH:BAL.

Although we have not yet located it, we are closing in on the elusive operand of the indirect indexed addressing mode. In fact, if the number Y in the Y register is added to BAL and any carry from this sum is added to BAH, then we have the address of the operand, which we call ADH:ADL.

Let us try to summarize this in English before going to symbols. The first byte of an instruction using the (IND),Y addressing mode is the op code. The second byte of the instruction is a zero-page address, called IAL. That is the end of the information in the program itself. In IAL is a number called BAL. In IAL + 1 is a number called BAH. The combination, BAH:BAL, is a 16-bit address. Adding Y to this 16-bit address gives ADH:ADL, the 16-bit address of the operand.

This addressing mode is diagrammed in Figure 8-3. The addresses in Figure 8-3 are related to the program in Example 8-8.

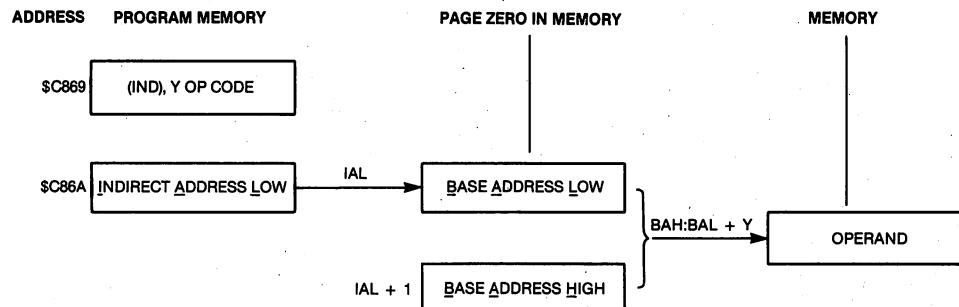


Figure 8-3. Diagram of the indirect indexed addressing mode.

Now we try a symbolic description of this mode, then we will give an example that will clarify this discussion. Recall that we use parentheses to mean "*the contents of*" the memory location represented by the symbol *inside* the parentheses. Thus, (IAL) means "the contents of IAL."

The indirect addressing mode can be described symbolically as follows. Let IAL represent the second byte of the instruction. Then

$$\begin{aligned} \text{BAL} &= (\text{IAL}) \\ \text{BAH} &= (\text{IAL} + 1) \\ \text{ADH:ADL} &= \text{BAH:BAL} + Y \\ \text{(operand)} &= (\text{ADH:ADL}) \end{aligned}$$

We can be even more concise by eliminating BAL and BAH. Thus

$$\begin{aligned} \text{ADL} &= (\text{IAL}) + Y \\ \text{ADH} &= (\text{IAL} + 1) + C \\ \text{(operand)} &= (\text{ADH:ADL}) \end{aligned}$$

where C is the carry from the first sum: it is *not* associated with the carry flag in the P register.

Thus, the instruction

STA (IAL),Y

literally means "store the number in the accumulator at the address found by adding *the contents of* IAL to Y and any carry from this sum to the contents of IAL + 1."

Here is how the microprocessor executes the STA (IAL),Y instruction. After it fetches and interprets the op code, it fetches IAL from program memory. It then fetches the numbers from IAL and IAL + 1, in turn. These two eight-bit numbers are placed together to form the 16-bit number BAH:BAL. The number in the Y register is added to this 16-bit number to form the 16-bit address of the operand, ADH:ADL. Finally, the microprocessor fetches the operand from this address and places it in the accumulator.

An example is overdue. Study Example 8-7.

Example 8-7. Locating the Operand in the (IND),Y Addressing Mode

Assemble the EOR (IAL),Y instruction, where IAL is assumed to be \$FB. What is the address of the operand if IAL contains \$00, IAL + 1 contains \$20, and the number \$3F is in the Y index register?

Solution: Using Table 2-1 to locate the EOR op code gives

EOR (IAL),Y 51 FB

for the assembled form of the instruction. Since (IAL) = (\$00FB) = \$00, then BAL = \$00. ADL = BAL + Y = \$00 + \$3F = \$3F, so the ADL of the operand is \$3F. There is no carry from this sum. Since (IAL + 1) = (\$00FC) = \$20, then

BAH = \$20. Since there was no carry from the first sum, the ADH of the operand is \$20. The operand is in location \$203F, and it will be EORed with the number in the accumulator. In symbols:

$$\begin{aligned} \$00 &= (\text{IAL}) \\ \$20 &= (\text{IAL} + 1) \\ \$203F &= \$2000 + \$3F \\ \text{operand} &= (\$203F) \end{aligned}$$

Our first example of a program that uses the (IND),Y addressing mode is shown in Example 8-8. It can be used to input hexadecimal codes and store them in memory. The program begins by calling subroutine GETBYT (Example 7-4) and storing the two-digit hexadecimal number in location IAL + 1. This will be the ADH of the location that is to receive the hexadecimal code. Subroutine GETBYT is called a second time to get another two-digit hexadecimal number to be stored in IAL. This second byte becomes the ADL of the location that is to receive the hexadecimal code. The third time GETBYT is called you input the number you want to store at the address that you just input. The number is stored in that location by the STA (IAL),Y instruction on line 19.

A useful modification of this program would allow you to input a starting address and the number of bytes to input. You would then input those bytes and the program would return.

Example 8-8. A Program to Illustrate the (IND),Y Addressing Mode

Object: Input an address and the byte to be stored at that address.

10	JSR GETBYT	;GET A BYTE FROM THE KEYBOARD.	C855 20 31 C8
11	STA +IAL+1	;MAKE IT THE BAH.	C858 85 FC
12	JSR GETBYT	;GET ANOTHER BYTE FROM THE KEYBOARD.	C85A 20 31 C8
13	STA +IAL	;MAKE IT THE BAL.	C85D 85 FB
14	LDA #\$20	;OUTPUT A SPACE.	C85F A9 20
15	JSR \$FFD2	;CHROUT IS AT \$FFD2.	C861 20 D2 FF
16	JSR GETBYT	;GET THE BYTE TO BE STORED.	C864 20 31 C8
17	LDY #0	;USE INDIRECT INDEXED MODE WITH Y=0.	C867 A0 00
18	;		
19	STA (IAL),Y	;STA AT ADL=(IAL)+Y, ADH=(IAL+1)+C.	C869 91 FB
20	;		
21	RTS		C86B 60

Our second programming example is shown in Example 8-9. It outputs a page of memory starting at the address that is input with two calls to subroutine GETBYT. The LDA instruction is used in the (IND),Y addressing mode on line 18. The program also makes use of subroutine PRBYTE in Example 7-3. Notice that the zero-page locations used to hold BAH and BAL must always be initialized in some manner *before* the indirect indexed addressing mode can be used. In both Examples 8-8 and 8-9, they are initialized with

information that is input from the keyboard. Notice how Y is incremented in a LOOP in Example 8-9. Study this program until you are sure you understand how the section delineated by spaces works.

Example 8-9. Using the LDA Instruction in the (IND),Y Addressing Mode

Object: Output a page of memory beginning at the address that is input from the keyboard.

10	JSR GETBYT	;GET A BYTE FROM THE KEYBOARD.	C870 20 31 C8
11	STA +IAL+1	;MAKE IT THE BAH.	C873 85 FC
12	JSR GETBYT	;GET ANOTHER BYTE FROM THE KEYBOARD.	C875 20 31 C8
13	STA +IAL	;MAKE IT THE BAL.	C878 85 FB
14	LDA #\$20	;OUTPUT A SPACE.	C87A A9 20
15	JSR \$FFD2	;CHROUT IS AT \$FFD2.	C87C 20 D2 FF
16	LDY #0	;START Y AT ZERO.	C87F A0 00
17	;		
18	LOOP LDA (IAL),Y	;LDA FROM ADL=(IAL)+Y, ADH=(IAL+1)+C.	C881 B1 FB
19	JSR PRBYTE	:OUTPUT THE BYTE TO THE SCREEN.	C883 20 13 C8
20	INY	;NEXT Y.	C886 C8
21	BNE LOOP	;OUTPUT 256 BYTES.	C887 D0 F8
22	;		
23	RTS		C889 60

The third example of the (IND),Y addressing mode illustrates how large blocks of memory can be modified. We will discuss the bit-mapped mode (BMM) of the VIC chip later, but before you can do bit-mapped graphics you must clear a block of 8,000 memory locations; otherwise, garbage will appear all over the screen. We are assuming the VIC chip is initialized so the pixels (picture elements) are obtained from memory locations \$2000 through \$3FFF. Later, we will illustrate how the initialization is accomplished.

The program in Example 8-10 illustrates how a short routine can be used to modify thousands of memory locations. Examine the program and you will observe that at the beginning (IAL) = \$00 and (IAL + 1) = \$20. Thus, the first location to be cleared will be \$2000. Lines 17 through 19 are used to clear one page of memory. After line 19, the page number in IAL + 1 is incremented from \$20 to \$21. Location IAL is never modified. Each time line 21 is executed the page number is incremented, and another page is filled with zeros. The program ends when the page number in IAL + 1 exceeds \$3F. Again, you should study the program until you understand how it works.

Example 8-10. A Program to Clear the Bit Map Screen

Object: Fill locations \$2000 through \$3FFF with zeros.

10	CLEAR LDA #\$20	;BITS START AT \$2000	C430 A9 20
11	STA +IAL+1	;SET UP BASE ADDRESS HIGH.	C432 85 FC
12	LDX #\$3F	;BITS END IN PAGE \$3F.	C434 A2 3F
13	LDA #00	;ALL BITS WILL BE CLEARED.	C436 A9 00
14	TAY	;START Y AT ZERO.	C438 A8
15	STA +IAL	;SET BASE ADDRESS LOW.	C439 85 FB
16	;		

```

17 LOOP STA (IAL),Y ;ADL = BAL + Y = (IAL) + Y.      C43B 91 FB
18 INY ;ADH = BAH = (IAL+1) + C. NEXT Y.    C43D C8
19 BNE LOOP ;IF Y <> 0 THEN GO TO LOOP.   C43E D0 FB
20 ;
21 INC +IAL+1 ;INCREMENT PAGE NUMBER.     C440 E6 FC
22 CPX +IAL+1 ;X - (IAL+1). ALL PAGES CLEARED? C442 E4 FC
23 BCS LOOP ;NO, CLEAR ANOTHER PAGE.      C444 B0 F5
24 RTS ;YES, TASK IS COMPLETE.            C446 60

```

Our final example illustrates how indirect indexed addressing might be used in a word processing application. The program is shown in Example 8-11. It accepts characters from the keyboard, outputs the characters to the screen so you can see what you have typed, and stores the character codes in a block of memory starting at \$4000 and extending upward until you terminate the keyboard entry mode by pressing the F1 key on the Commodore 64. When you exit the program by pressing the F1 key, the last element in the list will be \$85, the keyboard code for the F1 key. This enables you to find the end of the list.

The numbers in IAL and IAL + 1 also identify the address of the last location to receive a character. Numbers that identify a particular location in memory are sometimes called *pointers*. You can regard the 16-bit number stored in IAL and IAL + 1 as a pointer. When you exit the program in Example 8-9, the pointer identifies the last location filled with a character code.

Notice that in this example the index Y in the (IND),Y addressing mode was kept at zero, and the number in IAL was modified. Contrast this with the program in Example 8-10, where Y was incremented but the number in IAL was zero.

Example 8-11. A Program to Store Keyboard Codes in Memory

Object: Accept codes from the keyboard and store them in memory starting at \$4000.

```

10 START LDA #$40 ;INITIALIZE IAL+1.          C000 A9 40
11 STA +IAL+1
12 LDA #00 ;INITIALIZE IAL.                 C002 85 FC
13 STA +IAL
14 ENTER JSR GETIN ;GET A KEYBOARD CHARACTER. C004 A9 00
15 BEQ ENTER ;WAIT FOR NON-ZERO ENTRY.       C006 85 FB
16 STA +TEMP ;STORE A HERE FOR A MOMENT.    C008 20 E4 FF
17 JSR CHROUT ;OUTPUT THE CHARACTER.        C00B F0 FB
18 LDA +TEMP ;GET A BACK AGAIN.             C00D 85 02
19 LDY #00 ;USE (IND),Y MODE WITH Y=0.     C00F 20 D2 FF
20 STA (IAL),Y ;STA AT ADL=(IAL), ADH=(IAL+1). C012 A5 02
21 CMP #$85 ;IS IT F1 KEY?                C014 A0 00
22 BEQ OUT ;YES, THEN QUIT.               C018 91 FB
23 INC +IAL ;INCREMENT (IAL).              C01A F0 08
24 BNE ENTER ;GET ANOTHER CHARACTER.     C01C E6 FB
25 INC +IAL+1 ;IF (IAL)=0 THEN INCREMENT (IAL+1) C01E D0 E8
26 BNE ENTER ;GET MORE CHARACTERS.       C020 E6 FC
27 OUT RTS ;FINISHED WITH INPUT.         C022 D0 E4
28           ;                           C024 60

```

Additional programming examples will be provided at the end of the chapter.

V. The Indexed Indirect Addressing Mode

This addressing mode is used so seldom that we will confine ourselves to a very brief description. The indexed indirect addressing mode is symbolized by the notation "(IND,X)." Only the number in the X register is used as an index.

Like the (IND),Y addressing mode described above, the (IND,X) addressing mode is a two-byte instruction. Symbolize the second byte of the instruction by IAL. Then the ADL of the operand is found in the zero-page location whose address is the sum of IAL and X. The ADH of the operand is found in the next location, that is, at IAL + X + 1. In symbols:

$$\begin{aligned} \text{ADL} &= (\text{IAL} + \text{X}) \\ \text{ADH} &= (\text{IAL} + \text{X} + 1) \\ \text{operand} &= (\text{ADH:ADL}) \end{aligned}$$

The (IND,X) addressing mode is sometimes called "pre-indexed indirect" addressing. It is *indirect* because the instruction has no direct information about the address of the operand. The instruction informs the processor *where* this address is to be found. It is "pre-indexed" because the location of the address of the operand must be added to IAL *before* the page-zero location containing the address of the operand can be identified.

In contrast, the indirect indexed addressing mode discussed in the previous section is sometimes called "post-indexed indirect" addressing. Again, it is *indirect* because the instruction informs the processor *where* the address of the operand is to be found, rather than giving it the address directly. It is "post-indexed" because Y is added to the address stored in page zero *after* this address has been located.

The rarity with which the (IND,X) addressing mode is used precludes the necessity for any additional discussion. In virtually all programming situations, the (IND),Y addressing mode will accomplish the same objective as the (IND,X) addressing mode, but with fewer program bytes. The (IND,X) addressing mode simply wastes too much of page zero of memory to be a viable addressing mode.

VI. Additional Programming Examples

We will provide a few additional programming examples that use the addressing modes discussed in this chapter. You should be aware that many programming examples can be found in the extensive literature and software associated with the 6502 microprocessor, and this literature is also relevant to the 6510 microprocessor. Valuable references in this connection are *6502 Assembly Language Subroutines* by Lance A. Leventhal and Winthrop Saville (Osborne/McGraw-Hill, 1982), *6502 Assembly Language Programming* by Lance A. Leventhal (Osborne/McGraw-Hill, 1979), and *6502 Software Design* by Leo J. Scanlon (Howard W. Sams & Co., Inc., 1980). The subject matter in these

references is not intended for the beginning programmer, but once you have mastered the fundamentals of assembly-language programming you will want some of these books in your programming library.

A 1,000-byte section of memory, known as color memory, is used by the VIC chip to determine the colors of the characters on the screen. The section of memory starts at \$D800. A color number from \$0 to \$F placed in these locations determines the color of the character on the screen. See your *Programmer's Reference Guide* for details. The program in Example 8-12 sets all of the color locations to the color number in location \$0002. Notice the similarity between it and the program in Example 8-4. Both use the ABS,X addressing mode. In a moment we will use both of these programs in a low-resolution graphics application.

Example 8-12. A Routine to Fill Color Memory

Object: Place the color number found in location \$0002 in each location in the color memory for the VIC chip.

10	SETCLR	LDA +COLOR	;GET COLOR CODE FROM \$0002.	C414	A5	02
11		LDX #250	;START X AT 250.	C416	A2	FA
12	LOOP	DEX	;NEXT X, STEP -1.	C418	CA	
13		STA CLR,X	;ADH:ADL = CLR + X.	C419	9D	00 D8
14		STA CLR+250,X	;ADH:ADL = CLR + 250 + X.	C41C	9D	FA D8
15		STA CLR+500,X	;ADH:ADL = CLR + 500 + X.	C41F	9D	F4 D9
16		STA CLR+750,X	;ADH:ADL = CLR + 750 + X.	C422	9D	EE DA
17		BNE LOOP	;IF X<>0 THEN GO TO LOOP.	C425	D0	F1
18		RTS		C427	60	

The next example illustrates the (IND),Y addressing mode. This program will take an x-coordinate between 0 and 39 and a y-coordinate between 0 and 24 and place a character on the screen of the video monitor at the (x,y) coordinate. Refer to page 63 of your *Commodore 64 User's Guide* that came with your computer. The address corresponding to any (x,y) position on the 40-by-25 screen is given by the formula

$$\text{ADDR} = 1024 + x + 40*y$$

where x is the column number and y is the row number. Given (x,y), we would like to be able to place a character at that position. Let us put the equation in another form and use hexadecimal numbers:

$$\text{ADDR} = (\$0400 + \$28*y) + x$$

This is vaguely familiar to the form of the indirect indexed addressing mode. The number in parentheses is a BAH:BAL to be calculated. The number x becomes the number in the Y register. Then the instruction

STA (IAL),Y

will store the character code in the accumulator at the position (x,y) if zero-page locations IAL and IAL + 1 contain BAL and BAH, respectively, where

$$\text{BAH:BAL} = \$0400 + \$28*y$$

In other words, we can use the STA (IAL),Y instruction to reference any position on the screen identified by the (x,y) coordinate pair if we transfer x into the Y register and calculate the base address high (BAH) and the base address low (BAL) from the expression

$$\$0400 + \$28*y$$

and store this 16-bit number in zero-page location IAL and IAL + 1.

An identical technique will be used when we write our bit-mapped graphics routines, so you should make a serious attempt to understand this technique as it applies to low-resolution screen graphics.

The first 32 lines of the program in Example 8-13 are required to change the x- and y-coordinates into the address information. The program assumes that x is stored in XPOS and y is stored in YPOS. Line 12 places the x-coordinate in the Y register. That is the simple part. We could write a multiplication routine to find $\$28*y$, but it is simpler to make use of the fact that $\$28$ (40) is $\$5*\8 . Thus, multiplication by $\$28$ is equivalent to successive multiplications by five and eight.

To multiply by five, we first multiply by four by shifting the number two bits to the left. Adding this to the original number gives a number five times the original number. This is accomplished on lines 14 through 23 in Example 8-13. Notice that the LSB of the answer is placed in IAL and the MSB of the answer is placed in IAL + 1.

Recall that multiplication by eight can be accomplished with three shifts left. Lines 24 through 29 accomplish this. Finally, this result is added to $\$0400$ and the answer, BAH:BAL, is placed in IAL and IAL + 1.

This may seem like an immense amount of work to set up the base address for the STA (IAL),Y addressing mode, but it is the only way it can be done. In any case, the heart of the program in Example 8-13, and the part of the program that is most pertinent to this chapter, is line 35. On line 34, the code of the character we wish to plot is fetched from the memory location called CODE; on line 35, the STA (IAL),Y instruction is used to place it at the (x,y) coordinate on the screen of the video monitor.

Example 8-13. A Program to Place a Character at the Coordinate (x,y) on the Screen

Object: Place a character code at the location

$$\text{ADDR} = \$0400 + \$28*y + x$$

in memory. (x,y) are the coordinates of a point on the screen.

10	ADHADL	LDA #0	;CLEAR LOCATION FOR BASE	C430 A9 00
11		STA +IAL+1	;ADDRESS LOW (BAL).	C432 85 FC
12		LDY +XPOS	;Y REGISTER HOLDS X COORDINATE.	C434 A4 FD
13		LDA +YPOS	;GET Y COORDINATE INTO A.	C436 A5 FF
14		ASL A	;MULTIPLY Y COORDINATE BY FOUR	C438 0A
15		ROL +IAL+1	;WITH TWO SHIFTS LEFT AND	C439 26 FC
16		ASL A	;TWO ROTATES LEFT.	C43B 0A
17		ROL +IAL+1		C43C 26 FC

```

18      CLC      ;NEXT ADD THE Y COORDINATE TO THE    C43E 18
19      ADC +YPOS ;PREVIOUS RESULT TO OBTAIN     C43F 65 FF
20      STA +IAL  ;A MULTIPLICATION BY FIVE.       C441 85 FB
21      LDA #0
22      ADC +IAL+1
23      STA +IAL+1
24      ASL +IAL
25      ROL +IAL+1
26      ASL +IAL
27      ROL +IAL+1
28      ASL +IAL
29      ROL +IAL+1
30      LDA #04
31      ADC +IAL+1
32      STA +IAL+1
33      ;
34 PLOT   LDA +CODE ;GET SCREEN CODE TO PLOT.        C45B A5 02
35           STA (IAL),Y ;PLOT IT AT (X-COORD., Y-COORD.) C45D 91 FB
36      ;
37      RTS
38
39
40

```

Example 8-14 will test the programs in Examples 8-13, 8-12, and 8-4. It sets up various colors, clears the screen by calling Example 8-4, sets up color memory by calling Example 8-12, and then bounces a ball back and forth across the top of the screen with successive calls to the plotting routine in Example 8-13. The comments should explain most of the details. A delay routine slows down the ball so that you can observe it. Load the programs in the examples just mentioned and execute the program in Example 8-14. You should observe a ball bouncing back and forth. Try various delays.

Example 8-14. A Program to Test Examples 8-4, 8-12, and 8-13

```

10 TEST   EQU $C460
11      ;
12      ;
13 TEST   LDA #14      ;SET BACKGROUND COLOR TO LT BLUE.    C460 A9 0E
14      STA BKGLCR
15      LDA #13      ;SET BORDER COLOR TO LT GREEN.     C462 8D 21 D0
16      STA BRDCLR
17      JSR CLRSCN
18      JSR SETCLR
19      LDA #03      ;CLEAR THE SCREEN.          C465 A9 0D
20      STA +XPOS
21      LDA #01      ;SET COLOR MEMORY.        C467 8D 20 D0
22      STA +YPOS
23      INC +XPOS
24      JSR PLTBAL
25      LDA +XPOS
26      CMP #39
27      BCC RIGHT
28 LEFT   DEC +XPOS
29      JSR PLTBAL
30      LDA +XPOS
31      BEQ RIGHT
32      BNE LEFT
33 PLTBAL LDA #81
34      STA +CODE
35      JSR ADHADL
36      JSR DELAY
37      LDA #$20
38      STA +CODE
39      JSR ADHADL
40      RTS

```

41	DELAY	LDX #\$20	; DELAY LOOP.	C4A0 A2 20
42		LDY #\$FF		C4A2 A0 FF
43	BRANCH	DEY		C4A4 88
44		BNE BRANCH		C4A5 D0 FD
45		DEX		C4A7 CA
46		BNE BRANCH		C4A8 D0 FA
47		RTS		C4AA 60

VII. Summary

The indexed addressing modes have these designations: ABS,X; ABS,Y; Z-PAGE,X; Z-PAGE,Y; (IND),Y; and (IND,X). In each indexed addressing mode, the number in either the X or Y register is involved. That is why these registers are called *index* registers.

In the case of the absolute indexed addressing modes, the number in the index register is added to the address formed by the second and third bytes of the instruction to give the address of the operand.

In the case of the zero-page indexed addressing modes, the number in the index register is added to the second byte of the instruction to give the zero-page address of the operand.

In the case of the indirect indexed addressing mode, the number in the Y index register is added to an address stored in two locations, IAL and IAL + 1, in page zero of memory. The sum so obtained is the address of the operand. IAL is the second byte of the instruction.

In the case of the indexed indirect addressing mode, the address of the operand is stored in a page-zero location whose address is the sum of the second byte of the instruction and the number in the X index register.

An instruction using either the absolute or zero-page indexed addressing modes can reference up to one page of data. The indirect indexed addressing mode is used when the data occupies several pages of memory. The indexed indirect addressing mode is rarely used. Graphics and sound-generation programs make extensive use of these addressing modes.

VIII. Exercises

1. Where will the

LDA \$C123,Y

instruction find its operand if the number in the Y register is \$7F?

2. Where will the

ADC ZADDR,X

instruction find its operand if ZADDR is the zero page address \$57 and X is \$30?

3. Where will the

ORA (IAL),Y

instruction find its operand if the page-zero location IAL contains \$2F, IAL + 1 contains \$D0, and Y contains the number \$23?

4. Write an assembly-language program that performs the same function as this BASIC sprite-making program:

```
10 ADDR = 832
20 FOR I = 0 TO 62
30 POKE (ADDR + I),0
40 NEXT
```

5. Write a program to store zeros in all of the locations from \$3000 to \$30FF, page \$30 of memory.
6. Write a program to transfer the codes in page \$30 of memory to page \$2F of memory.
7. Write a program similar to the one in Example 8-6 that outputs the word "ACCUMULATOR" to the screen.
8. Write a program that complements Example 8-11 by printing on the screen all of the characters input by the program in Example 8-11. Stop printing when the character code \$85 for the F1 key is reached.
9. Write a program that transfers the numbers in locations \$D800 through \$DFFF to locations \$2000 through \$2800. Use the (IND),Y addressing mode.
10. Write a program that converts ASCII codes for the upper-case alphabetic characters into screen display codes. Refer to Appendix E in the *Programmer's Reference Guide*. Use the ABS,X addressing mode. An ASCII code in the X register should reference a location in memory that contains the screen code. Locate the screen codes in page \$C1 of memory. Devise a way to test your program.

Jumps, Subroutine Calls, And Interrupts

I. Introduction

The 6510 register featured in this chapter is the program counter, or PC. The stack pointer, S, will play a supporting role. The program counter is frequently considered to be made up of a low byte, called PCL for program counter low, and a high byte, called PCH for program counter high. The stack pointer is an 8-bit register. Sometimes it, too, is considered a 16-bit register, with the most-significant byte being \$01.

A machine-language program is an *ordered* set of instructions stored in memory. It is the program counter that makes the execution of a machine-language program an orderly process. Specifically, the program counter contains the 16-bit address of the location in memory where the next byte of the program is found. The 6510 automatically increments the number in the program counter each time it fetches a new byte of the program it is executing. If you could watch it, the program counter would appear to *count* as one byte of the program after another is fetched from memory.

The program counter increments in a systematic fashion at a rate of about one count every microsecond, *unless* it executes a branch instruction, jump instruction, or a subroutine call, or unless the program is interrupted. In any of these cases, the program counter will appear to jump forward or backward by more than one count. We have already described the branch instructions in Chapter 6, and we have made frequent use of subroutines.

In this chapter, we will examine subroutine calls and returns in greater depth than before. We will also introduce several new instructions that modify the program counter. These instructions include the JMP, JSR, RTS, BRK, and RTI instructions. Their op codes are listed in Table 9-1.

The *stack* is a special memory area located in page one, namely, locations with addresses \$0100 through \$01FF. The stack pointer contains the least-significant byte of a page one address. The area of memory known as the stack

Table 9-1. Op codes for instructions that modify the program counter.

Instruction	Description	Op Codes for Each Addressing Mode		
		Absolute	Implied	Indirect
JMP	Unconditional jump	\$4C		\$6C
JSR	Jump to subroutine	20		
RTS	Return from subroutine		\$60	
RTI	Return from interrupt		40	
BRK	Software interrupt		00	
CLI	0→I. Enable interrupts		58	
SEI	1→I. Disable interrupts		78	

is used in conjunction with the execution of subroutines and interrupt routines. Associated with it are a group of instructions described as *stack operation instructions*. These include the PHA, PLA, PHP, PLP, TSX, and TXS instructions. Their op codes are listed in Table 9-2.

Table 9-2. Op codes for the stack operation instructions.

Instruction	Description	Implied Addressing Mode	
		Op Code	
PHA	Push A on the stack; decrement stack pointer		\$48
PHP	Push P on the stack; decrement stack pointer		08
PLA	Increment stack pointer; pull A from the stack		68
PLP	Increment stack Pointer; pull P from the stack		28
TSX	Transfer S to X		8A
TXS	Transfer X to S		9A

The final topic of this chapter will be interrupts. It is possible for signals to be applied to the pins of the 6510 that will cause it to quit executing one program and make it start executing another. In this case, the first program was *interrupted*. We will take a detailed look at the techniques and the applications of interrupt processing. Two instructions that have not yet been mentioned above will be required to deal with interrupts. They are the CLI and SEI instructions listed in Table 9-1.

With this outline of what new topics we intend to pursue in this chapter, let us begin with a discussion of the JMP instruction.

II. The JMP Instruction

The jump instruction, JMP, is analogous to the BASIC GOTO instruction. The operand of the GOTO instruction is the line number of the next BASIC instruction to be executed. The operand of the JMP instruction is the address of the next instruction to be executed. At least this is the case when it uses the absolute addressing mode. Later we will describe its indirect addressing mode.

First consider the absolute addressing mode of the JMP instruction. The first byte is the op code. The second and third bytes are the new values for the program counter. In particular, the second byte of the JMP instruction becomes the new PCL and the third byte becomes the new PCH. After the JMP instruction is executed, the program continues with the op code stored in the location whose address is identical to the number in the program counter.

The assembly-language form of the JMP instruction in its absolute addressing mode is

JMP SOMWHR

where SOMWHR is the *label* of a location in memory where execution will continue after execution of the JMP instruction. Suppose SOMWHR is a label for the address \$CDEF. Then the assembled form of the JMP SOMWHR instruction is

JMP SOMWHR 4C EF CD

The next instruction to be executed begins at the location whose address is \$CDEF.

The program in Example 9-1 illustrates a JMP instruction in its absolute addressing mode. This is a simple program that gets a character from the keyboard using a Commodore 64 operating system subroutine, GETIN. Any non-zero character code is then sent to the operating system subroutine CHROUT to output the character to the screen. Finally, the JMP SOMWHR instruction on line 13 puts the program in an infinite loop to repeat the process.

Example 9-1. Illustrating the JMP Instruction in its Absolute Addressing Mode

Object: Use a JMP instruction to put the input and output subroutines in an infinite loop.

10	SOMWHR	JSR GETIN	;CALL SUBROUTINE GETIN.	C000
11		BEQ SOMWHR	;A = 0 IF NO KEY IS PRESSED.	C000 20 E4 FF
12		JSR CHROUT	;CALL SUBROUTINE CHROUT.	C003 F0 FB
13		JMP SOMWHR	;GO TO ADDRESS LABELED SOMWHR.	C005 20 D2 FF C008 4C 00 C0

Recall that the branch instructions permit the program counter to be modified by no more than +127 and no less than -128. The JMP instruction

is frequently used to branch outside of the -128 to +127 range, although sometimes a series of branch instructions will affect such a branch. It is good programming practice to use the JMP instruction as seldom as possible.

A program that contains JMP instructions cannot be *relocated* without modifying the JMP instructions. For example, if \$C000 is an undesirable starting location for the program in Example 9-1, then it must be moved. Suppose it is moved to \$CF00. Then the JMP instruction must be modified to reflect this change. Specifically, the assembled version of the JMP SOMWHR instruction would become

```
JMP SOMEWHERE    4C 00 CF
```

A program that will execute anywhere in memory without modification is said to be *relocatable*. Programs with no JMP or JSR instructions are always relocatable.

The operating system of the Commodore 64 provides another example of the use of the JMP instruction. This operating system contains a *jump table*, part of which is reproduced in Table 9-3. We digress for a moment to describe how Table 9-3 was obtained.

To obtain Table 9-3, we *disassembled* the part of the Commodore 64 operating system between \$FFB1 and \$FFF3. Many assemblers will have the capability to disassemble a program from memory. A program is disassembled if the binary instructions in memory are converted to assembly-language mnemonics and hexadecimal (or decimal) operands. It is very useful to be able to disassemble programs for which you have no assembly-language listings.

The Commodore 64 operating system uses a jump table to access important and useful subroutines in the operating system. If a new version of the operating system is designed with these subroutines in *different* locations than an older version, the jump table is modified to reflect the new location of the subroutine.

For example, subroutine RDTIM reads the real time clock. It is currently located in ROM at \$F6DD: refer to the eighth line from the bottom of Table 9-3. This line shows that at location \$FFDE in the jump table, we have a JMP \$F6DD instruction. This is a jump to the starting address of RDTIM. To access subroutine RDTIM, we would use a JSR \$FFDE instruction.

Location \$FFDE is not really the starting address of the subroutine; rather, it is a location in the jump table. This means we would *not* go directly to the subroutine. We would go to the subroutine by way of the jump table. If subroutine RDTIM is moved in a later ROM version of the Commodore 64 operating system, the jump table will be modified to reflect this change. Thus, the address \$F6DD might be changed, but the location of the JMP RDTIM instruction, \$FFDE in the jump table, *would not* change. Location \$FFDE will always contain a JMP RDTIM instruction, regardless of where RDTIM is located. This design prevents obsolescence of programs that have been designed around operating system subroutines. The alternative is to duplicate all of these subroutines in your own programs, though this is both inefficient and expensive.

Table 9-3. Commodore 64 operating system jump table.

EQU+\$FFB1	FFB1
JMP+\$ED0C	FFB1 4C 0C ED
JMP+\$ED09	FFB4 4C 09 ED
JMP+\$FE07	FFB7 4C 07 FE
JMP+\$FE00	FFBA 4C 00 FE
JMP+\$FDF9	FFBD 4C F9 FD
JMP+(\$031A)	FFC0 6C 1A 03
JMP+(\$031C)	FFC3 6C 1C 03
JMP+(\$031E)	FFC6 6C 1E 03
JMP+(\$0320)	FFC9 6C 20 03
JMP+(\$0322)	FFCC 6C 22 03
JMP+(\$0324)	FFCF 6C 24 03
JMP+(\$0326)	FFD2 6C 26 03
JMP+\$F49E	FFD5 4C 9E F4
JMP+\$F5DD	FFD8 4C DD F5
JMP+\$F6E4	FFDB 4C E4 F6
JMP+\$F6DD	FFDE 4C DD F6
JMP+(\$0328)	FFE1 6C 28 03
JMP+(\$032A)	FFE4 6C 2A 03
JMP+(\$032C)	FFE7 6C 2C 03
JMP+\$F69B	FFEA 4C 9B F6
JMP+\$E505	FFED 4C 05 E5
JMP+\$E50A	FFF0 4C 0A E5
JMP+\$E500	FFF3 4C 00 E5

Refer once again to Table 9-3. Some of the JMP instructions have parentheses in the operand field. These instructions are using the JMP instruction in its *indirect* addressing mode. In this mode, the operand field *does not* contain the new value for the program counter, PCL and PCH. Instead, it contains the *address* at which the new value of PCL will be found. The new value of PCH is found in the next location.

The assembly-language form of the JMP instruction in its indirect addressing mode is

JMP (ADDR)

where ADDR is a symbol for the address of the location that contains the new PCL. The new PCH will be in location ADDR + 1.

Here is an example of how the indirect JMP instruction is used. Subroutine CHROUT, the operating system that outputs a character, is accessed by a JSR \$FFD2. Examine the jump table listed in Table 9-3 and observe that \$FFD2 is the starting address of the

JMP (\$0326)

instruction. Memory location \$0326 is *not* where subroutine CHROUT begins. Location \$0326 contains, in fact, the LSB of the starting address of subroutine CHROUT. Location \$0327 (\$0326 + 1) contains the MSB of the starting address of CHROUT.

Thus, when the JMP (\$0326) is executed, the microprocessor fetches the new PCL from location \$0326 and the new PCH from location \$0327, and execution continues with the op code located at the address found in the program counter.

The numbers in the memory locations identified by the JMP instruction in its indirect mode are frequently called *jump vectors*. Thus, the jump vector for subroutine CHROUT is located at \$0326 and \$0327. A jump vector consists of two bytes, and it *points* to the location where execution will continue after the JMP indirect instruction is executed.

Does this seem awkward? Why not just jump directly to the subroutine instead of using an indirect jump? Consider the fact that there are several output devices, the screen, the cassette tape recorder, the disk drive, the printer, and so on. By changing the indirect jump vector, you can control where the character code is going to be sent. Thus, if the printer is currently the active output device you wish to use, the jump vector can point to the part of the CHROUT subroutine that outputs a character to the printer. If the screen is the output device, a different jump vector can be placed in locations \$0326 and \$0327 to point to the part of the CHROUT subroutine that prints characters on the screen. In each case, you call the same subroutine at the same location, namely, \$FFD2 in the jump table. However, the program jumps to a different subroutine, depending on which output device is currently active.

By changing the indirect jump vector, you can also vector the output to an output routine that you have written. Notice that Table 9-3 contains a number of indirect jumps. You may want to study your *Programmer's Reference Guide*, pages 272 through 273, to see which routines are called with these jumps.

Example 9-2 illustrates the indirect jump instruction in a program. This routine calls subroutine GETBYT twice to input two bytes that become the MSB and the LSB of the indirect jump vector. Subroutine GETBYT is the routine in Example 7-4. Once the indirect jump vector is in place, the program goes to a subroutine that begins with the indirect jump instruction. This is line 17 of Example 9-2. In other words, the program in Example 9-2 allows you to go to any machine-language subroutine you have in memory, for example, a program you are testing. The routine to which you jump should end in an RTS instruction, which will return control to the calling program in Example 9-2. Example 9-2 is the machine-language equivalent of the SYS xxxx command in BASIC.

We will see another use of the indirect jump when we study interrupts later in this chapter. We turn next to a discussion of how the JSR and RTS instructions function.

Example 9-2. A Program to "GO TO" Another Machine-Language Program

Object: Input an indirect jump vector from the keyboard and execute the routine identified by this vector.

11 GO	JSR GETBYT	; INPUT THE MSB OF THE ADDRESS.	C890 20 31 C8
12 STA +VECT+1	; MAKE IT THE MSB OF THE JMP VECTOR.	C893 85 FC	
13 JSR GETBYT	; INPUT THE LSB OF THE ADDRESS.	C895 20 31 C8	
14 STA +VECT	; MAKE IT THE LSB OF THE JMP VECTOR.	C898 85 FB	
15 JSR JUMP	; CALL THE INDIRECT JUMP SUBROUTINE.	C89A 20 9E C8	
16 RTS		C89D 60	
17 JUMP JMP (VECT)	; EXECUTE THE ROUTINE.	C89E 6C FB 00	

III. Subroutine Calls and Returns

In this book we have frequently called the programs *routines*. A routine is simply a set of instructions that achieve a particular objective. If a program uses the same routine several times, it will be advantageous to use it as a *subroutine*. Otherwise, you must duplicate the routine at each point in the program that it is used, a great waste of program memory and programming effort.

The JSR and RTS instructions have been used many times in this book. For purposes of review, the JSR instruction is a three-byte instruction. The first byte is the op code. The second and third bytes are the LSB and the MSB of the starting address of the subroutine. The JSR instruction works by placing the second and third bytes of the instruction into the program counter. Thus, the second byte of the JSR instruction becomes the new PCL and the third byte of the instruction becomes the PCH. Execution of the subroutine begins at the instruction whose address is PCH:PCL.

The RTS instruction is a single-byte instruction using the implied addressing mode. When executed, it causes the program to *return* to the calling program to execute the instruction immediately following the JSR instruction.

Re-examine Example 9-2. Subroutine GETBYT is located at \$C831, so the JSR GETBYT instruction on line 13 is assembled as follows:

JSR GETBYT C895 20 31 C8

where \$C895 is the address of the *op code* of the JSR instruction. After execution of the JSR GETBYT instruction, the program will continue at \$C831, the address at which subroutine GETBYT begins. Subroutine GETBYT *must end* with an RTS instruction. After this RTS instruction is executed, the program in Example 9-2 will continue with the STA + VECT instruction immediately following the JSR GETBYT instruction.

Now you should understand how the JSR instruction works, but you should have some questions about the RTS instruction. Since the same subroutine can be called from many different places in a main program, how does it know where to return? Refer again to Example 9-2. Subroutine GETBYT is called

twice from this program. The first time it is called, the program must return to location \$C893. The second time it is called, the program must return to location \$C898. The RTS instruction itself contains no information about the return address.

The answer is that the JSR instruction not only modifies the program counter, but it also causes the information about the return address to be stored in a special place in memory. This special place is called the *stack*, and it is confined to page one of memory, namely, locations \$01FF through \$0100. We will examine how this is accomplished in more detail.

IV. Operation of the Stack

The *stack* is page one of memory, and it must be R/W memory. The stack is operated in conjunction with a register in the 6510 microprocessor known as the *stack pointer*, S.

Study the description of the stack and the stack pointer in connection with Figure 9-1. The stack pointer contains the LSB of a page-one address: the MSB of this address is, of course, \$01. In what follows, we will think of the stack pointer as a 16-bit address whose MSB is *understood* to be \$01. The stack pointer identifies a memory location in page one that is available to store information. In Figure 9-1, we have arbitrarily assumed that the current value of the stack pointer is \$A3; consequently, it points to the memory location whose address is \$01A3. Each time information is stored there by the 6510, the stack pointer is *decremented after* the write operation so that it points to the next available location. This is why the stack is sometimes called a *push-down* stack. When information is read from the stack by the 6510, the stack pointer is *incremented before* the read operation to identify the correct location of the data.

Thus, the *first* eight-bit code pushed on the stack is the *last* eight-bit code to be pulled from the stack. This is why the stack is called a *first-in, last-out* (FILO) memory.

Now let us examine what happens when a jump-to-subroutine (JSR) instruction is executed. Assume the stack pointer is \$A3, as indicated in Figure 9-1, and assume we are dealing with the JSR GETBYT on line 11 of Example 9-2. This JSR GETBYT instruction is stored in locations \$C890 through \$C892. In particular, the *last byte* of the JSR GETBYT instruction is located at \$C892. Before the JSR GETBYT instruction changes the program counter to \$C831, the address of the subroutine GETBYT, the 6510 microprocessor stores the number \$C8 on the stack at location \$01A3 and decrements the stack pointer to \$A2. It then stores the number \$92 on the stack at location \$01A2 and decrements the stack pointer to \$A1.

When the RTS instruction in GETBYT is executed, the 6510 first increments the stack pointer to \$A2, reads the number \$92, and stores it in the PCL. The stack pointer is incremented to \$A3, then the 6510 reads the number \$C8 stored in location \$01A3 and stores it in the PCH. Finally, the program counter is incremented, and the next byte of the program is fetched from location \$C892 + 1 = \$C893.

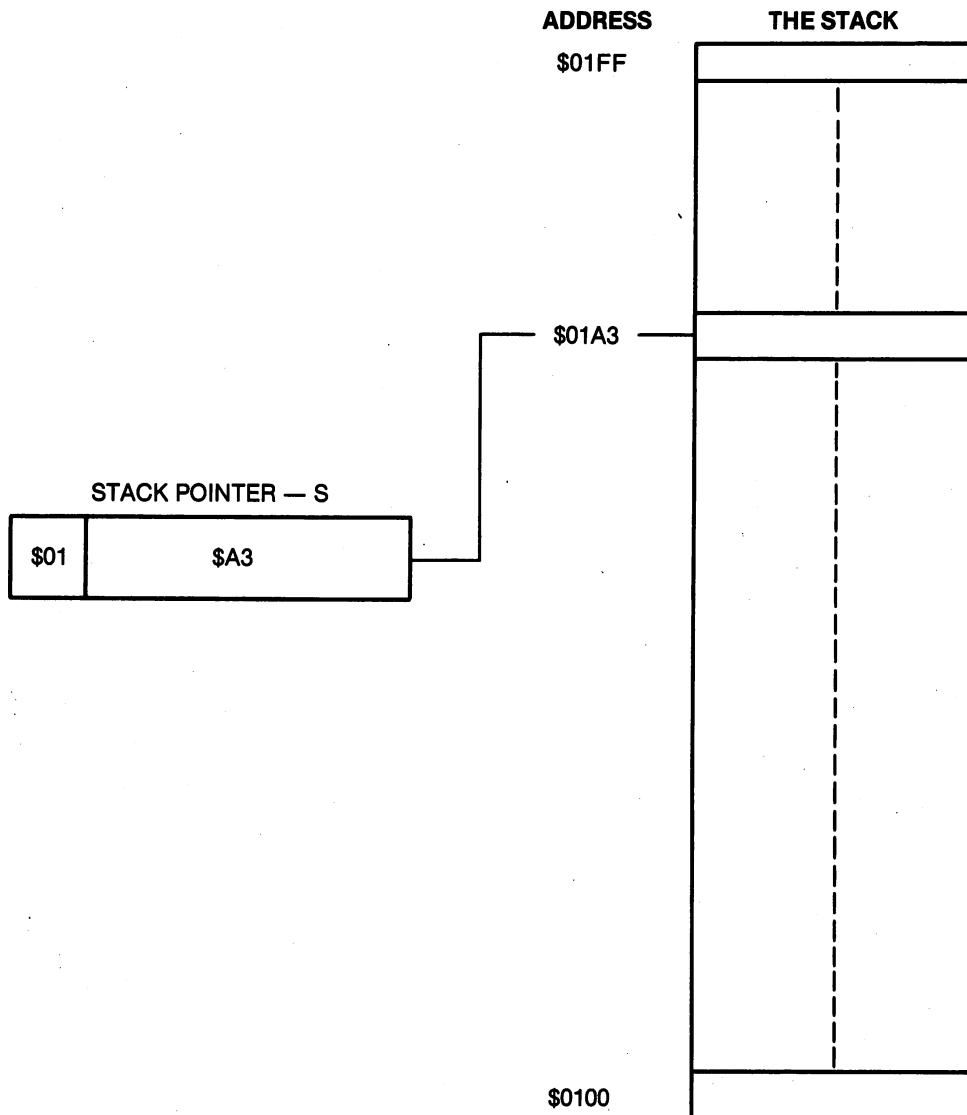


Figure 9-1. Diagram of the stack. The stack is page one of memory. The stack pointer S is a 6510 register that identifies one of the 256 memory locations on the stack. The number in S identifies the least-significant byte of a page-one memory location.

Thus, it is the address of the *third byte* of the JSR instruction that is stored on the stack as return address information. You probably expected that the 6510 would store the address of the next op code following the JSR instruction. But because the program counter is incremented after each program byte is fetched, the proper address to store is the one that identifies the location of the *third byte* of the JSR instruction.

What is the value of the stack pointer after a JSR and an RTS instruction? It has exactly the same value it had before the subroutine call. You will see that *all stack operations must occur in pairs*. For example, every JSR requires an RTS. Calamitous consequences may occur when a programmer writes programs that violate this principle.

We can now briefly summarize the JSR and RTS instructions:

- The JSR instruction pushes the address of its third byte on the stack in the order ADH, ADL. The second and third bytes of the JSR go into the program counter in the order PCL, PCH.
- The RTS instruction pulls two bytes from the top of the stack and places them in the program counter in the order PCL, PCH. The program counter is incremented.

We conclude this section by addressing several minor concerns. What happens in the case of *nested* subroutines, that is, when one subroutine calls another? The stack pointer will be decremented twice when the first JSR is executed. Another JSR instruction is encountered before an RTS instruction, and the stack pointer is decremented twice more. The stack is now *four deep* and there are two return addresses (four bytes) on the stack. Next, the first RTS instruction is executed, and the second return address on the stack is the first return address off the stack. The stack is now only two deep. Finally, the second RTS instruction is executed, and the stack returns to its original condition.

It should be clear that two bytes of stack memory are required for each subroutine. If the subroutines are nested six deep, at least 12 bytes of stack storage are required.

What happens when the stack pointer decrements to \$00, identifying the stack location \$0100? The next time the stack pointer is decremented it becomes \$FF, pointing to the location \$01FF. In other words, nothing significant happens when the stack pointer decrements through zero.

In the unlikely case, however, that you nest your subroutines more than 128 deep, requiring more than 256 bytes of stack storage, then you are in serious trouble. The last subroutine call wipes out the first return address, and your program will surely crash. In short, the number of stack locations required by your programs must not exceed 256, the number of memory locations in page one; otherwise, expect disaster.

V. Stack Operation Instructions

The stack may be used for other purposes besides storing return addresses for subroutine calls. Two instructions, PHA and PHP, are used to store information on the stack, while two other instructions, PLA and PLP, are used to recall information from the stack. We define each of these instructions below. The notation " M_s " symbolizes a memory location M on the stack that is identified by the stack pointer S.

- PHA—Push the code in the accumulator on the stack. Decrement the stack pointer.
Symbolically: $A \rightarrow M_s$, $S = S - 1$.
- PHP—Push the code in the processor status register on the stack. Decrement the stack pointer.
Symbolically: $P \rightarrow M_s$, $S = S - 1$.
- PLA—Increment the stack pointer and pull the number from the stack and place it in the accumulator.
Symbolically: $S = S + 1$, $M_s \rightarrow A$.
- PLP—Increment the stack pointer and pull the number from the stack and place it in the P register.
Symbolically: $S = S + 1$, $M_s \rightarrow P$.

The PHA and PLA instructions are illustrated in Figures 9-2 and 9-3, respectively. A diagram of the PHP and PLP instructions would be identical to Figures 9-2 and 9-3, respectively, if the 6510 status register were to replace the accumulator. Notice that the stack pointer is decremented *after* a push (PHA, PHP) operation. The stack pointer is incremented *before* a pull (PLA, PLP) operation.

Use of the PHA and PLA instructions in a program is illustrated in Example 9-3. This program is identical in function to the PRBYTE routine in Example 7-3, and you should compare it with the latter example. Notice that instead of using precious zero-page memory storage, we use the stack. Compare lines 10 and 28 of Example 7-3 with lines 10 and 28 of Example 9-3. Instead of using the location TEMP to save the number in the accumulator, in Example 9-3 we have chosen to use the stack. The stack is generally used to save the code in a register during a subroutine call.

Example 9-3. Illustration of the PHA and PLA Instructions

Object: Output a byte of data in the accumulator as two hexadecimal digits. Return with the accumulator contents preserved.

10	PRBYTE	PHA	;PUSH A ON STACK. S=S-1.	C813 48
11		PHA	;SAVE 2ND COPY OF A ON STACK. S=S-1	C814 48
12		LSR A	;SHIFT THE NUMBER IN THE	C815 4A
13		LSR A	;ACCUMULATOR FOUR BITS TO THE RIGHT.	C816 4A
14		LSR A	;MOST SIGNIFICANT NIBBLE IS ZERO.	C817 4A
15		LSR A	;NUMBER IS NOW IN LOW NIBBLE.	C818 4A
16		JSR HEXCII	;CONVERT NUMBER TO ASCII.	C819 20 00 C8
17	;	JSR CHROUT	;OUTPUT ITS CODE TO THE SCREEN.	C81C 20 D2 FF
18	;			
19	;			
20		PLA	;S=S+1. PULL 2ND COPY FROM STACK.	C81F 68
21		AND #\$0F	;MASK MOST SIGNIFICANT NIBBLE.	C820 29 0F
22		JSR HEXCII	;CONVERT SECOND DIGIT TO ASCII.	C822 20 00 C8
23		JSR CHROUT	;OUTPUT ITS CODE.	C825 20 D2 FF
24	;			
25	;			
26		LDA #\$20	;OUTPUT ASCII SPACE BY SENDING	C828 A9 20
27		JSR CHROUT	;SPACE CODE TO OUTPUT ROUTINE.	C82A 20 D2 FF
28		PLA	;S=S+1. PULL A FROM THE STACK.	C82D 68
29		RTS		C82E 60

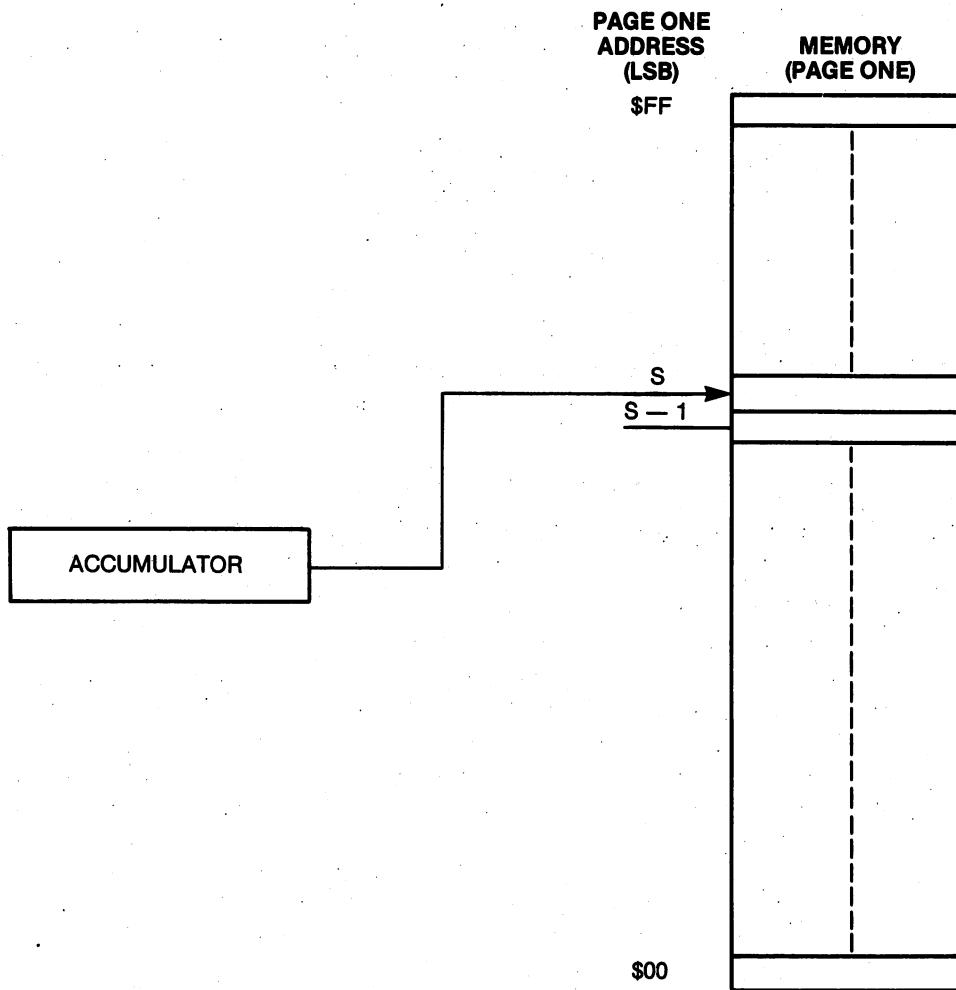


Figure 9-2. Diagram of the PHA instruction. S is the number in the stack pointer before the PHA instruction is executed. S – 1 is the number in the stack pointer after the PHA instruction is executed.

It is also convenient to use the stack for temporary storage of a code. Refer to lines 11 and 20 in Example 9-3. On line 11, the code in the accumulator is saved on the stack because we will need this code again for the instructions on lines 21 through 23.

How many stack storage locations does the subroutine in Example 9-3 require? Two locations are required for the JSR instruction that calls the subroutine. The routine also requires one stack location for each of the two

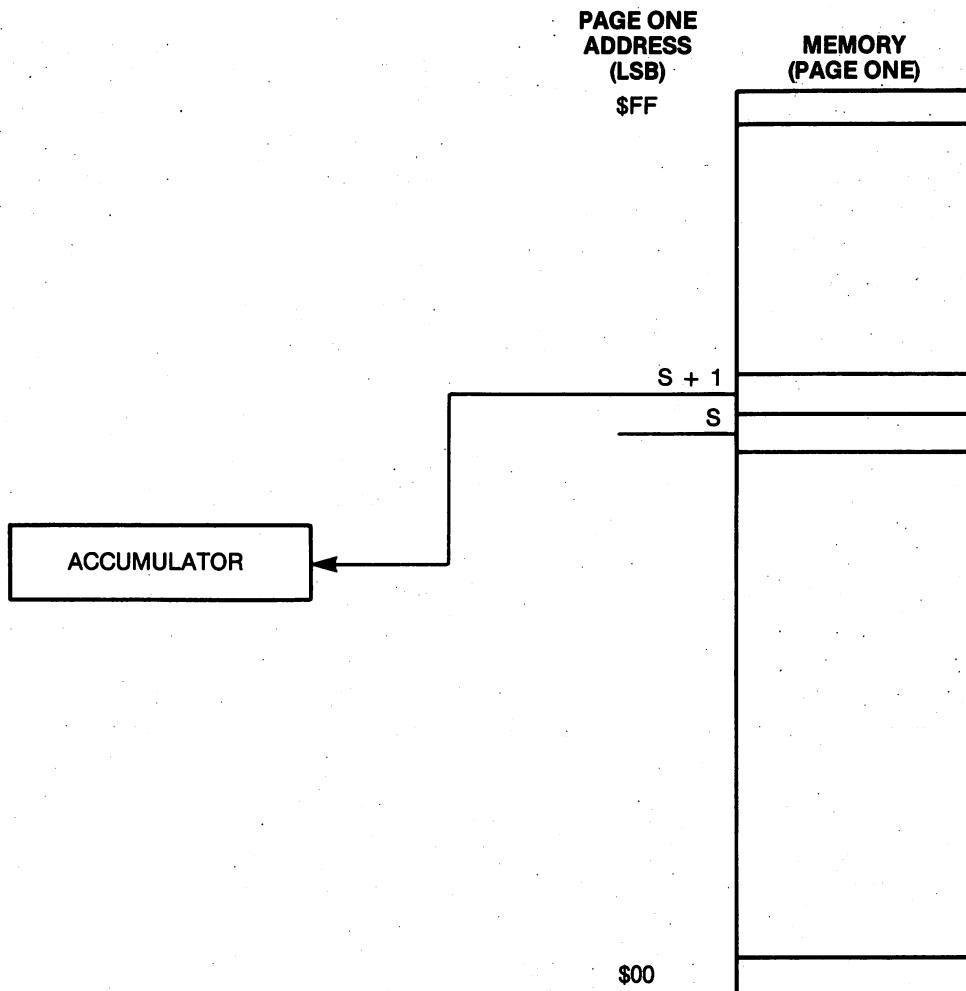


Figure 9-3. Diagram of the PLA instruction. S is the number in the stack pointer before the PLA instruction is executed. S + 1 is the number in the stack pointer after the PLA instruction is executed.

PHA instructions, giving a total of four stack locations. Notice that for each PHA there is a PLA; in other words, the stack operations must occur in pairs. Try running a program for which this is not true and observe what happens.

A programmer must be very careful that the memory locations sacred to a program are not modified by some inconspicuous subroutine. It is good practice to make a list of memory assignments for each program and subroutine, to make sure there are no conflicts. Of course, in certain cases it is desirable for both the main program and a subroutine to make use of the same memory locations. In fact, that is one of the simplest ways to pass information from the main program to the subroutine and back again.

Another problem arises when both the main program and one of its subroutines need to use the same 6510 registers. For example, the operating system subroutine SCNKEY modifies the A, X, Y, and P registers. Example 9-4 indicates how these registers may be restored, intact, after the subroutine. Line 10 saves the P register. Line 11 saves the accumulator, A. Lines 12 and 13 save the X register, and lines 14 and 15 save the Y register.

Refer, next, to lines 19 through 24. These six lines restore the registers to their values before subroutine SCNKEY was called. Observe that the order in which the registers are restored is the *reverse* of the order in which they were saved. Also observe that the stack operations occur in pairs. A subroutine that preserves the value of certain registers is said to be *transparent* to those registers.

Example 9-4. Saving and Restoring Registers

Object: Call subroutine SCNKEY without modifying the 6510 registers.

10	SAVE	PHP	; PUSH P ON STACK. S=S-1.	C000 08
11		PHA	; PUSH A ON STACK. S=S-1.	C001 48
12		TXA	; TRANSFER X TO A.	C002 8A
13		PHA	; PUSH A (X) ON STACK. S=S-1.	C003 48
14		TYA	; TRANSFER Y TO A.	C004 98
15		PHA	; PUSH A (Y) ON STACK. S=S-1.	C005 48
16		;		
17		JSR SCNKEY	; JUMP TO SUBROUTINE SCNKEY.	C006 20 87 EA
18		;		
19	RESTORE	PLA	; S=S+1. PULL A (Y) FROM STACK.	C009 68
20		TAY	; TRANSFER A TO Y.	C00A A8
21		PLA	; S=S+1. PULL A (X) FROM STACK.	C00B 68
22		TAX	; TRANSFER A TO X.	C00C AA
23		PLA	; S=S+1 ;PULL A FROM THE STACK.	C00D 68
24		PLP	; S=S+1. PULL P FROM THE STACK.	C00E 28
25		RTS	; RETURN WITH ALL REGISTERS INTACT.	C00F 60

By now you can see that the stack and the stack pointer are similar to an elevator in an office building. On the way down, the elevator unloads one item on each floor; on the way up, it picks up the same item.

Two other stack operation instructions need to be mentioned, although they are used infrequently. The TSX instruction transfers the number in the stack pointer to the X register. The TXS instruction transfers the number in the X register to the stack pointer. It is possible to locate and read various codes on the stack with these instructions. Their use will be illustrated when we discuss interrupts, the next topic of this chapter.

VI. IRQ-Type Interrupts*

In addition to being able to execute a routine from any point in a program by jumping to the routine with a JSR instruction, it is also possible to *interrupt* the execution of a program to run another program segment called an *interrupt routine*. To do this requires some events that occur in the *hardware*

of the Commodore 64. In brief, a certain signal on one of the 40 pins of the 6510 will cause it to jump out of the program it is executing, execute a program segment called an interrupt routine, and then return to continue running the program that was interrupted.

There are two kinds of interrupts that the 6510 is designed to handle, and associated with each is a corresponding pin on the microprocessor. One kind of interrupt we will call an IRQ-type interrupt, and the other we will call an NMI-type interrupt. These are related to the pins on the 6510 microprocessor that are designated IRQ and NMI, respectively. The letters "IRQ" stand for *interrupt request*, while the letters "NMI" stand for *non-maskable interrupt*. The Commodore 64 operating system uses an interrupt scheme to input information from its keyboard. We will begin by describing the IRQ-type interrupt.

An IRQ-type interrupt is *requested* when the voltage on the IRQ pin on the 6510 microprocessor changes from five volts to zero volts and stays at zero for at least six clock cycles. Figure 9-4 describes the signal on the IRQ pin that is used to request an interrupt. The duration of the interrupt request signal should not exceed the time it takes to execute the interrupt routine. The interrupt request will be *recognized* only if the I flag in the processor status register is *clear*. The I flag is bit two in the processor status register. If the I flag is set, the interrupt request will be ignored.

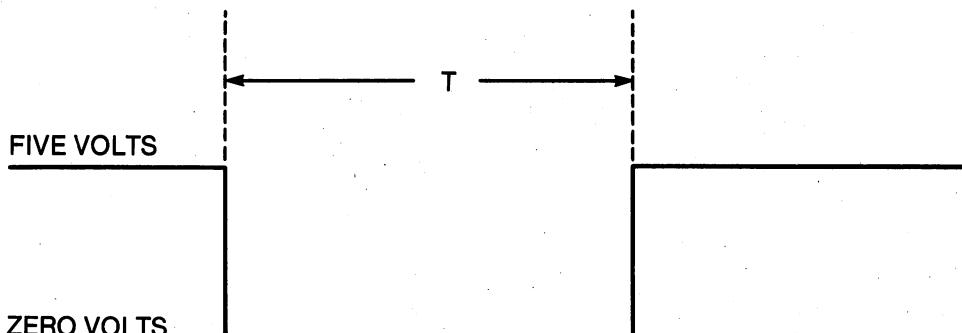


Figure 9-4. The interrupt request (IRQ) signal. The duration T of the zero-voltage condition should be at least six clock cycles but not longer than the time it takes to process the interrupt routine.

Refer to Table 9-1 and observe that a CLI instruction will clear the I flag, allowing the recognition of interrupt request signals on the IRQ pin. An SEI instruction will set the I flag, preventing the recognition of interrupt request signals. In a moment we will illustrate these instructions.

These are the events that follow the *recognition* of an IRQ signal:

- The instruction currently being executed is completed. (At this point, the 16-bit number in the program counter identifies the address of the *next* instruction in the program.)
- The high-order byte of the program counter, PCH, is pushed on the stack and the stack pointer, S, is decremented.

- The low-order byte of the program counter, PCL, is pushed on the stack and the stack pointer is decremented.
- The code in the P register is pushed on the stack and the stack pointer is decremented.
- The I flag in the P register is set to disable further interrupts.
- The 6510 replaces the number in its PCL with the number in \$FFFE.
- The 6510 replaces the number in its PCH with the number in \$FFFF.
- Execution continues with the op code whose address was stored in locations \$FFFE (LSB) and \$FFFF (MSB).

The number stored in locations \$FFFE and \$FFFF is called the *IRQ vector*. The IRQ vector is the *address* of the first op code in the series of instructions called the *IRQ routine*. In short, when an interrupt request is recognized, the program is interrupted, and execution continues at the location pointed to by the interrupt vector.

The last instruction in the interrupt routine must be an RTI instruction. Execution of the RTI instruction means:

- The stack pointer is incremented and the code found in this stack location is placed in the P register, restoring it to the value it had before the program was interrupted. (This ensures that the I flag will be cleared after the interrupt routine is processed.)
- The stack pointer is incremented and the number found in this stack location is placed in the PCL.
- The stack pointer is incremented and the number found in this stack location is placed in the PCH.
- Execution continues with the op code found in the location whose address is PCH:PCL. This will be the address of the first instruction following the instruction that was interrupted.

Let us reinforce these concepts with some simple examples. Suppose the program sequence being executed is

```
LDA NUM1    $CDEF AD 00 03
BEQ PAST    $CDF2 F0 3F
```

and the interrupt occurs *during* the LDA NUM1 instruction. Assume that the I flag is clear when the interrupt occurs. Suppose that the stack pointer is \$A3 before the interrupt occurs, and suppose the interrupt routine is located at \$1234. What happens when the interrupt signal occurs and how will the stack be used in this case?

The LDA NUM1 instruction will be completed, that is, the number stored \$0300 will be loaded into the accumulator. The MSB, \$CD, of the program counter is stored on the stack at location \$01A3, and the LSB, \$F2, of the program counter is stored at location \$01A2. The stack pointer is decremented again and the code in the P register is saved at location \$01A1 on the stack. After this, the stack pointer is \$A0. Notice that the *address* stored on the stack identifies the BEQ PAST instruction.

Continuing with this same example, the I flag will be set, and the 6510 will look in locations \$FFFE and \$FFFF to find where the interrupt routine is located. Since we are assuming that the interrupt routine begins at location

\$1234, the number \$34 must be stored in location \$FFFE and the number \$12 must be stored in location \$FFFF. The number \$1234 is the interrupt vector. The processor fetches the interrupt vector and executes the interrupt routine.

When the RTI instruction in the interrupt routine is executed, the stack pointer is incremented to \$A1 and the code stored at location \$01A1 is placed in the P register. Among other things, this clears the I flag. The stack pointer is incremented to point to location \$01A2 and the number, \$F2, stored there is placed in the PCL. The stack pointer is incremented for the last time, and the number stored at location \$01A3, \$CD, is placed in the PCH. Execution continues with the BEQ PAST instruction located at \$CDF2.

Note that any P register flags that were modified by the LDA NUM1 instruction will have the same value they had after its execution, even though an interrupt routine containing many instructions was executed between the LDA NUM1 instruction and the BEQ PAST instruction. The obvious reason for this is that the P register was saved on the stack during execution of the interrupt routine.

Before describing the interrupt handling routines that are unique to the Commodore 64, we must introduce the BRK instruction. Execution of the BRK instruction by the 6510 microprocessor causes it to execute the IRQ-type interrupt routine. In other words, you can produce an IRQ-type interrupt with *software*. In this case the sequence of events described above will take place *without* a signal on the IRQ pin of the 6510. In the case of a software-forced interrupt, however, the B (break) flag in the processor status register will be set *before* the P register is saved. The B flag will be cleared with the execution of an RTI instruction. Also, a BRK instruction is *not* disabled when the I flag is set.

One peculiarity is associated with the BRK instruction. On return from interrupt, the program counter will contain an address that is larger by *two* than the address of the BRK instruction. Since the BRK instruction requires only one byte, any op code that immediately follows it will be missed. Thus, the location after the BRK instruction can contain anything. The first op code following the BRK instruction should be two locations higher in memory. Thus, if a break instruction is located at \$CBA9, the next op code to be executed after the interrupt routine must be located at \$CBAB.

VII. The Commodore 64 Interrupt Structure*

To use interrupts on the Commodore 64, you must understand its interrupt handling routines. An explanation follows.

When you first apply power to the Commodore 64, it switches its operating system into the memory locations at the top of memory. The operating system is located in ROM, so the IRQ vector is in ROM. If you examine locations \$FFFE and \$FFFF, you find the numbers \$48 and \$FF, respectively. Thus, the Commodore 64 *interrupt vector* is \$FF48, and the interrupt routine begins at \$FF48.

A disassembled portion of the program that begins at \$FF48 is shown in Table 9-4. Study this carefully. The first PHA instruction saves the number in the accumulator during each interrupt. The next two instructions in Table 9-4 save the number in the X register on the stack. The two instructions that follow save the number in the Y register.

Table 9-4. Disassembled form of part of the Commodore 64 IRQ-type interrupt.

EQU+\$FF48	FF48
PHA+	FF48 48
TXA+	FF49 8A
PHA+	FF4A 48
TYA+	FF4B 98
PHA+	FF4C 48
TSX+	FF4D BA
LDA+\$0104,X	FF4E BD 04 01
AND#\$10	FF51 29 10
BEQ+@+5 FF58	FF53 F0 03
JMP+(\$0316)	FF55 6C 16 03
JMP+(\$0314)	FF58 6C 14 03

Now comes a tricky move on the part of the programmer who wrote this routine. The P register was the last item stored on the stack when the IRQ-type interrupt occurred. Notice in Table 9-4 that the interrupt routine begins with three stack-push operations and, of course, the stack pointer is decremented after the third push. Thus, the code in the P register is stored on the stack *four* locations up from the location identified by the stack pointer.

After transferring the stack pointer to the X index register with the TSX instruction, the program reads the code that was stored when the P register was pushed on the stack. The LDA \$0104,X instruction does this. The "4" in this instruction comes from the fact that the P register is currently stored four locations above the location currently identified by the stack pointer.

In short, the code in the P register just before the interrupt is now in the accumulator. If the interrupt was caused by a BRK instruction, then bit four, the break flag, in the P register will be set; otherwise, the B flag will be cleared. The AND #\$10 instruction in Table 9-4 masks all of the bits except the B flag bit. Thus:

- If the interrupt was produced by a BRK instruction, the JMP (\$0316) instruction is executed.
- If the interrupt was produced by a signal on the IRQ pin, the JMP (\$0314) instruction is executed.

Hence, the first portion of the interrupt routine saves the A, X, and Y registers, as well as modifying the program flow, depending on whether a software or hardware interrupt occurred.

The JMP instructions in Table 9-4 are both indirect jumps to locations in R/W memory. Users can, therefore, modify these jump vectors to point to their own interrupt or break routines.

When you apply power to the Commodore 64, it will set the IRQ indirect jump vector to \$EA31 and the BRK indirect jump vector to \$FE66. You may wish to disassemble the routines starting at these locations to see what happens during an interrupt or a BRK. Among other things, the IRQ interrupt *scans the keyboard* to see if a key has been pressed. If so, the ASCII code corresponding to the key is stored in a buffer. A timer in the 6526 CIA #1 produces interrupts at a rate of 60 Hz, and therefore the keyboard is scanned 60 times every second.

We do not wish to pursue this further because we have enough information to begin writing our own interrupt routines, which will give us a better idea of how interrupts work. The interrupt routine used by the Commodore 64 operating system is very complex, and we will not delve more deeply into it. Instead, we will make use of the SCNKEY subroutine in the operating system to design our own interrupt-driven keyboard routine.

VIII. Using Interrupts to Read the Keyboard*

To illustrate these concepts, we will provide a programming example in which the keyboard on the Commodore 64 is read on an interrupt basis. For the moment, we will not be concerned with how the 6526 CIA produces interrupts: we will merely accept the fact that this chip produces an IRQ signal on the IRQ pin of the 6510 at a rate of 60 Hz. The form of the interrupt signal is illustrated in Figure 9-4. The 6526 CIA switches the IRQ pin to zero volts when it produces an interrupt request. Near the conclusion of the interrupt routine, the signal is switched back to five volts by reading location \$DC0D.

Our program is listed in Example 9-5 and a flowchart of this program is shown in Figure 9-5. The so-called main program begins by setting the interrupt flag. We do not wish any interrupts to be recognized while we are modifying the indirect jump vector in the Commodore 64 interrupt routine. Refer to Table 9-4 and our discussion in the previous section. Once we have modified the indirect jump vector with lines 12 through 15 in the main program, the interrupt flag is cleared to allow interrupts, and the program loops endlessly, unless interrupted, in the JMP HERE loop on line 17.

As we described above, the interrupt routine starts at \$FF48 in the Commodore 64 operating system. Then, by means of the indirect jump, it can be made to jump to our routine starting at location \$C100. The listing of the IRQ routine starts on line 22 of Example 9-5. SCNKEY is an operating system subroutine that scans the keyboard. If a key is pressed, it stores the keycode at location \$0277 + X, and then increments X, which it saves at location \$00C6. If a key is not pressed, X will not be incremented. The initial value of X is zero. Thus, if X remains at zero, no key was pressed

and the branch on line 24 produces a quick exit from the interrupt routine. On the way out, the registers are restored with lines 30 through 34. Recall that the Commodore 64 routine saves the registers (see Table 9-4).

If a key is pressed, the keycode is read by loading the code in \$0277 into the accumulator. The operating system subroutine CHROUT is called to output the character to the screen. The interrupt routine is then completed by restoring the registers and executing the RTI instruction.

Example 9-5. An Interrupt Routine to Read the Keyboard

Object: Read the Commodore 64 keyboard on an interrupt basis and output the characters to the screen.

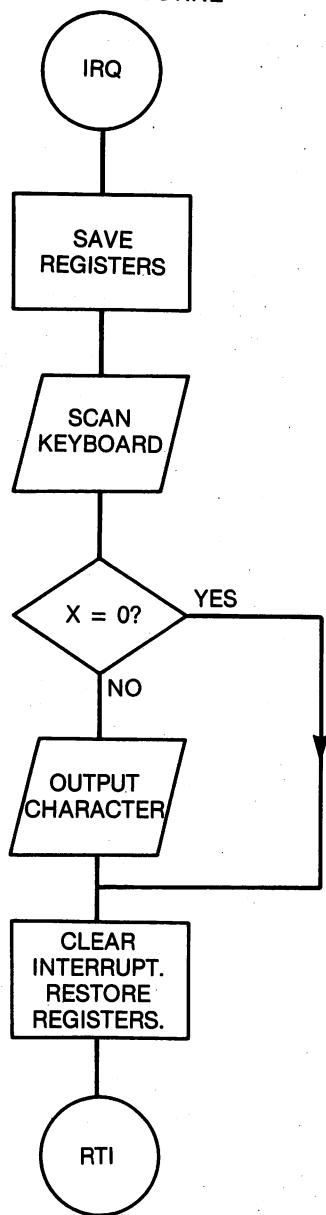
10	MAIN PROGRAM		
11	MAIN SEI	;DISABLE IRQ WITH I=1.	C000 78
12	LDA #00	;SET UP INDIRECT JUMP VECTOR TO	C001 A9 00
13	STA USRVTR	;POINT TO USER INTERRUPT ROUTINE.	C003 8D 14 03
14	LDA #\$C1	;INDIRECT JUMP VECTOR IS SC100.	C006 A9 C1
15	STA USRVTR+1	;VECTOR STORED AT \$0314-\$0315.	C008 8D 15 03
16	CLI	;ENABLE IRQ WITH I=0.	C00B 58
17	HERE JMP HERE	;STAY IN THIS INFINITE LOOP.	C00C 4C 0C C0
18	;		
19	;		
20	EQU SC100		C100
21	;INTERRUPT ROUTINE.		
22	IRQ JSR SCNKEY	;SCAN THE KEYBOARD.	C100 20 87 EA
23	LDX +\$C6	;X=0 IF NO KEY HAS BEEN PRESSED.	C103 A6 C6
24	BEQ OUT	;CONCLUDE INTERRUPT ROUTINE.	C105 F0 09
25	DEX	;RETURN X TO ZERO.	C107 CA
26	STX +\$C6		C108 86 C6
27	LDA \$0277	;GET THE CHARACTER CODE.	C10A AD 77 02
28	JSR CHROUT	;OUTPUT IT.	C10D 20 D2 FF
29	OUT LDA IRQCTL	;RESET THE IRQ SIGNAL TO 5V.	C110 AD 0D DC
30	PLA	;RESTORE THE REGISTERS BEGINNING	C113 68
31	TAY	;WITH THE Y REGISTER.	C114 A8
32	PLA	;PULL A (X) FROM THE STACK.	C115 68
33	TAX	;TRANSFER A TO X.	C116 AA
34	PLA	;FINALLY, GET A FROM THE STACK.	C117 68
35	RTI	;RETURN FROM INTERRUPT.	C118 40

Load the program in Example 9-5. Execute it with a SYS 49152 command. Type on the keyboard and observe the results. Study the program in conjunction with the flowchart in Figure 9-5. Make sure you understand how it works.

The program in Example 9-6 is another program to read the keyboard on an interrupt basis. A flowchart is given in Figure 9-6. It differs from the program in Example 9-5 in several important ways. First, the interrupt routine stores the character codes in a memory buffer consisting of one page of memory. The buffer extends from \$CF00 to \$CFFF. Each time a character is entered it is stored in this buffer and the pointer, symbolized by IAL, is incremented to point to the next location in the buffer.

In the main program, the buffer is emptied. The first code placed in the buffer will be the first code taken out of the buffer. This kind of buffer is called a *first-in, first-out (FIFO)* buffer. In Example 9-6, we decided not to output the character to the screen. Instead, we decided to output the hexadeciml code using subroutine PRBYTE in Example 9-3. In this way, you can determine the codes that correspond to each key.

INTERRUPT ROUTINE



MAIN PROGRAM

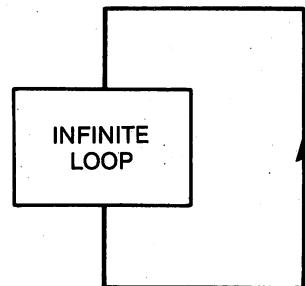


Figure 9-5. Flowchart of the program in Example 9-5.

Study the program, the comments, and the flowchart. Ideas similar to these are used frequently for input and output of information, so it will be worth your while to come to grips with these programs. Notice that the (IND), Y addressing mode is used to fill the buffer in the interrupt routine and to empty the buffer in the main program. Run the program and make a table of characters and keycodes.

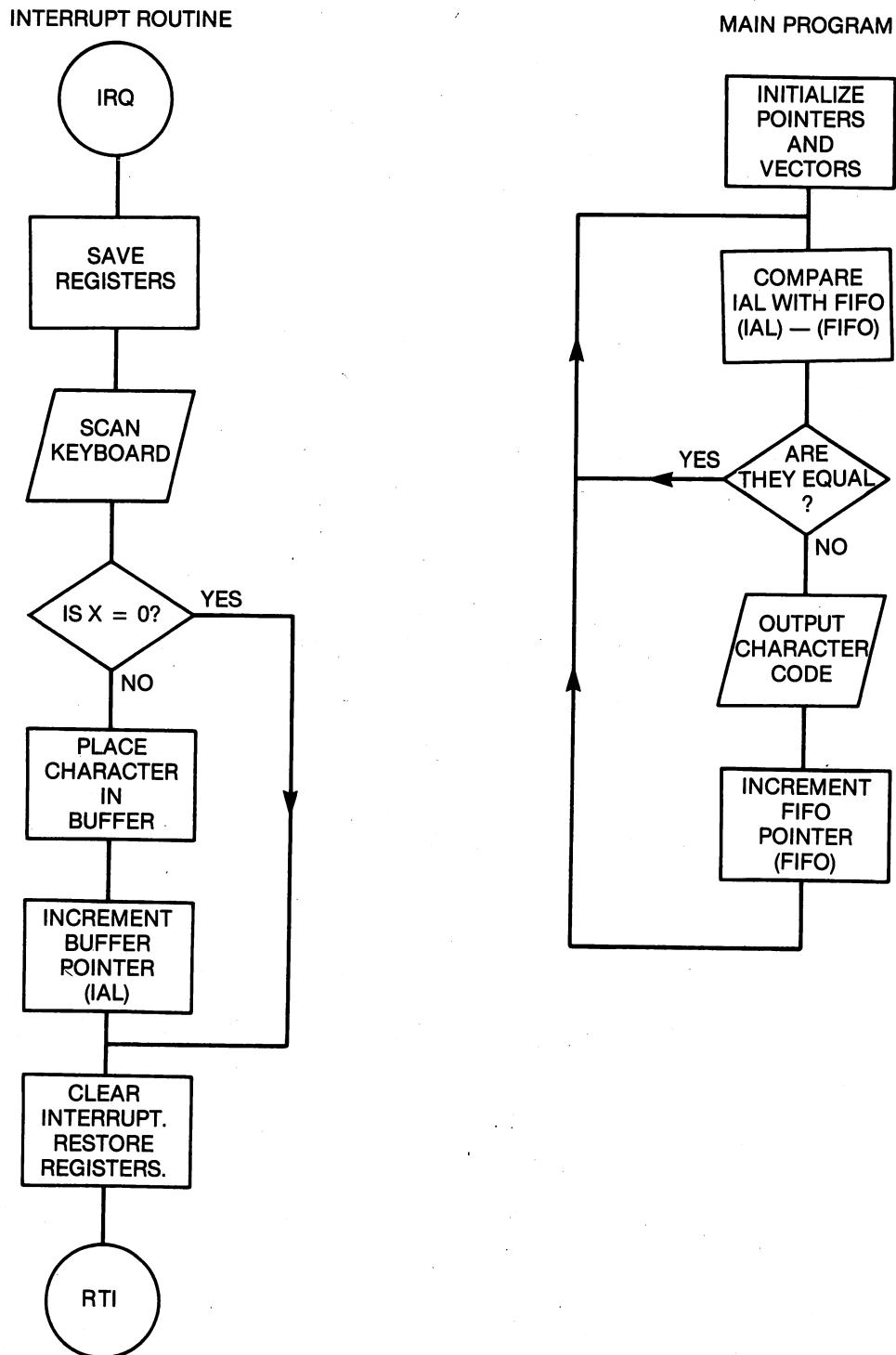


Figure 9-6. Flowchart of the program in Example 9-6.

Example 9-6. An Interrupt-Driven Keyboard with a Buffered Input

Object: Read the keyboard on an interrupt basis and store the codes in a one-page buffer. The main program should output the codes from the buffer.

```

11      ;MAIN PROGRAM
12 MAIN   SEI          ;DISABLE IRQ WITH I=1.           C000 78
13       LDA #$C1        ;SET UP INDIRECT JUMP VECTOR TO    C001 A9 C1
14       STA USRVTR+1   ;POINT TO USER INTERRUPT ROUTINE.    C003 8D 15 03
15       LDA #$00        ;INDIRECT JUMP VECTOR IS $C100.     C006 A9 00
16       STA USRVTR    ;VECTOR STORED AT $0314-$0315.    C008 8D 14 03
17       STA +IAL       ;SET UP INDIRECT INDEXED POINTERS.  C00B 85 FB
18       STA +FIFO      ;SET UP FIFO BUFFER.                C00D 85 FD
19       LDA #$CF        ;FIFO BUFFER WILL BE LOCATED      C00F A9 CF
20       STA +IAL+1    ;FROM $CF00 TO $CFFF.             C011 85 FC
21       STA +FIFO+1   ;ADVANCE THE POINTER.            C013 85 FE
22       CLI          ;ENABLE IRQ WITH I=0.            C015 58
23 LOOP   LDA +IAL     ;DO BUFFER POINTERS MATCH.        C016 A5 FB
24       CMP +FIFO      ;IS (IAL) += (FIFO)?            C018 C5 FD
25       BEQ LOOP      ;YES, SO WAIT IN THIS LOOP.      C01A F0 FA
26       LDY #0         ;USE (IND),Y MODE WITH Y=0.      C01C A0 00
27       LDA (FIFO),Y   ;GET CODE FROM THE BUFFER.      C01E B1 FD
28       JSR PRBYTE    ;PRINT THE BYTE IN THE BUFFER.    C020 20 13 C8
29       INC +FIFO      ;ADVANCE THE POINTER.            C023 E6 FD
30       CLV          ;FORCE A BRANCH.              C025 B8
31       BVC LOOP      ;GO BACK TO CHECK THE BUFFER.    C026 50 EE
32 ;
33 ;
34       EQU $C100      ;INTERRUPT ROUTINE.            C100
35 IRQ    ;INTERRUPT ROUTINE.
36       JSR SCNKEY    ;SCAN THE KEYBOARD.           C100 20 87
37       LDX +$C6        ;X=0 IF NO KEY HAS BEEN PRESSED. C103 A6 C6
38       BEQ OUT        ;CONCLUDE INTERRUPT ROUTINE.    C105 F0 0C
39       DEX          ;RETURN X TO ZERO.            C107 CA
40       STX +$C6        ;GET THE CHARACTER CODE.        C108 86 C6
41       LDA $0277      ;USE (IND),Y MODE WITH Y=0.      C10A AD 77 02
42       LDY #0         ;STORE CODE IN THE BUFFER.      C10D A0 00
43       STA (IAL),Y   ;INCREMENT THE POINTER.        C10F 91 FB
44       INC +IAL       ;RESET THE IRQ SIGNAL TO 5V.    C111 E6 FB
45 OUT    LDA IRQCTL   ;RESTORE THE REGISTERS BEGINNING C113 AD 0D DC
46       PLA          ;WITH THE Y REGISTER.          C116 68
47       TAY          ;PULL A (X) FROM THE STACK.    C117 A8
48       PLA          ;TRANSFER A TO X.            C118 68
49       TAX          ;FINALLY, GET A FROM THE STACK. C119 AA
50       PLA          ;RETURN FROM INTERRUPT.        C11A 68
51       RTI          ;RTI.                         C11B 40

```

IX. NMI-Type Interrupts*

The NMI-type interrupt is similar to the IRQ-type interrupt, so we will only describe the differences.

One, an NMI-type interrupt is produced by a negative *transition* on the NMI pin on the 6510 microprocessor. This signal is illustrated in Figure 9-7.

Two, as its name implies, an NMI-type interrupt is nonmaskable. An NMI-type interrupt is *always* recognized. Setting the I flag with the SEI instruction does not prevent an NMI-type interrupt.

FIVE VOLTS

ZERO VOLTS

Figure 9-7. Signal to produce an NMI-type interrupt.

Three, the NMI vector is found in the locations with addresses \$FFFA and \$FFFB.

Other than this, NMI-type interrupts work in the same way as IRQ-type interrupts. Since different locations are used for the NMI vector, you can have either an NMI-type interrupt routine, an IRQ-type interrupt routine, or both, in different places in memory. This concludes our brief discussion of the NMI-type interrupt. It would be a good exercise for you to trace the NMI-type interrupt handling structure of the Commodore 64.

X. Summary

The 6510 microprocessor provides three ways in which you can exit one program to execute another routine. The routine can be a subroutine, an IRQ-type interrupt routine, or an NMI-type interrupt routine. A subroutine is called with a JSR instruction. An IRQ-type interrupt routine is executed with the appropriate signal on the IRQ pin or with a BRK instruction. The NMI-type interrupt routine is executed with the appropriate signal on the NMI pin.

A special portion of memory known as the stack is used to process these routines. The stack is used to store return address information and to save the contents of the various registers in the 6510.

To fully use the power of interrupts, you must understand the interrupt handling routines that are built into the operating system of the Commodore 64.

XI. Exercises

1. POKE any number into location \$0315. What happens? Why?
2. Modify subroutine GETBYT in Example 7-4 by replacing the TXA and TAX instructions with stack operation instructions.
3. Subroutine GETBYT in Example 7-4 calls the operating system subroutine GETIN that modifies A, X, and Y. How can you preserve X and Y during a call to GETIN?
4. Since the Commodore 64 provides interrupts at a rate of 60 Hz, you can make a 24-hour clock. Write a routine that increments a memory location, call it SIXTY, during each interrupt. When the number in SIXTY reaches 60, it

should be cleared and another location, call it SECNDS, is incremented. Extend this thinking to a location called MINUTS and HOURS. Do this using BCD arithmetic, and use subroutine PRBYTE to display each location. You now have a 24-hour clock program.

5. Place a BRK instruction at \$C000 and then execute it from this BASIC program:

```
10 SYS 49152 : PRINT "HELLO"
```

Explain what happens.

6. A JSR instruction is located beginning at \$89AB. If the stack pointer is \$31, what information will be stored at what locations on the stack when the JSR instruction is executed? What will the stack pointer be after the JSR instruction is executed? What numbers are stored on the stack when the JSR HEXCII instruction in Example 9-3 is executed?
7. In Example 9-5, it is always the JMP HERE instruction that is interrupted. What values for the program counter will be stored on the stack when this instruction is executed?
8. What will happen in Example 9-6 if the TAY and TAX instructions are interchanged? Try it? Why was the order TAY first, TAX second used in this program?
9. Disassemble and analyze as much of the Commodore 64 IRQ-type interrupt routine as you can.
10. Write an interrupt routine to read and display the TOD clock on the 6526 CIA during each interrupt. Refer to Example 7-7 for some hints. The main program should set the clock and start it. The interrupt routine should read and display each of the four TOD registers.

10

Programming the 6581 Sound Interface Device

I. Introduction

In the concluding three chapters of this book, we will explore some of the many applications that are made possible by the special-purpose integrated circuits (chips) found in the Commodore 64. These integrated circuits include the 6581 SID (sound interface device), the 6567 VIC (video interface chip), and the 6526 CIA (complex interface adapter). In this chapter, we will examine the music-making capabilities of the 6581 SID.

There are an extraordinary number of sound effects that can be produced with the SID. The focus of this chapter will be a series of routines that will play a song using the SID's three voices. In the process you will learn some programming techniques, and a programming style will emerge. Thus, in addition to learning about the SID, you will also be reviewing the topics covered in the previous nine chapters, and you will be exposed to a particular programming perspective.

The SID also has two analog inputs that are generally used as game paddle inputs. These inputs prove to be useful as simple analog-to-digital converters. To use the two *potentiometer* inputs, however, you must also program the 6526 CIA, so we will postpone this topic until Chapter 12.

II. A Brief Introduction to Music Synthesis

The SID is a powerful, three-voice music synthesizer, not significantly different from a Moog synthesizer. The Moog synthesizer depended on *analog voltages* to control its various modules. The modules were interconnected by numerous patch cords. On the other hand, the SID is a *digital* device, that is, the circuit modules in the SID are controlled by *numbers* that are written to its registers. There is no need for patch cords. Parameters are set and switches are thrown by writing numbers to the SID registers under the control of a

computer program. The SID is controlled by a program rather than knobs and patch cords. There are advantages and disadvantages associated with either the analog or digital technique.

If you are interested in computer music applications, you should obtain a book on music synthesis. Space permits only a brief, oversimplified introduction to the topic in this book.

There are four characteristics that define the sound perceived by you when you hear an instrument play. These are the *loudness* (volume), *pitch* (frequency), *quality* (timbre), and the *dynamics* (time varying characteristic) of a note. Of these four, loudness and pitch are probably already familiar concepts to you. You are familiar with loudness because it is a parameter that you can control on devices that reproduce music, for example, your stereo. One of the SID registers allows you to control the volume on a scale of 0 to 15, where 0 is silence.

The pitch of a sound is determined by its *frequency*. The frequency is the number of vibrations per second that the instrument makes. For example, each string on a guitar vibrates back and forth at a frequency determined by the length and mass of the string. Frequency is expressed in Hertz (Hz), and on the scale used by most music synthesizers, the A note above middle C is exactly 440 Hz. Each of the 12 notes, C, C#, D, D#, E, F, F#, G, G#, A, A#, and B, has its own pitch. These notes are repeated in *octaves* above and below this central or middle octave. For example, the A above the middle A has a frequency of 880 Hz, while the A below the middle A has a frequency of 220 Hz. Each octave corresponds to a factor of two in frequency.

To control the frequency of a tone, the SID uses two eight-bit registers as one 16-bit frequency control number. Thus, there is a MSB (most-significant byte) and a LSB (least-significant byte) for the frequency. The number written to the SID frequency-control register is *not* the frequency. The frequency is determined by the number in the register and the frequency of the Commodore 64 system clock. The formula for the frequency, F, of the note is

$$F = 0.06097 * N$$

where N is the 16-bit number in the frequency register.

Now we must pursue the more difficult topic of timbre (note quality). When a note is played on a musical instrument, such as a piano, there is more than one frequency present. Suppose you strike the middle A key. The dominant frequency, called the *fundamental*, will be 440 Hz. But there will also be some musical energy present at 880 Hz, 1,320 Hz, 1,760 Hz, and so on. These other frequencies are called *harmonics*. They are *multiples* of the fundamental frequency. An important characteristic of a sound is the amount of energy present in each of the harmonics. This characteristic is called *timbre*, or quality. Timbre is determined by the *harmonic structure* of the sound. A trumpet, saxophone, and violin produce notes with quite different harmonic structures, so that the most untrained ears can still distinguish between these instruments. Even two brass instruments, the trumpet and the French horn, for example, have a different harmonic structure. *The essential idea* behind

electronic music synthesis is the summation of harmonics from a number of electronic oscillators in the appropriate amounts to produce the desired timbre. In principle, it is possible to synthesize the tone from a Stradivarius violin. Synthesizing a specific quality turns out to be very difficult, and a different approach is used in most electronic synthesizers, including the SID.

Modifying the harmonic structure, and consequently the timbre, of a sound has a somewhat surprising effect on the *waveform*. The waveform is the amplitude of the sound as a function of time. The flute, for example, has almost no harmonic content above the fundamental frequency, and its waveform is almost a pure sine wave, one cycle of which is shown in Figure 10-1. When the A note is played, there are 440 of these sine wave cycles that occur in one second. A pure sine wave has no harmonics.

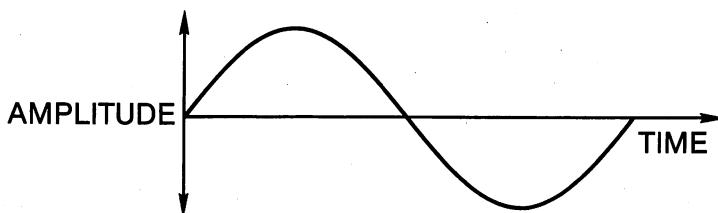


Figure 10-1. Sine waveform.

On the other hand, a triangular waveform, pictured in Figure 10-2, has some harmonics. When compared by ear, the triangle waveform will have a slightly richer quality to it than the sine waveform. An even richer tone is produced with the ramp waveform, pictured in Figure 10-3. The ramp waveform frequently is used to duplicate the sound of a stringed instrument. Another waveform with rich harmonic content is the square waveform pictured in Figure 10-4. The harmonic structure of a square waveform can be modified by changing the *pulse width*. In Figure 10-5, we picture two cycles of a square waveform with a 25 percent pulse width. It is high for 25 percent of the time and low for 75 percent of the time.

The typical electronic synthesizer *does not* combine different frequency components to produce the desired waveform. The typical synthesizer offers several possible waveforms that are then *filtered* to produce the desired harmonic content. The SID is no exception. Under program control, you can select either a triangle waveform, a ramp waveform, or a square waveform. In addition, the pulse width of the square waveform is programmable.

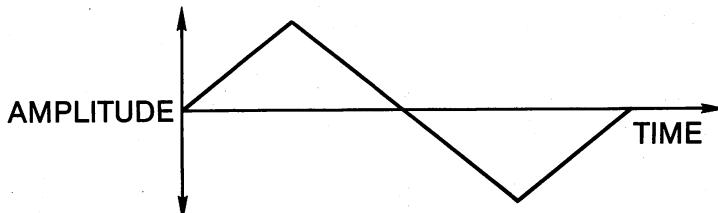


Figure 10-2. Triangle waveform.

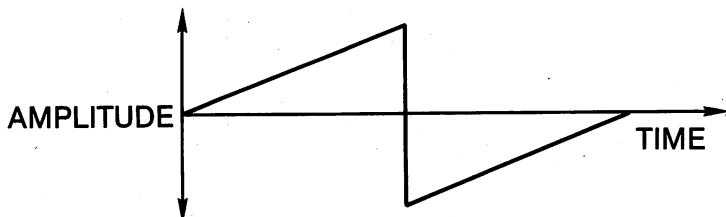


Figure 10-3. Ramp waveform.

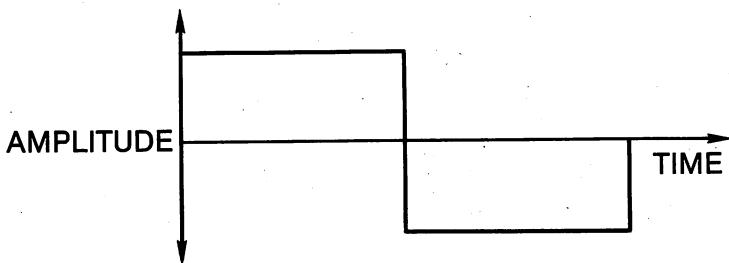


Figure 10-4. Square waveform.

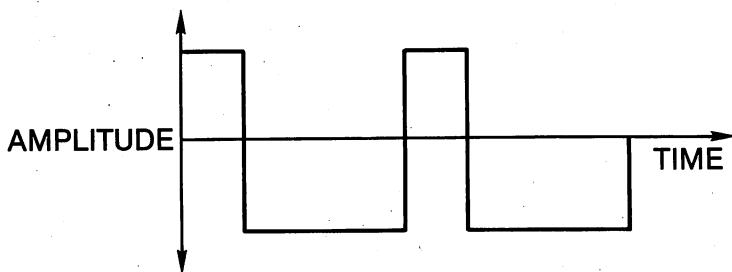


Figure 10-5. Two cycles of a square waveform with a 25 percent pulse width.

The usual approach to music synthesis is to start with a waveform that has a rich harmonic structure, such as the ramp waveform, and filter, or reject, the undesired frequencies. This is also how the SID works. The SID has a filter that can be programmed as either a *low-pass*, *band-pass*, or *high-pass filter*. A low-pass filter allows frequencies below a certain *cut-off frequency* to pass through it, but frequencies above the cut-off frequency are severely attenuated. If you apply a triangle waveform to a low-pass filter whose cut-off frequency is slightly higher than the frequency of the triangle waveform, then the fundamental will pass through the filter, but the higher harmonics will not. The result is a pure sine wave. The same effect can be produced with a ramp or square waveform.

A high-pass filter allows the frequencies *above the cut-off frequency* to pass through, and it attenuates the frequencies below the cut-off frequency. A band-pass filter allows a band of frequencies near a center frequency to pass

through, attenuating frequencies outside of the band. The SID can be programmed to behave as either a low-pass, high-pass, or band-pass filter. The cut-off frequency is programmable, as is the center frequency of the band-pass filter. A variety of interesting effects, including WAH-WAH, can be produced by varying the cut-off frequency of the filter as a note is being played.

We conclude this section with the concept of note dynamics. So far, we have discussed notes as if they were a more or less continuous sound. Each note, however, has a definite time duration, and, moreover, within this interval there is a structure to the intensity of the sound. Thus, not only are there whole notes, half notes, quarter notes, etc., each of which has a beginning and an end, but there is also an intensity structure between the beginning and the end. This is what is meant by note dynamics.

The intensity structure is frequently described in these terms: *attack*, *decay*, *sustain*, *release*. These concepts are illustrated in Figure 10-6. A diagram such as this is called an *ADSR envelope*. The sketch in Figure 10-6 illustrates a single note. The start of the note initializes the attack phase. During this period, the length of which is symbolized by A, the volume increases to a peak value. The part of the note symbolized by R is called the release. During this interval, the volume decreases to a constant level where it is sustained. The duration of the sustain interval is essentially the length of the note. When you are playing a trumpet, for example, and you run out of wind, the sustain level must come to an end. During the decay the sound volume decays quickly to zero.

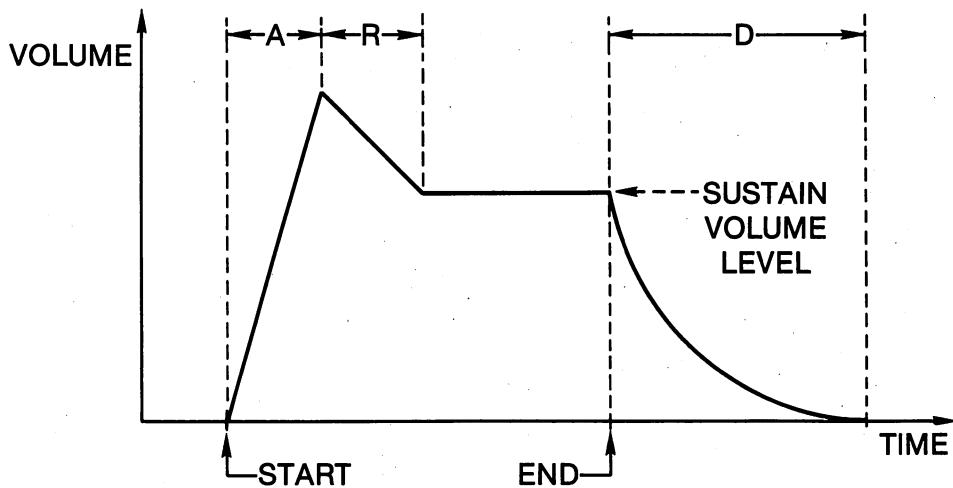


Figure 10-6. ADSR note envelope showing the attack, release, sustain, and decay phases.

A musical instrument can be recognized by its ADSR envelope. A pipe organ has attack, release, and decay times that are essentially zero. In other words, the volume rises almost instantaneously to its sustain level, remains there throughout the duration of the note, and then drops quickly to zero at the end of the note. A percussion instrument has no sustain

level. The release moves right into a rapid decay. A plucked stringed instrument, such as a guitar, has a very short attack, no sustain interval, but a very long release-decay time.

As in the case of the other parameters mentioned so far, the SID's ADSR envelope is programmable. Each of the four parameters, A, D, S, and R, is under the control of the programmer. Remember that A, D, and R are time intervals, while S is a volume level. The actual duration of the sustain interval must be controlled with either a timing loop or a timing device. We will use a simple timing loop in our programs.

To summarize, the SID gives you control over the *volume*, *frequency*, *timbre* (harmonic content of the waveform and the filter parameters), and the note *dynamics*. It does this for each of three voices that are independently programmable. Besides the three waveforms mentioned above, you can select a noise source to produce the sounds of surf, snare drums, shots, and 747's taking off.

This concludes our brief description of the basic elements of music synthesis. Perhaps it should be noted that the motivation of electronic music synthesis has gone far beyond trying to duplicate other musical instruments, and electronic sound generation has become a musical endeavor in its own right. Are the sounds associated with an arcade-style video game music?

III. The Registers of the 6581 SID Chip

Programming the SID means writing the appropriate numbers to some of its 29 registers. The registers are listed in Table 10-1. Twenty-five of the 29 SID registers are *write-only registers*, while the other four are *read-only registers*. That the registers are either write-only or read-only has important implications for programming the SID. Several of the instructions in the 6510 instruction set are so-called read-modify-write instructions. These instructions, including INC, DEC, ASL, LSR, ROL, and ROR, involve both a read and a write. They are, therefore, perfectly useless when dealing with the SID.

In the programs that follow, we will deal with this problem in the following way. An *image* of the 25 write-only registers will be maintained in the R/W memory locations whose addresses include \$CB80 through \$CB98. The codes in these locations will be processed just as if these locations were the SID registers. Read-modify-write instructions can be used. When our image registers are correctly programmed, we will simply transfer numbers in the image locations \$CB80 through \$CB98 to the registers in the SID chip, locations \$D400 through \$D418, respectively.

Since the first step in programming the SID chip is to clear all of the write-only registers, we can illustrate the ideas mentioned above with the first programming example in this chapter, Example 10-1. Example 10-1 is a subroutine to clear the image registers. In a subsequent example we will provide a subroutine to transfer all of the image register contents to the SID. Thus, in this case, clearing the SID registers involves two steps. First clear the image, then transfer the image to the SID. Notice the use of the ABS,Y addressing mode in Example 10-1.

The subroutine to transfer the image to the SID registers is listed in Example 10-2. It merely transfers the numbers in the 25 memory locations in the SID image to the SID itself.

Table 10-1. Memory map of the SID registers.

Address (Hex)	Register	Data Bit Number							
		D7	D6	D5	D4	D3	D2	D1	D0
Voice One									
\$D400	FREQ LO	F7	F6	F5	F4	F3	F2	F1	FØ
\$D401	FREQ HI	F15	F14	F13	F12	F11	F1Ø	F9	F8
\$D402	PW LO	PW7	PW6	PW5	PW4	PW3	PW3	PW1	PWØ
\$D403	PW HI	PW15	PW14	PW13	PW12	PW11	PW1Ø	PW9	PW8
\$D404	CONTROL	NOIS	SQAR	RAMP	TRNG	TEST	RING	SYNC	GATE
\$D405	ATK/DCY	ATK3	ATK2	ATK1	ATKØ	DCY3	DCY2	DCY1	DCYØ
\$D406	STN/RLS	STN3	STN2	STN1	STNØ	RLS3	RLS2	RLS1	RLSØ
Voice Two									
\$D407	FREQ LO	F7	F6	F5	F4	F3	F2	F1	FØ
\$D408	FREQ HI	F15	F14	F13	F12	F11	F1Ø	F9	F8
\$D409	PW LO	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PWØ
\$D40A	PW HI	PW15	PW14	PW13	PW12	PW11	PW1Ø	PW9	PW8
\$D40B	CONTROL	NOIS	SQAR	RAMP	TRNG	TEST	RING	SYNC	GATE
\$D40C	ATK/DCY	ATK3	ATK2	ATK1	ATKØ	DCY3	DCY2	DCY1	DCYØ
\$D40D	STN/RLS	STN3	STN2	STN1	STNØ	RLS3	RLS2	RLS1	RLSØ
Voice Three									
\$D40E	FREQ LO	F7	F6	F5	F4	F3	F2	F1	FØ
\$D40F	FREQ HI	F15	F14	F13	F12	F11	F1Ø	F9	F8
\$D410	PW LO	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PWØ
\$D411	PW HI	PW15	PW14	PW13	PW12	PW11	PW1Ø	PW9	PW8
\$D412	CONTROL	NOIS	SQAR	RAMP	TRNG	TEST	RING	SYNC	GATE
\$D413	ATK/DCY	ATK3	ATK2	ATK1	ATKØ	DCY3	DCY2	DCY1	DCYØ
\$D414	STN/RLS	STN3	STN2	STN1	STNØ	RLS3	RLS2	RLS1	RLSØ
Filter									
\$D415	FC LO	-	-	-	-	-	FC2	FC1	FCØ
\$D416	FC HI	FC1Ø	FC9	FC8	FC7	FC6	FC5	FC4	FC3
\$D417	RES/FLT	RES3	RES2	RES1	RESØ	EXT	FLT3	FLT2	FLT1
\$D418	MOD/VOL	30FF	HIPS	B DPS	LOPS	VOL3	VOL2	VOL1	VOLØ
Miscellaneous									
(Read Only)									
\$D419	POT X	PX7	PX6	PX5	PX4	PX3	PX2	PX1	PXØ
\$D41A	POT Y	PY7	PY6	PY5	PY4	PY3	PY2	PY1	PYØ
\$D41B	OSC/RND	07	06	05	04	03	02	01	0Ø
\$D41C	ENVLP 3	E7	E6	E5	E4	E3	E2	E1	E

Example 10-1. Clearing the SID Image

Object: Clear the SID image located from \$CB80 through \$CB98.

10 CLEAR	LDA #00	;CLEAR SID'S REGISTERS.	CA00 A9 00
11	LDY #\$19		CA02 A0 19
12 BR1	STA IMAGE-1,Y		CA04 99 7F CB
13	DEY		CA07 88
14	BNE BR1		CA08 D0 FA
15	RTS		CA0A 60

Example 10-2. Transferring the Image to the SID

Object: Transfer the codes in the 25 memory locations that make up the SID image to the SID.

10 TNSFER	LDY #\$19		CAE0 A0 19
11 LOOP	LDA IMAGE-1,Y	;GET IMAGE BYTE.	CAE2 B9 7F CB
12	STA SID-1,Y	;STORE IT IN SID.	CAE5 99 FF D3
13	DEY		CAE8 88
14	BNE LOOP		CAE9 D0 F7
15	RTS		CAEB 60

Refer once again to the SID register summary in Table 10-1. There are seven registers associated with each voice. These registers must be properly initialized with the frequency, pulse width (in the case of a square waveform), and ADSR envelope parameters before starting to play a note. The desired waveform must also be chosen. In a moment, we will deal with the two frequency registers, FREQ LO and FREQ HI, for each of the three voices. Let us begin by studying the CONTROL register.

Associated with each of the three voices is a control register, CONTROL. The high-order nibble of each control register is used to select the desired waveform, noise, square, ramp, or triangle. You place a one in the bit that corresponds to the waveform you want the voice to output. To begin, we suggest that you do not complicate matters by choosing more than one waveform for a given voice, but you may wish to experiment with this later.

The test bit will be of no real use to us. It stops the oscillator in a known condition for testing purposes. Consult the *Programmer's Reference Guide*, page 463, for details.

We are still focusing on the control registers. The RING and SYNC bits, when set, tie the voices together and produce unusual and complex harmonic structures. You are encouraged to experiment with these bit settings once our song routines are in place. For the time being, we will settle for the more conventional harmonies of *Auld Lang Syne*, by Robert Burns.

When switched to one, the GATE bit starts the voice. In particular, it starts the attack part of the ADSR envelope. The ADSR envelope will stay in the ADS phase as long as the gate bit is one. In other words, it will sustain the note as long as the gate bit is set. The end of a note begins when the gate bit is cleared. This event initiates the release phase of the ADSR envelope.

The ADSR envelope parameters are determined by the numbers stored in the ATK/DCY and STN/RLS registers. Refer to Table 10-1 to identify these registers, and refer to Table 10-2 to find the relationship between the numbers that you will store and the corresponding time. To choose an attack time of 24 milliseconds, store \$3 in the high-order nibble of the ATK/DCY register. To choose an equal decay time, store a \$1 in the low-order nibble of the ATK/DCY register. The STN/RLS register is treated in the same way. The sustain nibble contains a number from 0 to 15 that controls the volume of the sustain level. The release nibble in the STN/RLS register is chosen with reference to Table 10-2. To have a sustain level of 11 and a release time of 24 milliseconds, load the STN/RLS register with \$B1.

Table 10-2. Attack, release, and decay parameters.

<i>Register Value (Decimal)</i>	<i>Attack Time (Hex)</i>	<i>Release or Decay Time (Milliseconds)</i>		
0	\$0	2	6	
1	1	8	24	
2	2	16	48	
3	3	24	72	
4	4	38	114	
5	5	56	168	
6	6	68	204	
7	7	80	240	
8	8	100	300	
9	9	250	750	
10	A	500	1500	1.5s
11	B	800	2400	2.4s
12	C	1000	1s	3s
13	D	3000	3s	9s
14	E	5000	5s	15s
15	F	8000	8s	24s

Example 10-3 illustrates some of these concepts. In this example, we initialize Voice #1 to produce a triangle waveform with the ADSR parameters just mentioned. Remember, we begin by storing these numbers in the SID image. Notice that, for the time being, the gate for Voice #1 is off.

Example 10-3. Initialize Voice #1

Object: Set up Voice #1 to produce a triangle waveform with equal attack, decay, and release times of 24 milliseconds, and a sustain level of 11.

```

10 VOICE1 LDA #$10      ;SELECT TRIANGLE WAVEFORM.          CA0B A9 10
11 STA CR1       ;CONTROL REGISTER FOR VOICE 1.      CA0D 8D 84 CB
12 LDA #$31       ;SELECT ATTACK/DECAY RATE FOR      CA10 A9 31
13 STA ATKDK1    ;VOICE 1.                                CA12 8D 85 CB
14 LDA #$B1       ;SET SUSTAIN LEVEL AND RELEASE      CA15 A9 B1
15 STA STNRL1    ;RATE FOR VOICE 1.                  CA17 8D 86 CB

```

Example 10-4 is much like Example 10-3 except a ramp waveform is selected for Voice #2, and different parameters are selected for the ADSR envelope. Study Table 10-2 and identify the ADSR parameters from the program.

Example 10-4. Initialize Voice #2

Object: Set up the Voice #2 registers. Select a ramp waveform.

10	VOICE2	LDA #S20	;SELECT RAMP WAVEFORM.	CA1A A9 20
11		STA CR2	;CONTROL REGISTER FOR VOICE 2.	CA1C 8D 8B CB
12		LDA #S83	;SELECT ATTACK/DECAY RATE FOR	CA1F A9 83
13		STA ATKDK2	;VOICE 2.	CA21 8D 8C CB
14		LDA #SA3	;SET SUSTAIN LEVEL AND RELEASE	CA24 A9 A3
15		STA STNRL2	;RATE FOR VOICE 2.	CA26 8D 8D CB

In our next example, we initialize Voice #3 by choosing a square waveform. An attack time of 24 milliseconds followed by a decay time of 100 milliseconds is chosen. Notice that the sustain level is only four. The note will begin to decay to a very low volume as soon as it reaches its peak volume. A long release time, 300 milliseconds, was also chosen.

If a square waveform is selected, then a decision about the pulse width must be made, and the pulse width, PW, registers must be initialized. Continue to refer to Example 10-5. The PW registers require a 12-bit number whose low order nibble is stored in the register called PW LO (see Table 10-1). The pulse width, expressed as a percent of the entire cycle time, is given by the formula:

$$\text{PW\%} = N/40.95$$

where N is the 12-bit number stored in the PW registers. An N of \$FFF = 4,095 produces a constant output that produces no sound. A 50 percent pulse width or a perfectly square wave is obtained by choosing N = 2,048 = \$800. In Example 10-5, we chose \$900 to be our number, giving a pulse width of approximately 56 percent. You are encouraged, once again, to experiment with these values to obtain different sounds. In any case, Example 10-5 illustrates how to initialize a voice for a square waveform.

Example 10-5. Initialize Voice #3

Object: Set up Voice #3 to produce a square waveform with a pulse width of 56 percent.

10	VOICE3	LDA #S40	;SELECT PULSE WAVEFORM.	CA29 A9 40
11		STA CR3	;CONTROL REGISTER FOR VOICE 3.	CA2B 8D 92 CB
12		LDA #S38	;SELECT ATTACK/DECAY RATE FOR	CA2E A9 38
13		STA ATKDK3	;VOICE 3.	CA30 8D 93 CB
14		LDA #S48	;SET SUSTAIN LEVEL AND RELEASE	CA33 A9 48
15		STA STNRL3	;RATE FOR VOICE 3.	CA35 8D 94 CB
16		LDA #00	;SET PULSE WIDTH FOR PULSE	CA38 A9 00
17		STA PWLO3	;WAVEFORM ON VOICE 3.	CA3A 8D 90 CB
18		LDA #S09		CA3D A9 09
19		STA PWHI3		CA3F 8D 91 CB

Notice that Examples 10-3 to 10-5 follow each other in memory. We will conclude this set of programs with Example 10-6, which ends with an RTS instruction. Thus, these four programs become a single subroutine that initializes the three voices and the filter.

Again, you will need to examine Table 10-1 to understand the discussion that follows. The cut-off frequency of the filter is determined by the 11-bit number stored in FC LO and FC HI. In the case that the band-pass filter is chosen, the number in FC LO and FC HI selects the center frequency of the band-pass filter. The cut-off frequency varies from 30 Hz to 12 kHz, but the literature is inconsistent regarding the exact relationship between the number and the frequency. It appears that trial and error is called for. We found that loading the high byte of the filter with \$D0 produced some filtering, but did not cut off Voice #1 completely. That is why you find this number in Example 10-6.

The RES/FLT register performs two functions. The RES nibble enhances the frequencies near the cut-off frequency. It can produce unusual effects. A nibble value of zero produces no resonance effects, while a nibble value of 15 (\$F) produces the most enhanced resonance effects. The FLT nibble of the RES/FLT register selects which voices will go through the filter. We chose to illustrate this by routing Voice #1 through the filter, leaving the other voices unfiltered. Experiment with other options.

The VOL nibble in the MOD/VOL register determines the volume. Volume levels from 0 to 15 are possible. The MOD nibble allows you to choose the kind of filter you want—low-pass, band-pass, or high-pass—by setting the appropriate bit to one. You can even experiment with setting more than one bit. The 3OFF bit in the MOD/VOL register disconnects Voice #3 from the audio output. If Voice #3 is also disconnected from the filter, then it can be used to ring-modulate or synchronize the other voices without being heard. Example 10-6 illustrates how we fixed the filter parameters for our song.

Example 10-6. Initialize the Filter

Object: Route Voice #1 through a low-pass filter.

10	FILTER	LDA #\$00	;SELECT CUTOFF FREQUENCY.	CA42 A9 00
11		STA FCLO	;LOW THREE BITS OF FREQUENCY.	CA44 8D 95 CB
12		LDA #\$D0	;HIGH EIGHT BITS OF CUTOFF	CA47 A9 D0
13		STA FCHI	;FREQUENCY.	CA49 8D 96 CB
14		LDA #01	;PUT VOICE 1 THROUGH FILTER.	CA4C A9 01
15		STA RESFIL	;NO RESONANCE SELECTED.	CA4E 8D 97 CB
16		LDA #\$1F	;SELECT LOW PASS FILTER. SET	CA51 A9 1F
17		STA MDVOL	;VOLUME TO MAXIMUM.	CA53 8D 98 CB
18		RTS		CA56 60

Except for setting the frequency, we have completed the initialization sequence that is normally accomplished before playing notes. Of course, for many sounds you need only one voice. In the next section, we will illustrate techniques to set the frequencies so that a song can be played, but before concluding this section, we should complete our discussion of the SID registers.

The POT X and POT Y are simple analog-to-digital converter registers that we will describe in some detail in Chapter 12. The OSC/RND register contains the most-significant bits of the output of Voice #3. If a ramp waveform is selected for Voice #3, for example, then the number in the OSC/RND register will increment linearly with time from 0 to 255, and then it will drop back to 0 and repeat this cycle again and again. If a noise waveform is selected for Voice #3, then a random number can be read in the OSC/RND register. Note that this is a read-only register.

Of what use is the number in this register? It may be used to *modulate* the numbers in the other registers. For example, to produce *vibrato*, you would use your program to add some fraction of the number in the OSC/RND register to the frequency of one of the voices. Since vibrato must take place at a slow rate, approximately 10 Hz, this eliminates Voice #3 as an audio source. Modulating the cut-off frequency also produces some nice sound effects, particularly if one of the inputs to the filter is a very narrow pulse waveform. Place a zero in the 3OFF bit in the MOD/VOL register for these applications. By scaling down the number in the OSC/RND register and adding it to the VOL nibble, you can produce *tremolo*.

The ENVLP 3 register contains the output of the ADSR envelope chosen for Voice #3. This number can be added to the frequency of a voice or to the frequency of the filter to produce unusual effects. Experiment with these possibilities.

Perhaps you think that we have left too much to experimentation. Regardless of whether you are using an analog synthesizer similar to the Moog synthesizer or a digital synthesizer such as the SID, experimentation is at the heart of electronic music synthesis. Begin your experiments with a thorough understanding of what the various parameters are supposed to do. Only then should you begin to experiment with those parameters. The advantage of an analog synthesizer is, perhaps, the ease with which experimentation can take place. The advantage of the digital synthesizer is the precision with which it may be controlled: its disadvantage is that it is not particularly easy to adjust in real time.

IV. Setting the Voice Frequencies

We are in the process of writing a program that will play a song. With that general objective in mind, we have already listed several subroutines that we will need. The SID image must be cleared. Then the SID itself is cleared by a subroutine to transfer the image in R/W memory to the write-only registers of the SID. To these two subroutines we have added a subroutine that initializes each of the three voices and the filter. These subroutines have been described in the previous sections of this chapter.

Notice that we did not attempt to write one long program to accomplish our objective. Instead, we wrote a number of subroutines that accomplish more specific objectives. These subroutine modules will be combined in a

subsequent program, and together they will meet the general objective. We believe that the programming technique of dividing a large task into many smaller tasks using subroutines as the basic building block is a good technique.

We turn next to the problem of setting the frequencies. When initializing the SID, we did not set the frequency registers of any of the voices. That is the task we will perform in this section. The SID is capable of playing notes over a range of eight octaves. Each octave has 12 frequencies. Each frequency is specified by a two-byte register in the SID (see Table 10-1). Thus, a table consisting of 192 bytes would store all of the frequency information needed to play any note the SID is capable of playing. A table that gives this information is provided in Appendix E, page 384, of the *Programmer's Reference Guide*. We will use a technique, also suggested in the *Programmer's Reference Guide*, that reduces the amount of frequency information that needs to be stored in memory.

Corresponding notes in adjacent octaves differ in frequency by a factor of two. For example, the A in the treble cleff has a frequency of 440 Hz. The A in the next higher octave has a frequency of 880 Hz, while the A in the octave below has a frequency of 220 Hz. To determine the frequency of any note, you need two items of information, the *note* and the *octave*.

Given the frequency of the note in the highest octave and the number of octaves *down* of the note you actually wish to play, you can find the frequency of this note by dividing the top octave frequency by *two* for each octave below the highest octave.

Table 10-3 identifies the frequency information we need for the notes in the top octave, called octave 7. The lowest octave is called octave 0. The second column in Table 10-3 gives the number to be loaded into the FREQ LO and FREQ HI registers to play the note specified in column one. The third and fourth columns divide the data in the third column into the numbers for FREQ HI and FREQ LO, expressed both in decimal and hexadecimal. The last column in Table 10-3 identifies the address of the location in memory where this information can be accessed by our program.

Table 10-3. Top octave frequency data.

Note	Frequency Number (Decimal)	High Byte (Decimal)	High Byte (Hex)	Low Byte (Decimal)	Low Byte (Hex)	Address
C 7	34334	134	\$86	30	\$1E	\$CF00-01
C# 7	36376	142	8E	24	18	\$CF02-03
D 7	38539	150	96	139	8B	\$CF04-05
D# 7	40830	159	9F	126	7E	\$CF06-07
E 7	43258	168	A8	250	FA	\$CF08-09
F 7	45830	179	B3	6	06	\$CF0A-0B
F# 7	48556	189	BD	172	AC	\$CF0C-0D
G 7	51443	200	C8	243	F3	\$CF0E-0F
G# 7	54502	212	D4	230	E6	\$CF10-11
A 7	57743	225	E1	143	8F	\$CF12-13
A# 7	61176	238	EE	248	F8	\$CF14-15
B 7	64814	253	FD	46	2E	\$CF16-17

Once again refer to Table 10-3. To play the A# note in octave seven, we need to load \$F8 from location \$CF14 and store it in the FREQ LO register of the SID, and we need to load \$EE from location \$CF15 and store it in the FREQ HI register of the SID. Observe that the LSB of the frequency number is stored *below* the MSB of the frequency. To play A# in octave four, we need to halve the frequency number in Table 10-3 three (7 - 4) times. Recall that a number may be divided by two with one application of an LSR or ROR instruction.

Figure 10-7 illustrates how we intend to store the codes required to specify the frequency. The least-significant nibble of a byte will be used to specify the note. Since there are 12 notes, C through B, we need 12 numbers; and we shall use the numbers 0 through 11 to correspond to the notes C through B, respectively. One nibble is sufficient since a nibble can contain a number as large as 15. The most-significant nibble of the same byte will be used to specify the octave of the note. It is *convenient* to have the number zero specify the *top octave* defined in Table 10-3, while the number *seven* will specify the *lowest octave*.

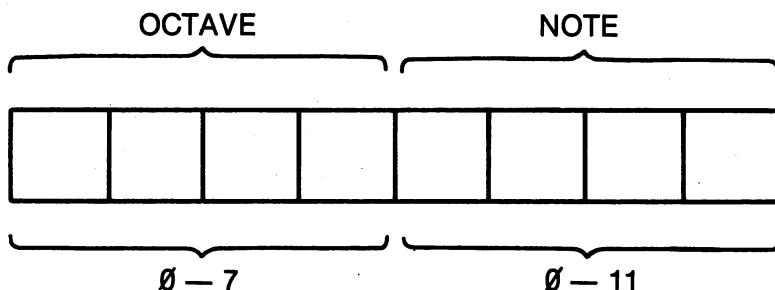


Figure 10-7. The octave/note byte for one voice.

Here is how the information in the octave/note byte illustrated in Figure 10-7 is used. The byte is first read and the top nibble is masked (AND #\$0F). After being multiplied by two, the number in the low nibble is transferred to the Y register to serve as an index to obtain the frequency information from the data stored in Table 10-3 with

LDA \$CF00,Y and LDA \$CF01,Y

instructions. Multiplication by two is necessary because there are twice as many bytes in the table as frequencies: each note requires two bytes to specify its frequency. The LDA \$CF00,Y instruction fetches the low byte of the frequency, while the LDA \$CF01,Y instruction fetches the high byte.

Once the frequency data is stored in the SID image registers, the octave information in the byte pictured in Figure 10-7 is recalled. It is first shifted to the low nibble. In the low nibble, this number correctly specifies the number of *halvings* required to calculate the frequency for that octave. Thus, the number of times the frequency in the SID image locations must be divided by two is specified by the number in the octave nibble. This number will serve

as the loop counter in a loop whose task is to divide the frequency number by two. Since the SID has three voices, we need three bytes of octave/note information like the one pictured in Figure 10-7 in order to play three notes in harmony.

Notice in Table 10-1 that the SID's frequency registers are located at \$00/\$01, \$07/\$08, and \$0E/\$0F in page \$D4. The corresponding SID image locations are \$80/\$81, \$87/\$88, and \$8E/\$8F in page \$CB of memory. It will be convenient to reference these frequency registers using

STA \$CB80,X and STA \$CB81,X

instructions where X takes on the values \$0, \$7, and \$E for Voices #1, #2, and #3, respectively.

We are now ready to introduce our subroutine that provides the frequency for a voice, given the proper voice number in the X register, and given the proper octave/note byte in the accumulator. An octave/note byte is pictured in Figure 10-7. The subroutine is listed in Example 10-7. The voice, octave, and note parameters are passed to the subroutine in the X and A registers. The frequency information is obtained from the table stored at \$CF00 (refer to Table 10-3). It is stored in the appropriate FREQ HI and FREQ LO register in the SID image. Next, the octave information is used to divide this frequency to place it in the proper octave, completing the subroutine. Observe that if the octave number is zero, no halving occurs. An octave number of one results in one division by two, and so forth.

Example 10-7. Frequency Control Subroutine

Object: Turn the codes in an octave/note byte into the correct frequency numbers in the SID image.

```

1      ;SUBROUTINE FREQUENCY
2 NOTE   EQU $CF00
3 LO     EQU $CB80
4 HI     EQU $CB81
5 FREQ   EQU $CA57
6      ;ENTER SUBROUTINE WITH NOTE (0-11) IN LOW
7      ;NIBBLE OF ACCUMULATOR.
8      ;OCTAVE (0-7) IN HIGH NIBBLE OF ACCUMULATOR.
9      ;X REGISTER HAS 0 FOR VOICE 1, 7 FOR VOICE 2,
10     ;AND E FOR VOICE 3.
11     ;
12     ;
13 FREQ   PHA          ;SAVE A FOR OCTAVE INFORMATION.           CA57 48
14     AND #$0F        ;ISOLATE NOTE INFORMATION.             CA58 29 OF
15     ASL A          ;THERE ARE TWICE AS MANY BYTES       CA5A 0A
16     TAY          ;IN THE TABLE AS NOTES.                 CA5B A8
17     LDA NOTE+1,Y  ;GET NOTE FROM THE TABLE.            CA5C B9 01 CF
18     STA HI,X      ;PUT IT IN THE FREQUENCY REGISTER.    CA5F 9D 81 CB
19     LDA NOTE,Y   ;GET LSB OF NOTE FROM TABLE.       CA62 B9 00 CF
20     STA LO,X      ;PUT IT IN THE FREQUENCY REGISTER.    CA65 9D 80 CB
21     PLA          ;GET THE OCTAVE INFORMATION BACK.    CA68 68
22     AND #$F0        ;DISREGARD NOTE INFORMATION.        CA69 29 F0
23     BEQ OUT        ;GET OUT FOR TOP OCTAVE.           CA6B F0 0F
24     LSR A          ;MOVE OCTAVE INFORMATION TO LOW.    CA6D 4A
25     LSR A          ;NIBBLE.                           CA6E 4A
26     LSR A          ;NIBBLE.                           CA6F 4A
27     LSR A          ;NIBBLE.                           CA70 4A

```

28	TAY	; PUT OCTAVE IN Y REGISTER.	CA71 A8
29	AGAIN CLC	; DO NOT ROTATE CARRY.	CA72 18
30	ROR HI,X	; DIVIDE BY 2 FOR EACH OCTAVE.	CA73 7E 81 CB
31	ROR LO,X		CA76 7E 80 CB
32	DEY		CA79 88
33	BNE AGAIN		CA7A D0 F6
34	OUT RTS	; FREQUENCY IS SET.	CA7C 60

One reason for using an image of the SID in R/W memory becomes clear in Example 10-7. The ROR HI,X and ROR LO,X instructions will not work with write-only registers because these instruction involve reading and writing to a register. In a moment, we will see a similar reason for adopting the SID image. When we gate a voice by setting bit zero in the control register, we would like to use an ORA instruction so we set bit zero without modifying the other bits. An ORA instruction requires a read operation, so it will not work with a write-only register.

V. Note Duration and Volume: Gating the SID

A note is played on the SID by performing all of the operations previously mentioned, after which bit zero (the GATE bit) in the control register of the voice is *set*. At the end of the note this bit must be *cleared*, beginning the decay phase of the ADSR envelope.

The subroutine in Example 10-8 sets the gate bits of all three voices, starting the note. It also calls the subroutine to transfer the SID image to the SID itself. The routines mentioned previously will have initialized the SID and set the frequency parameters. After setting the gate bits in the three control registers, the SID image is transferred to the SID, causing the SID to operate.

Example 10-8. Gate the SID

Object: Set the gate bits in each of the voice control registers, then transfer the image to the SID to begin playing.

10	GATEON LDA #\$01	; SET GATE BIT.	CAA0 A9 01
11	ORA CTR1	; VOICE 1.	CAA2 0D 84 CB
12	STA CTR1		CAA5 8D 84 CB
13	LDA #\$01	; SET GATE BIT.	CAA8 A9 01
14	ORA CTR2	; VOICE 2.	CAA9 0D 8B CB
15	STA CTR2		CAAD 8D 8B CB
16	LDA #\$01	; SET GATE BIT.	CAB0 A9 01
17	ORA CTR3	; VOICE 3.	CAB2 0D 92 CB
18	STA CTR3		CAB5 8D 92 CB
19	JSR TNSFER	; TRANSFER IMAGE TO SID.	CAB8 20 E0 CA
20	RTS		CABB 60

To stop the note, the gate bits in the control registers must be cleared, and the SID image must once again be transferred to the SID itself. The program in Example 10-9 accomplishes this task.

The note is played during the time interval after the gate is set and before it is cleared. The duration of this interval must be controllable, because it determines the length of the note: whole, half, quarter, eighth, sixteenth, or one

of the dotted varieties. It is clear that a delay subroutine is required. The delay routine that we used is given in Example 10-10. The part of the routine beginning at the instruction labeled LOOP delays approximately one millisecond for each count in the X register. The X register is loaded from R/W memory location \$0002. Thus, the number of millisecond delays is controlled by the number you store in location \$0002.

Example 10-9. Switch off the SID

Object: Clear the gate bits in each of the voice control registers, then transfer the SID image to the SID.

10	GATFF	LDA #\$FE	;CLEAR GATE BIT.	CAC0 A9 FE
11		AND CTR1	;VOICE 1.	CAC2 2D 84 CB
12		STA CTR1		CAC5 8D 84 CB
13		LDA #\$FE	;CLEAR GATE BIT.	CAC8 A9 FE
14		AND CTR2	;VOICE 2.	CACA 2D 8B CB
15		STA CTR2		CACD 8D 8B CB
16		LDA #\$FE	;CLEAR GATE BIT.	CAD0 A9 FE
17		AND CTR3	;VOICE 3.	CAD2 2D 92 CB
18		STA CTR3		CAD5 8D 92 CB
19		JSR TNSFER	;TRANSFER IMAGE TO SID.	CAD8 20 E0 CA
20		RTS		CADB 60

The number of times the delay loops starting at LOAF are executed is one larger than the number in the accumulator when the subroutine is called. The number of delays determines the length of the note. It will be 16 for a whole note, 12 for a dotted half note, 8 for a half note, 6 for a dotted quarter note, 4 for a quarter note, 3 for a dotted eighth note, 2 for an eighth note, and 1 for a sixteenth note.

Example 10-10. The Delay/Tempo Subroutine

Object: Loop for a time interval determined by the number in location \$0002 and the number in the accumulator.

10	DELAY	JSR LOAF	;WASTE SOME TIME.	CA7D 20 86 CA
11		SEC	;NUMBER OF DELAYS IS IN A.	CA80 38
12		SBC #01	;DECREMENT A.	CA81 E9 01
13		BPL DELAY		CA83 10 F8
14		RTS		CA85 60
15	LOAF	LDX +TEMPO	;NUMBER OF MILLISECOND INTERVALS	CA86 A6 02
16	LOOP	LDY #\$CB	;IN A SIXTEENTH NOTE.	CA88 A0 CB
17	REPET	DEY		CA8A 88
18		BNE REPET	;THIS LOOP TAKES 1 MILLISECOND.	CA8B D0 FD
19		DEX		CA8D CA
20		BNE LOOP	;REPEAT X TIMES FOR X MILLISECONDS.	CA8E D0 F8
21		RTS		CA90 60

We already require three bytes of octave/note information for each note in the song, one octave/note byte for each of the three voices. We will call the collection of this information our *song table*. Now we see that the song table must also include information about the length of the note. We need,

therefore, four bytes of information: three octave/note bytes and one length/volume byte that specifies the length and volume of the note. The length of the note is determined by a number between 0 and 15, which fits in one nibble. Recall that the SID volume is also a one-nibble quantity. Thus, the length and volume can be stored in the single byte diagrammed in Figure 10-8.

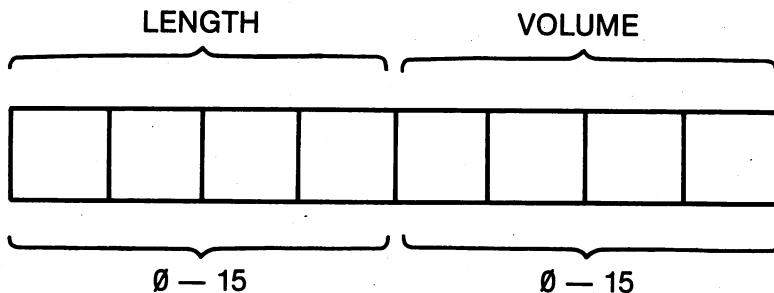


Figure 10-8. The length/volume byte for one voice.

We decided to store our song table in memory starting at location \$CC00. The order of the information must be understood. The first three bytes provide the octave/note information for Voices #1, #2, and #3, respectively. The next byte contains the length/volume information. Table 10-4 is a song table for *Auld Lang Syne*. The song program that uses all of the subroutines to play a song will be described shortly. It terminates when the volume nibble is zero. A program to load notes in the table will also be described shortly. A *rest* may be inserted in the song for any voice by choosing an octave number of eight. This will halve the frequency until the result is zero, producing no tone.

VI. The Program to Play a Song

Now we must combine the subroutines to play a song. A program to do this is given in Example 10-11, and what follows is a description of this program.

Line 18 in Example 10-11 prevents interrupts from the operating system while our song is playing. Lines 19 through 22 set up the starting address of the song table, \$CC00, for the *indirect indexed by Y* addressing mode. The (IND),Y addressing mode was chosen to read the information in the song table because it permits us to calculate the address of the location to be accessed. Lines 23 and 24 clear the SID image and the SID itself before starting to play the song.

The loop starting at line 27 and ending with line 34 reads the first three bytes with the octave/note information from the song table and converts them to frequencies in the SID image. On line 27, an octave/note byte is read. On line 28, the value of X that identifies the FREQ LO/HI registers is read from the following table, which must be stored in memory:

```
$CF18 $00
$CF19 $07
$CF1A $0E
```

Y is saved on line 29 because the subroutine that determines the frequency, called on line 30, modifies the Y register. Y is restored on line 31, and it is incremented on line 32 to fetch another octave/note byte if necessary.

All three octave/note bytes have been read and converted to the appropriate data in the SID image when Y reaches three. Thus, on line 35, the loop has ended and the length/volume byte is read from the song table. The PHA instruction on line 36 saves this data on the stack while the volume nibble is processed. If the volume nibble is zero, the song ends with the BEQ OUT instruction on line 38. Otherwise, the volume is stored in the MOD/VOL register in the SID image. This takes place on lines 39 and 40. The note begins on line 41 with a subroutine call to GATEON. This subroutine turns on the gates and transfers the SID image to the SID, starting the note.

Now the length information from the length/volume byte is processed into a delay on lines 42 through 47. After the delay the gates are turned off, and, except for the decay phase, the note ends with the subroutine call on line 48 to GATOFF. Now the address accessed with the LDA (IAL),Y instruction must be updated. Since each note requires four bytes, four must be added to the two-byte number in IAL and IAL + 1. This is accomplished with subroutine ADD in Example 10-12.

Example 10-11. A Program to Play a Song

Object: Play the song stored in the song table starting at \$CC00.

17	EQU \$CB00		CB00
18	SEI	; PREVENT INTERRUPTS.	CB00 78
19	LDA #00	; SET UP (IND),Y MODE POINTER.	CB01 A9 00
20	STA +IAL	; START AT PAGE BOUNDARY.	CB03 85 FB
21	LDA #\$CC	; SONG TABLE STARTS IN PAGE \$CC.	CB05 A9 CC
22	STA +IAL+1		CB07 85 FC
23	JSR CLEAR	; CLEAR SID REGISTERS.	CB09 20 00 CA
24	JSR TNSFER	; TRANSFER IMAGE TO SID REGISTERS.	CB0C 20 E0 CA
25	JSR VOICE1	; INITIALIZE VOICES.	CB0F 20 0B CA
26 UP	LDY #00	; Y INDEXES ONE SET OF NOTES.	CB12 A0 00
27 BR1	LDA (IAL),Y	; GET OCTAVE AND NOTE.	CB14 B1 FB
28	LDX VOICE,Y	; GET VOICE INDEX.	CB16 BE 18 CF
29	STY +\$FD	; SAVE Y FOR A MOMENT.	CB19 84 FD
30	JSR FREQ	; SET THE FREQUENCY.	CB1B 20 57 CA
31	LDY +\$FD	; RESTORE Y.	CB1E A4 FD
32	INY	; INCREMENT Y FOR ANOTHER VOICE.	CB20 C8
33	CPY #3	; ALL THREE VOICES FINISHED?	CB21 C0 03
34	BCC BR1	; NO, SO SET UP ANOTHER VOICE.	CB23 90 EF
35	LDA (IAL),Y	; GET DELAY AND VOLUME PARAMETERS.	CB25 B1 FB
36	PHA	; SAVE A FOR A MOMENT.	CB27 48
37	AND #\$0F	; ISOLATE VOLUME PARAMETER.	CB28 29 0F
38	BEQ OUT	; ZERO VOLUME SIGNALS END OF SONG.	CR2A F0 19
39	ORA \$CB98	; STORE IN MODE/VOLUME REGISTER	CB2C 0D 98 CB
40	STA \$CB98	; IN SID CHIP.	CB2F 8D 98 CB
41	JSR GATEON	; TURN ON THE GATES.	CB32 20 A0 CA
42	PLA	; GET DELAY/VOLUME BACK.	CB35 68
43	LSR A	; SHIFT DELAY INTO LOW NIBBLE.	CB36 4A
44	LSR A	; SHIFT VOLUME INTO BIT BUCKET	CB37 4A
45	LSR A	; IN THE SKY.	CB38 4A
46	LSR A		CB39 4A
47	JSR DELAY	; WAIT FOR DURATION OF NOTE.	CB3A 20 7D CA
48	JSR GATOFF	; TURN OFF THE GATES.	CB3D 20 C0 CA
49	JSR ADD	; UPDATE (IND),Y POINTER.	CB40 20 91 CA
50	BNE UP	; RETURN FOR MORE MUSIC.	CB43 D0 CD

51 OUT	PLA	:FIX UP THE STACK.	CB45 68
52	JSR CLEAR	:CLEAR THE SID CHIP.	CB46 20 00 CA
53	JSR TNSFER	:TRANSFER IMAGE TO SID.	CB49 20 E0 CA
54	CLI	:ALLOW INTERRUPTS.	CB4C 58
55	RTS		CB4D 60

Example 10-12. Subroutine Add

Object: Add four to the two-byte address stored in IAL and IAL + 1.

10 ADD	CLC	:CLEAR CARRY FOR ADDITION.	CA91 18
11	LDA #04	:INCREMENT (IND), Y POINTER	CA92 A9 04
12	ADC +IAL	:BY FOUR.	CA94 65 FB
13	STA +IAL		CA96 85 FB
14	LDA +IAL+1		CA98 A5 FC
15	ADC #0		CA9A 69 00
16	STA +IAL+1		CA9C 85 FC
17	RTS		CA9E 60

The BNE UP instruction on line 50 forces a branch back to get the next note. When the song ends, the SID is cleared again with the instructions on lines 52 and 53. Interrupts are enabled again on line 54 and the program returns to the calling program. The program in Example 10-11 can be executed from BASIC with a SYS 51968 command.

Try the program by loading it into memory. Be sure to load the three-byte table mentioned above as well as the table in Table 10-3 and all of the subroutines, including Example 10-12. Using the program in Example 2-8, store the codes listed in Table 10-4 for *Auld Lang Syne*. The starting address is \$CC00. Perform a POKE 2,128 command to set the tempo, then use the SYS 51968 command to start the program. Turn up the volume on your TV set or monitor, or, better yet, connect the audio output directly to your stereo. How does it sound?

Table 10-4. Song table for *Auld Lang Syne*.

40	40	50	4F	35	30	45	6F
35	30	47	2F	35	30	49	4E
39	35	45	4F	37	34	40	6F
35	32	40	2F	37	34	40	4F
39	34	40	4F	35	49	45	6F
35	49	45	2F	39	35	45	4F
20	35	45	4F	22	35	4A	CF
22	35	4A	4F	20	35	49	6F
39	35	45	2F	39	35	45	4F
35	49	45	4F	37	34	40	6F
35	32	40	2F	37	34	40	4F
39	34	40	4F	35	32	45	6F
32	49	42	2F	32	4A	5A	4F
30	4A	40	4F	35	49	55	CF
45	35	22	4F	45	35	20	6F
40	30	39	2F	45	35	39	4F
59	35	69	4F	40	34	37	6F

Table 10-4. Song table for *Auld Lang Syne* (continued).

40	32	35	2F	40	34	37	4F
40	34	22	4F	45	35	20	6F
40	30	39	2F	45	35	39	4F
49	30	59	4E	4A	35	32	CF
4A	35	25	4F	49	35	20	6F
45	35	39	2F	45	35	39	4F
59	30	35	4F	40	34	37	6F
40	32	35	2F	40	34	37	4F
41	34	39	2F	41	49	37	2F
42	32	35	6C	45	49	32	2C
5A	4A	32	4D	40	44	30	4F
55	45	35	FF	55	45	35	C0

19

VII. A BASIC Music Interpreter

It is a tedious task to translate music into the information required to store in the song table. As a small aid, we provide the simple note interpreter listed in Example 10-13. This program allows you to input the notes and octaves for all three voices in turn, and then you input the note length and volume, before getting the data for the next note in the song.

The note input is of the form

? G#,3 RETURN

where ? is the input prompt, and G# represents the note requested in octave three. There are 12 letter or letter and # sign note possibilities. You may need to be reminded that B flat is the same as A#, if the song has some flats in it. The lowest octave number is zero and the highest is seven. Most of the notes in the treble clef are in octave four. An octave number of eight will produce a rest for that voice.

The length/volume input is of the form

? 8,15 RETURN

where 8 is a half note, and 15 corresponds to the maximum volume. The input ends when you enter a volume of zero.

Example 10-13. A Simple BASIC Song Interpreter

```

5 REM MUSIC INTERPRETER
10 K=0:J=52224
30 FOR I=1 TO 3
40 PRINT "INPUT VOICE #";I;" NOTE,OCTAVE."
50 INPUT NS,OCT
55 IF OCT>8 THEN 85
60 OCT=(7-OCT)*16
70 IF NS="C" THEN N=0
71 IF NS="C#" THEN N=1

```

```

72 IF N$="D" THEN N=2
73 IF N$="D#" THEN N=3
74 IF N$="E" THEN N=4
75 IF N$="F" THEN N=5
76 IF N$="F#" THEN N=6
77 IF N$="G" THEN N=7
78 IF N$="G#" THEN N=8
79 IF N$="A" THEN N=9
80 IF N$="A#" THEN N=10
81 IF N$="B" THEN N=11
85 POKE J+K,OCT+N:PRINT OCT+N:K=K+1
90 NEXT
100 PRINT "INPUT NOTE LENGTH,VOLUME."
110 INPUT L,V
120 Z=(L-1)*16+V
130 POKE (J+K),Z:K=K+1
140 IF V>0 THEN 30

```

VIII. Summary

The 6581 SID is a three-voice, programmable sound synthesizer. The waveform, frequency, pulse width, and ADSR envelope parameters of each voice are controlled by the program. Triangle, ramp, square, and noise waveforms are available. These voices can be routed through a filter that can be programmed to be a low-pass, high-pass, or band-pass filter, with a programmable cut-off frequency.

The frequency of each voice covers an eight-octave range. A look-up table provides a convenient way to select the frequency corresponding to a particular note. If the frequencies of the 12 notes in the highest octave are stored in the table, then a divide-by-two routine may be used to generate the frequencies of the notes in lower octaves.

To play a song, each voice must be initialized, the filter must be programmed, and, as the song is played, the frequencies and durations of the notes are read from a song table. A useful technique for writing a program as lengthy as the one required to play a song is to divide the program into modules to be called as subroutines. This makes each module easy to understand, and it is possible to focus on one programming problem at a time.

Although the focus of this chapter was a single program to play a song, it is possible to generate many complex sounds for arcade-type video games with the 6581 SID.

IX. Exercises

We assume that you have successfully loaded and executed the machine-language routines in this chapter. The purpose of the exercises is to explore some of the other options offered by the SID.

1. The attack/decay, sustain/release parameters may be modified in our program by POKEing new numbers into the following memory locations: 51729, 51734, 51744, 51749, 51759, and 51764. These locations hold the operands of some of the LDA instructions in Example 10-3. POKE the number 12 into each one of

these locations. Describe what kind of ADSR envelope you expect with this parameter. Run the program and listen to the song. Adjust the tempo so the notes have time to release/decay to a low volume. Later you may wish to experiment with numbers other than 12; for now, proceed to the next exercise.

2. The numbers stored in the control registers may be modified by POKEing numbers into these three locations: 51724, 51739, and 51754. POKE 32 in each of these locations, then play the song. Now each voice has the same ramp waveform. Is the sound that results more like a guitar or a piano?
3. Repeat Exercise 2, with each voice playing a triangular waveform. What number must you POKE into the locations given in Exercise 2 to do this? Characterize the sound after playing the song.
4. Repeat Exercise 2 after POKEing 20 into each of the locations specified in Exercise 2. This activates the ring modulator, and the sounds you perceive when you play the song will have some bell-like characteristics.
5. Repeat Exercise 2, but POKE 18 into the locations mentioned there. Now you will set the SYNC bit in each control register. Play the song.
6. Some of the most interesting effects are produced by generating noise instead of one of the other waveforms. Repeat Exercise 2, but POKE 128 into the locations mentioned. Play the song. Do you think you can synthesize a gun shot?
7. Play Voice #3 by itself and then modify the pulse width to observe the effect of pulse width on the sound. A voice may be quieted by POKEing zero into the location given in Exercise 2. The pulse width may be modified by POKEing various numbers into location 51774 and then playing the song.
8. Experiment with the filter settings by modifying the program. Try band-pass, low-pass, and high-pass filters with various cut-off frequencies.

Applications Using the 6567 Video Interface Chip

I. Introduction

The purpose of this chapter is to describe some of the many video display options that can be programmed with the use of the 6567 VIC (video interface chip), usually called the VIC II. The VIC is responsible for all of the video display operations on the Commodore 64. In certain modes, such as the bit-mapped mode, the video display uses large amounts of memory, so we will discuss some memory-management techniques when the need arises.

The VIC chip has 46 registers, which begins to suggest its complexity. Movable object blocks, otherwise known as sprites, bit-mapped color graphics, multiple character sets, and screen switching are just a few of the video display capabilities that can be programmed. Space does not permit a complete description of all its capabilities; therefore, we will concentrate on a few applications and attempt to make these discussions as useful and complete as possible. Consult the *Programmer's Reference Guide* for a more elaborate description of the 6567 VIC.

II. 6567 VIC Programming Fundamentals

A. Bank switching the VIC

The main purpose of the VIC is to *map* codes from R/W memory into video information that appears on the video monitor or TV. The VIC accesses some of the same memory as the 6510 microprocessor in order to display information that the processor stores. One of the four 16K blocks of memory in the 64K memory space of the 6510 microprocessor is used, in part, for video display purposes. The 16K *bank* of memory that the VIC uses to get its information is programmable. Bits zero and one of Port A in the 6526 CIA #2 control

which 16K bank of memory the VIC uses for video information. The address of this port is \$DD00, and its data direction register (DDR) is located at \$DD02. (I/O ports and DDRs will be discussed in more detail in the next chapter.) Refer to Table 11-1 to find the information necessary to select the 16K memory bank that the VIC uses. When power is first applied, the microcomputer system is configured so that the VIC looks at the lowest 16K bank of memory for its information.

Table 11-1. Bank control codes for the VIC.

16K Memory Bank	Port A Data		Base Address			ORA #
	Bit 1	Bit 0	Bank #	(Decimal)	(Hex)	
\$0000-\$3FFF*	1	1	0	0	\$0000	\$03
\$4000-\$7FFF	1	0	1	16384	\$4000	\$02
\$8000-\$BFFF	0	1	2	32768	\$8000	\$01
\$C000-\$FFFF	0	0	3	49152	\$C000	\$00

*Normal or default value.

The following program segment will select the bank of memory that you want the VIC to use. The operand of the second ORA instruction is zero, one, two, or three, depending on whether bank number three, two, one, or zero, respectively, is selected. Refer to the last column of Table 11-1 to find the correct operand of the ORA instruction.

```

LDA #03      ;Configure the DDR so bits zero
ORA $DD02    ;and one are output bits.
STA $DD02
LDA #$FC     ;Mask bits zero and one.
AND $DD00
ORA #$01     ;Select bank #2, $8000 - $BFFF.
STA $DD00

```

B. Screen memory

How is text displayed on the video monitor? First note that the *text screen* consists of 25 rows and 40 columns, giving a total of 1,000 screen locations. There is a one-to-one correspondence between these 1,000 locations on the screen and 1,000 locations in *R/W memory*. This block of memory is called *screen memory*.

The first 40 locations in screen memory are mapped by the VIC into characters in the top row on the screen. The second 40 locations in screen memory correspond to the second row on the screen, and so on, until all 1,000 memory locations are mapped onto the screen.

Each screen memory location holds a *screen character code*, also called a screen code. Writing a certain screen character code to a specific memory location in screen memory will cause the corresponding character to appear on the screen. The screen codes are *not* the same as ASCII. The screen character codes are listed in Appendix E of the *Commodore 64 User's Guide* that came with your computer.

Our principal question is: where is screen memory? Table 11-2 describes the location of screen memory *relative* to the lowest address, the *base address*, of the 16K memory bank being used by the VIC (Table 11-1). The location of screen memory is determined by the number in the most-significant nibble of the VIC memory pointer register at \$D018. Remember that you must *add* the address in Table 11-2 to the base address in Table 11-1 to find the actual address of screen memory.

Table 11-2. Location of screen memory.

<i>Starting Address (Hex)</i>	<i>Most-Significant Nibble (Location \$D018)</i>	<i>ORA #</i>
\$0000	0	\$00
\$0400	1024*	1
\$0800	2048	2
\$0C00	3072	3
\$1000	4096	4
\$1400	5120	5
\$1800	6144	6
\$1C00	7168	7
\$2000	8192	8
\$2400	9216	9
\$2800	10240	A
\$2C00	11264	B
\$3000	12288	C
\$3400	13312	D
\$3800	14336	E
\$3C00	15360	F

*Default value.

Here is an assembly-language program segment that selects the location you wish to use for screen memory without affecting the other bits in the VIC control register at \$D018:

```
LDA # $0F      ;Mask the most-significant nibble.
AND $D018
ORA # $10      ;Choose screen memory number $10,
STA $D018     ;putting screen memory at $0400.
```

In this program segment, we have chosen the address \$0400 to add to the base address of the bank to identify the screen memory address. The operand of the ORA instruction is found in the last column of Table 11-2. Recall that in the previous program segment, we selected \$8000 as the base address of the bank, so screen memory is at \$8000 + \$0400 = \$8400.

It should be obvious that you can set up 16 different screens of text information and switch from one to the other very quickly. Of course, on the Commodore 64, text information is not confined to letters and numbers. Any of the graphics characters on the keyboard can be stored in screen memory. It

may be worth pointing out that the Commodore 64 operating system assumes that screen memory starts at \$0400, and if you switch it to another part of memory you will get garbage on the screen.

You will see that screen memory has a *different* function when performing bit-mapped graphics. When the bit-mapped mode is used, each location in screen memory corresponds to a location on the screen, but the screen memory location does not contain a code for the character. Instead, the two nibbles of a screen memory location contain two color codes, one for "on" picture elements and the other for "off" picture elements. More about this later.

A program to clear screen memory was given in Example 8-4.

C. Color memory

Corresponding to the bank of memory known as screen memory is a bank of 1,000 memory locations known as *color memory*. The location of color memory is fixed; it always occupies locations \$D800 through \$DBE7. The color of the character is determined by its corresponding color code. Thus, if screen memory starts at \$0400, the *color code* of the character whose screen code is stored at \$0400 is stored in color memory at \$D800. The 16 possible colors and their corresponding codes are given in Table 11-3.

Table 11-3. Color codes.

<i>Color</i>	<i>Color Code</i>
Black	\$0
White	1
Red	2
Cyan	3
Purple	4
Green	5
Blue	6
Yellow	7
Orange	8
Brown	9
Light red	A
Dark gray	B
Medium gray	C
Light green	D
Light blue	E
Light gray	F

A program to store color codes in color memory was given in Example 8-12. Of course, if the characters are to be visible, they must differ in color from the background.

- The background color is determined by storing one of the 16 color codes in Table 11-3 in the VIC's *background color register* at \$D021.
- The border, or exterior color, is determined by storing one of the color codes in the VIC's *exterior color register* at \$D020.

D. Character memory

Each of the 1,000 locations on the screen consists of a rectangular *dot matrix* consisting of 64 dots, 8 dots wide and 8 dots high. Each *dot* is represented by a *bit* in a certain memory location.

- If the bit is one, then the dot on the screen will have the same color as the color specified for that character in color memory.
- If the bit is zero, then the dot on the screen will have the same color as the color specified in the background color register.

A dot is actually called a picture element, or *pixel* for short. Each character is represented in memory by 64 bits, called the *character pattern*. One character can appear in each of the 1,000 screen locations.

Some simple arithmetic shows that eight bytes of memory are needed to specify the on/off information for the 64 pixels required to display one character. Thus, eight bytes of memory are needed to store one character pattern. Each screen code is an eight-bit number, thus there are 256 possible screen codes representing 256 possible character patterns. Since each character pattern requires eight bytes of memory, we need $8 \times 256 = 2048$ memory locations to store 256 unique character patterns. The 2K bank of memory locations required to store 256 character patterns is called *character memory*.

The location of character memory within the 16K bank of memory that the VIC is using is controlled by bits one, two, and three in the VIC register located at \$D018 (refer to Table 11-4). A program segment to accomplish the selection of the character memory bank follows:

```
LDA #$F1      ;Mask bits one, two, and three.  
AND $D018  
ORA #$04      ;$1000 will start character memory.  
STA $D018
```

The operand of the ORA instruction is found in Table 11-4. Since an earlier program segment selected \$8000 as the base address of the 16K bank used by the VIC, both program segments make \$9000 as the base address of character memory. The character patterns must start at location \$9000, and extend upward in memory in groups of 64 bytes for each character pattern.

Here is how the VIC places a character on the screen. It looks in a screen memory location for the screen code. It also looks in color memory to find the color of the character. The screen code is multiplied by eight and added to the base address of character memory. The character pattern is fetched from character memory, and the pattern is displayed on the screen. The eight comes from the fact that eight bytes are devoted to each character code. Of course, the base address of character memory is determined by bits one, two, and

three in \$D018 (see Table 11-4) and the base address of the bank (see Table 11-1) being used by the VIC. This all takes place inside the VIC: the microprocessor has nothing to do with it.

Table 11-4. Location of the 2K character memory bank.

<i>Starting Address (Hex)</i>	<i>(Decimal)</i>	<i>Register \$D018 Bit Values</i>			<i>ORA #</i>
		<i>Bit 3</i>	<i>Bit 2</i>	<i>Bit 1</i>	
\$0000	0	0	0	0	\$00
\$0800	2048	0	0	1	\$02
\$1000	4096*	0	1	0	\$04
\$1800	6144	0	1	1	\$06
\$2000	8192	1	0	0	\$08
\$2800	10240	1	0	1	\$0A
\$3000	12288	1	1	0	\$0C
\$3800	14336	1	1	1	\$0E

*Default value.

E. Character ROM

If the VIC fetches character patterns from R/W memory, then the character patterns are lost each time the power is removed. Since the operating system requires characters to display on the screen when power is first supplied to the Commodore 64, it is desirable to have character patterns stored in ROM rather than R/W memory. With some hardware tricks, the Commodore 64 system does just that. When either bank zero or bank two is selected for the VIC and character memory is selected to be at \$1000, the VIC actually looks at ROM locations \$D000 through \$D7FF. Likewise, if character memory is selected to be at \$1800, the VIC actually looks at ROM locations \$D800 through \$DFFF. The first character set is the uppercase/graphics set, the second is the upper/lowercase set.

Once again, this works only if banks zero or two are selected for the VIC to use. It is all accomplished in hardware; you do not have to worry about it. Moreover, in this case you can have your own information or programs stored in character memory. It will not be disturbed and it will not affect what you see on the screen. If you want to generate your own character patterns, a procedure thoroughly described in the *Programmer's Reference Guide*, you must choose either banks one or three for the VIC, or you must choose a starting address for character memory that is different from the ones just described.

F. A demonstration program

It is time to provide a programming example that will demonstrate the concepts just described. The instructions in our program will be sequenced, as closely as possible, with the topics we have discussed so far. The program is listed in Example 11-1. It displays most of the characters in the Commodore

character ROM, using all of the possible colors. Its purpose is simply to illustrate the assembly-language programming involved in accomplishing the tasks described in the previous paragraphs. Each program segment is delineated by spaces and a comment that describes its function. Study the program as you review the topics we have just presented.

Example 11-1. A Program to Demonstrate Some of the Video Display Options of the VIC

Object: Use locations \$8000 through \$BFFF for the VIC. Start screen memory at \$8400 and character memory at \$9000. Fill the screen with a variety of ROM characters in a variety of colors.

```

10      ;VIC BANK SELECT ROUTINE.
11          LDA #03      ;MAKE BITS 0 AND 1 OF PORT A      C000 A9 03
12          ORA $D002    ;OUTPUT BITS.                  C002 0D 02 DD
13          STA $D002    ;                         C005 8D 02 D0
14          LDA #$FC      ;MASK BITS 0 AND 1.      C008 A9 FC
15          AND $DD00    ;                         C00A 2D 00 DD
16          ORA #$01      ;SELECT BANK #2, $8000 - $BFFF.   C00D 09 01
17          STA $D000    ;STORE IN PORT A OF CIA #2.   C00F 8D 00 DD
18      ;
19      ;SELECT LOCATIONS FOR SCREEN MEMORY.
20          LDA #$0F      ;MASK MOST SIGNIFICANT NIBBLE.   C012 A9 0F
21          AND $D018    ;$D018 IS REGISTER $18 IN THE VIC.   C014 2D 18 D0
22          ORA #$10      ;PUT SCREEN MEMORY AT $8400.     C017 09 10
23          STA $D018    ;                         C019 8D 18 D0
24      ;
25      ;FILL SCREEN MEMORY WITH SCREEN CODES.
26          LDX #250     ;SET UP SCREEN MEMORY TO      C01C A2 FA
27 LOOP1    DEX        ;DISPLAY THE CHARACTER SET.   C01E CA
28          TXA       ;                         C01F 8A
29          STA SCN,X    ;PUT CODES IN ALL 1000 LOCATIONS.   C020 9D 00 84
30          STA SCN+250,X   ;                         C023 9D FA 84
31          STA SCN+500,X   ;                         C026 9D F4 85
32          STA SCN+750,X   ;                         C029 9D EE 86
33          BNE LOOP1    ;                         C02C D0 F0
34      ;
35      ;FILL COLOR MEMORY WITH COLOR CODES.
36          LDX #250     ;SET UP COLOR MEMORY FOR A      C02E A2 FA
37 LOOP2    TXA       ;                         C030 8A
38          AND #$0F      ;FOR A VARIETY OF COLORS.     C031 29 0F
39          DEX       ;                         C033 CA
40          STA CLR,X    ;STORE COLOR CODES IN      C034 9D 00 D8
41          STA CLR+250,X   ;1000 COLOR MEMORY LOCATIONS.   C037 9D FA D8
42          STA CLR+500,X   ;                         C03A 9D F4 D9
43          STA CLR+750,X   ;                         C03D 9D EE DA
44          BNE LOOP2    ;                         C040 D0 EE
45      ;
46      ;CHOOSE BACKGROUND AND EXTERIOR COLORS.
47          LDA #$0F      ;CHOOSE LT. GRAY FOR THE      C042 A9 0F
48          STA $D021    ;BACKGROUND COLOR.      C044 8D 21 D0
49          LDA #$02      ;CHOOSE RED FOR THE      C047 A9 02
50          STA $D020    ;BORDER COLOR.      C049 8D 20 D0
51      ;
52      ;SELECT LOCATION FOR CHARACTER MEMORY.
53          LDA #$F1      ;MASK BITS 1, 2, AND 3.      C04C A9 F1
54          AND $D018    ;PUT CHARACTER MEMORY AT $9000.   C04E 2D 18 D0
55          ORA #$04      ;                         C051 09 04
56          STA $D018    ;                         C053 8D 18 D0
57      ;
58 HERE    JMP HERE    ;LOOP HERE FOREVER.      C056 4C 56 C0

```

III. Bit-Mapped Graphics

The focus of this section will be a set of subroutines that can be called either from BASIC or machine-language programs and that give the Commodore 64 the capability of doing bit-mapped graphics.

A. Fundamentals of the bit-mapped graphics mode

Now that we have learned how to program the 6567 VIC to display characters, we move to another VIC display option, bit-mapped graphics. In the *character display mode*, each of the 1,000 eight-by-eight dot matrices displays one character on the screen. In the *bit-mapped mode*, we are not necessarily interested in making characters that are confined to an eight-by-eight dot matrix: we want to control individual dots to create any desired shape of any desired size.

Recall that each rectangular dot matrix has 64 pixels that are mapped from 64 bits occupying eight bytes of memory. Thus, eight bytes are mapped into one dot matrix. Since there are 1,000 eight-by-eight dot matrices on the screen, there are 64,000 picture elements, and 64,000 bits of memory will be required to store the information to be displayed on the screen. Since there are eight bits in a byte, 8,000 bytes of memory are required to store the screen information. (8,000 bytes is approximately 8K bytes of memory.)

The 8K block of memory that stores the information mapped onto the screen will be called *bit-mapped memory*. Bit-mapped memory starts at the same location as character memory, defined in the previous section (see Table 11-4). It is important to start character memory so that it provides 8K of memory in the 16K bank used by the VIC. Referring to Table 11-4, this means that the starting address can be no larger than \$2000.

Since there are 40 columns and 25 rows of rectangular dot matrices on the screen, the bit-mapped memory screen consists of a rectangular array of $40 \times 8 = 320$ dots by $25 \times 8 = 200$ dots. The description "320-by-200 pixels" defines the *resolution* that is available on the Commodore 64 in its bit-mapped graphics mode.

B. Preparing for bit-mapped graphics

The VIC goes into the bit-mapped mode when bit five of register \$11 is set. The address of this register is \$D011. One step in our initialization sequence of instructions will require us to set this bit. The next step is to decide which 16K bank to use for the VIC. Let us choose the bank from \$8000 to \$BFFF, since this will introduce some interesting features of the Commodore 64 system.

Now we must decide where to put the bit-mapped memory within this bank. Let us use the top 8K block of memory in this bank for bit-mapped

memory. Refer to Table 11-4 and observe that we must choose \$2000 as the starting address to use the top 8K of the 16K bank \$8000 to \$BFFF. Thus, bit-mapped memory will extend from \$A000 to \$BFFF.

This block of memory, \$A000 to \$BFFF, is also occupied by the BASIC interpreter in ROM. In a moment we will describe the memory-management techniques required to switch out the BASIC ROM and switch in the R/W memory "underneath" the ROM. For now, it will be sufficient to realize that it is possible to *write* information to the R/W memory underneath the BASIC ROM without switching out the ROM.

We are now ready to produce two of the subroutines that we will use to produce bit-mapped graphics. The first subroutine is given in Example 11-2. This subroutine selects the memory bank to be used by the VIC. It places the bit-mapped memory at \$8000 + \$2000 = \$A000, and it sets the bit-mapped mode, bit five in register \$D011.

Example 11-2. Bit-Mapped Graphics Initialization Subroutine

Object: Make the VIC use the memory space from \$8000 to \$BFFF. Put screen memory at \$0400 and character memory (bit-mapped memory) at \$2000. Set the BM_M (bit-mapped mode) bit in VIC register \$11.

10	MEMSET	LDA #\$03	;MAKE BITS ZERO AND ONE	C800 A9 03
11		ORA \$DD02	;OUTPUT BITS.	C802 0D 02 DD
12		STA \$DD02		C805 8D 02 DD
13		LDA #\$FC	;MASK BITS ZERO AND ONE.	C808 A9 FC
14		AND \$DD00	;SELECT VIC MEMORY BANK FROM	C80A 2D 00 DD
15		ORA #\$01	\$8000 TO \$BFFF.	C80D 09 01
16		STA \$DD00	;RESULT INTO PORT A OF CIA #2.	C80F 8D 00 DD
17		LDA #\$18	;PUT SCREEN MEMORY AT \$0400,	C812 A9 18
18		STA \$D018	;BIT MAP AT \$2000 + \$8000.	C814 8D 18 D0
19		LDA #\$20	;SET BIT FIVE IN REGISTER \$11	C817 A9 20
20		ORA \$D011	;TO GET BIT MAPPED MODE.	C819 0D 11 D0
21		STA \$D011		C81C 8D 11 D0
22		RTS	;RETURN FROM MEMORY INITIALIZATION.	C81F 60

The next routine clears the bit-mapped memory locations in order to start with a blank screen. Since we decided to use the 16K bank of memory from \$8000 to \$BFFF for the VIC, and since we set character memory at \$2000, the bit-mapped memory extends from \$A000 to \$BFFF. The subroutine in Example 11-3 uses the (IND), Y addressing mode to clear this 8K block of memory.

In the bit-mapped mode, the colors of the pixels come from screen memory. The color of the pixels in each of the 1,000 rectangular dot matrices on the screen is determined by the codes in the 1,000 screen memory locations as follows:

- If the bit corresponding to a pixel is one, then the color of the pixel is determined by the color code in the most-significant nibble.
- If the bit corresponding to a pixel is zero, then the color of the pixel is determined by the color code in the least-significant nibble.

Thus, each byte of screen memory is used to control the colors in a 64-dot matrix on the screen.

Example 11-3. Clear Screen Subroutine

Object: Clear the bit-mapped memory locations \$A000 to \$BFFF.

10	CLEAR	LDA #\$AO	;SET UP (IND), Y MODE POINTERS	C820 A9 A0
11		STA +\$FB+1	;TO POINT TO THE BASE OF THE	C822 85 FC
12		LDX #\$BF	;BIT MAPPED MEMORY BANK.	C824 A2 BF
13		LDA #00	;ALL LOCATIONS WILL BE CLEARED.	C826 A9 00
14		STA +\$FB		C828 85 FB
15		TAY		C82A A8
16	LOOP	STA (\$FB),Y	;ADL=(\$FB)+Y. ADH=(\$FB+1).	C82B 91 FB
17		INY	;NEXT Y.	C82D C8
18		BNE LOOP	;IF Y<>0 THEN GO TO LOOP.	C82E D0 FB
19		INC +\$FB+1	;INCREMENT PAGE NUMBER.	C830 E6 FC
20		CPX +\$FB+1	;IS X<(\$FB+1)? YES, C=0. NO, C=1.	C832 E4 FC
21		BCS LOOP	;NO, CLEAR ANOTHER PAGE.	C834 B0 F5
22		RTS		C836 60

To keep things simple, our subroutine uses only one color for the background, corresponding to the bits that are zero, and one other color for the bits that are one. These colors are determined by the two nibble-length codes stored in location \$0002. The color codes are described in Table 11-3. The subroutine is given in Example 11-4.

Example 11-4. Subroutine to Determine the Colors

Object: Get the two color nibbles from location \$0002 and store them in screen memory for the bit-mapped mode.

10	SETCLR	LDX #250		C840 A2 FA
11		LDA +\$02	;GET COLOR CODES FROM \$0002.	C842 A5 02
12	MORE	DEX		C844 CA
13		STA SCN,X	;STORE COLORS IN SCREEN MEMORY.	C845 9D 00 84
14		STA SCN+250,X		C848 9D FA 84
15		STA SCN+500,X		C84B 9D F4 85
16		STA SCN+750,X		C84E 9D EE 86
17		BNE MORE		C851 D0 F1
18		RTS		C853 60

To test these three routines, we suggest that you execute them one at a time from BASIC. After the three subroutines are loaded, use these commands:

```
SYS 51200
SYS 51232
SYS 51264
```

Notice the effect on the screen after each command is executed. The last two commands must be executed "in the dark," as you will see.

C. Calculating the location of a pixel's bit

Each of the 64,000 bits in the bit-mapped memory turns on (or off) one of the 64,000 pixels. The most convenient way of identifying a pixel is with a pair of coordinates, x and y. (We will use lowercase letters for the x- and

y-coordinates to distinguish them from the X and Y registers that will also appear in this discussion.) Let (x, y) be the position of a pixel. Since the screen has 320 pixels across and 200 pixels down, x and y are bounded as follows:

$$\begin{aligned} 0 &\leq x \leq 319 \\ 0 &\leq y \leq 199 \end{aligned}$$

The pixel in the upper left-hand corner of the screen is identified with the coordinate pair $(0, 0)$. The coordinate x increases to the right, while the coordinate y increases downward (refer to Figure 11-1).

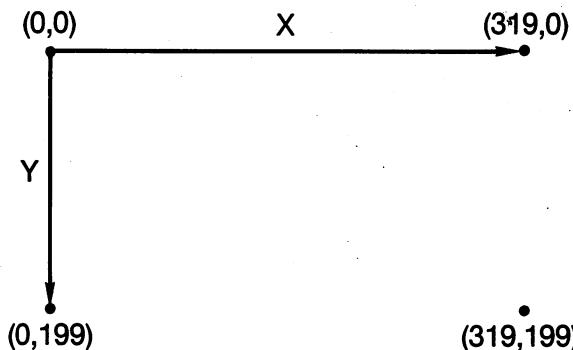


Figure 11-1. Diagram of the bit-mapped screen.

Here are two questions we must answer in order to do bit-mapped graphics. Given the pixel identified by (x, y) , what is the address of the memory location of the bit that turns on this pixel? Which of the eight bits in that memory location is the one that corresponds to the pixel?

To answer these questions, we must know how bit-mapped memory maps into pixels. This is described in Table 11-5. The first byte in bit-mapped memory, identified by "BYTE 0" in Table 11-5, controls the eight pixels in the upper left-hand corner of the screen. The actual address of the byte is found by adding the byte number to the base address of bit-mapped memory, namely, \$A000. Bit seven of BYTE 0 controls the pixel at $(0, 0)$, bit six controls the pixel at $(1, 0)$, and so on, until we find that bit zero controls the pixel at $(7, 0)$. BYTE 8 controls the pixels whose y-coordinates are zero and whose x-coordinates fall between 8 and 15. Thus, any byte in bit-mapped memory maps into a set of eight *horizontal* pixels. If the coordinate of the left-most pixel in this set is x, then refer to Figure 11-2 and observe that the coordinates of the other seven pixels are

$$x + 1, x + 2, \dots, x + 7$$

Notice that once a byte is identified, a number from zero to seven identifies a bit number. This is diagrammed in Figure 11-2. Thus, if x were represented by a binary number, the three least-significant bits of x, which represent a number from zero to seven, can be used to identify the bit number corresponding to a pixel. The formula for the bit number n is

$$n = 7 - (x \text{ AND } 7)$$

where x is the coordinate of the pixel. The AND in the formula is the logical AND described in Chapter 5.

Table 11-5. Memory map for the bit-mapped screen.

x	0-7	8-15	16-23	.	.	.	312-319
	COLUMN #0	COLUMN #1	COLUMN #2	COLUMN #39			
y							
0	BYTE 0	BYTE 8	BYTE 16	.	.	.	BYTE 312 B L
1	BYTE 1	BYTE 9	BYTE 17	.	.	.	BYTE 313 O C
.	K
.	#
.	0
7	BYTE 7	BYTE 15	BYTE 23	.	.	.	BYTE 319 # 0
8	BYTE 320	BYTE 328	BYTE 336	.	.	.	BYTE 632 B L
9	BYTE 321	BYTE 329	BYTE 337	.	.	.	BYTE 633 O C
.	K
.	#
.	1
15	BYTE 327	BYTE 335	BYTE 343	.	.	.	BYTE 639 # 1
.
.
.
.
.
192	BYTE 7680	BYTE 7688	BYTE 7696	.	.	.	BYTE 7992 B L
193	BYTE 7681	BYTE 7689	BYTE 7697	.	.	.	BYTE 7993 O C
.	K
.	#
199	BYTE 7687	BYTE 7695	BYTE 7703	.	.	.	BYTE 7999 2 4

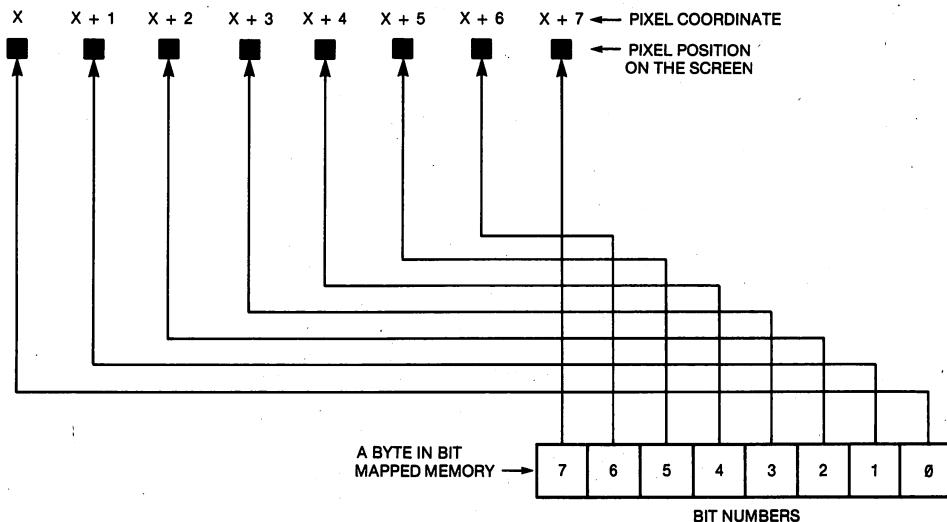


Figure 11-2. The relationship of bit numbers in a byte of bit-mapped memory to the coordinate of a pixel on the screen.

Now that we know how to identify the bit number within a byte by its x -coordinate, let us return to the problem of identifying the byte number. Refer again to Table 11-5. The 320 pixels across the top of the screen are mapped from bytes

0, 8, 16, ..., 312

The 320 pixels in the next row, corresponding to $y = 1$, are mapped from bytes
1, 9, 17, ..., 313

The first eight rows of pixels come from the block of memory with byte numbers 0 to 319. The next eight rows of pixels come from the next 320 bytes, numbered 320 to 639, and so on, until the bottom eight rows of pixels are mapped from the last 320 bytes in bit-mapped memory, with byte numbers 7,680 to 7,999.

Thus, bit-mapped memory can be divided into 25 blocks of memory, each containing 320 bytes and each controlling eight horizontal rows of pixels. This is illustrated in Table 11-5. Every eighth y value starts a new block of 320 bytes. Thus, the byte number of the first byte in any block is given by

$$320 * (\text{INT}(y/8))$$

where y is the y -coordinate of the pixel. The INT function simply means that any remainder from the division is disregarded.

Suppose that x is between zero and seven, in other words, the byte falls in Column #0 in Table 11-5. Then the byte number *within* a block is determined by adding the number represented by the three least-significant bits of y to the byte number of the first byte in the block.

Thus, the pixel at $(0,7)$ is in the byte whose byte number is $0 + 7$. Consider the pixel at $(5,9)$. The $\text{INT}(9/8) = 1$, so the byte that controls this pixel is in

Block #1, which begins at 320. Dropping all but the three least-significant bits of nine leaves a one. Thus, the byte number of this pixel is $320 + 1$, or 321. Refer to Table 11-5 to make sure this is correct. In other words, to determine the byte number for the columns of pixels from $x = 0$ to $x = 7$, we must add

$y \text{ AND } 7$

to the byte number of the beginning of the block.

What if x is a number from 8 to 15? Every *eighth* x begins a new column of bytes. The next column of bytes in Table 11-5 has byte numbers that are eight more than the byte numbers in the first column of bytes, provided the vertical position, y , is unchanged. Thus, for these values of x , we must not only add ($y \text{ AND } 7$) but we must also add eight. If x is a number from 16 to 23, we must add 16 to the byte number found in the first column. If x is a number from 24 to 31, we must add 24 to the byte number found in the first column, and so on. Thus, the x -coordinate is also involved in finding the byte number. In fact, from the pattern

8, 16, 24, ...

that develops, you can see that we must add the quantity

$8 * (\text{INT}(x/8))$

to the quantity ($y \text{ AND } 7$) and the byte number, $320 * (\text{INT}(y/8))$, of the beginning of the block to get the byte number for *any* x and y . Thus, the byte number is given by the expression

$320 * (\text{INT}(y/8)) + (y \text{ AND } 7) + 8 * (\text{INT}(x/8))$

The address, ADH:ADL, of this byte is found by adding the base address of bit-mapped memory. Thus, ADH:ADL for any byte in the bit-mapped memory is given in terms of the x - and y -coordinates by the expression

$\text{ADH:ADL} = \$A000 + \$140 * (\text{INT}(y/8)) + (y \text{ AND } 7) + \$8 * (\text{INT}(x/8))$

where we have made use of the fact that $320 = \$140$.

This finishes our calculation of the address of any byte in the bit-mapped memory corresponding to any point (x,y) on the screen. It is unfortunate that this is such a complex procedure, but that seems to be a characteristic of bit-mapped graphics. We will now relate our calculation to the assembly-language instructions required to address the appropriate byte in the bit mapped memory, given any x and y .

Recall from our discussion of the (IND), Y addressing mode in Chapter 8 that

$\text{ADH:ADL} = \text{BAH:BAL} + Y$

where BAH and BAL are eight-bit numbers stored in zero page locations symbolized by IAL and IAL + 1, and IAL is the operand of the instruction using this addressing mode. In other words, to identify the address of the operand in the bit-mapped memory, we must have numbers for BAH, BAL, and Y corresponding to any point (x,y) . That is, we must set

$\text{BAH:BAL} + Y = \$A000 + \$140 * \text{INT}(y/8) + (y \text{ AND } 7) + 8 * \text{INT}(x/8)$

Since y is between zero and 199, it can be represented by an eight-bit number. It will take two bytes to represent x , since x can be as large as 319, and the largest eight-bit number is 255. Let XLO symbolize the location that contains the eight least-significant bits of x , and let XHI symbolize the location that contains the most-significant bits of x .

Observe that $(y \text{ AND } 7)$ produces a three-bit number. Observe also that $8 * \text{INT}(x/8)$ is equivalent to dividing by eight, discarding the remainder, and then multiplying by eight. Dividing by eight is equivalent to three shifts right. Multiplying by eight is equivalent to three shifts left. The net effect of these shifts is to clear the three least-significant bits in XLO , an operation that can also be accomplished by ANDing the number in XLO with $\$F8$. Since the number $(y \text{ AND } 7)$ occupies only three bits, it can be combined with the number $(XLO \text{ AND } \$F8)$ to form a single eight-bit number. In our program, we shall let

$$Y = (XLO \text{ AND } \$F8) + (y \text{ AND } 7)$$

Since the two numbers in parentheses occupy different bits, we can more easily combine them with an ORA instruction rather than the "+" operation.

With this agreement, we are left with

$$\text{BAH:BAL} = \$A000 + \$140 * (\text{INT}(y/8)) + (XHI)$$

where (XHI) means "the number in XHI ." We need to separate this equation into two equations, one involving BAH and the other BAL . It should be clear that $\$A0$ and (XHI) go into BAH .

Since

$$\$140 * (\text{INT}(y/8)) = (\$100 + \$40) * (\text{INT}(y/8))$$

then

$$\$140 * (\text{INT}(y/8)) = \$100 * (\text{INT}(y/8)) + \$40 * (\text{INT}(y/8))$$

Multiplication by $\$100$ is the same as multiplication by 256, which is equivalent to eight shifts left. In other words, multiplication by $\$100$ shifts $(\text{INT}(y/8))$ from a low byte to a high byte, where it becomes part of BAH .

So far we have

$$\text{BAH} = \$A0 + \text{INT}(y/8) + (XHI)$$

leaving

$$\text{BAL} = \$40 * (\text{INT}(y/8))$$

but $\$40 * (\text{INT}(y/8))$ may not be able to be contained in BAL . In that case, we will have to add its most-significant byte to the expression for BAH given above. We begin by calculating BAL .

Since $\$40 = \$8 * \$8$, we may write

$$\text{BAL} = \$8 * (\$8 * (\text{INT}(y/8)))$$

Earlier we pointed out that dividing by eight, discarding the remainder, and multiplying by eight is equivalent to ANDing with $\$F8$. Thus, the previous formula reduces to

$$\text{BAL} = \$8 * (y \text{ AND } \$F8)$$

To calculate BAL, we begin by ANDing y with \$F8. Then we multiply by \$8 with a succession of three shifts left. This may move part of the result into a more significant byte. The low byte of this result becomes BAL. The high byte of this result is added to the sum

$$\$A0 + \text{INT}(y/8) + (\text{XHI})$$

to give BAH, finishing the calculation.

The subroutine to perform the calculation of Y, BAL, and BAH is given in Example 11-5. Its function is to set up the memory locations IAL and IAL+1 and the Y index register with the correct numbers, given the position (x,y) of a pixel. If you have understood the previous discussion, you should have no difficulty with the program since the function of each block is clearly identified and the calculation proceeds exactly as we have outlined above. The location symbolized by XPOS holds the least-significant byte of x, while XPOS+1 holds the most-significant byte of x. The y-coordinate is in the memory location symbolized by YPOS. Study this subroutine as you review the calculations we have described.

Example 11-5. Subroutine to Calculate ADH:ADL from (x,y)

Object: Load IAL, IAL+1, and Y with the numbers to identify the byte in bit-mapped memory that controls the pixel at (x,y).

```

10    ; CALCULATE Y FOR (IND),Y ADDRESSING MODE.
11  ADHADL  LDA #$F8      ;CLEAR BITS 0, 1, AND 2 OF X.          C860 A9 F8
12    AND +XPOS             C862 25 FD
13    STA +TEMP             C864 85 02
14    LDA +YPOS              ;ISOLATE THREE LOW BITS OF Y.   C866 A5 FF
15    AND #07                C868 29 07
16    ORA +TEMP              ;COMBINE WITH PREVIOUS RESULT. C86A 05 02
17    TAY                   ;PUT IN Y REGISTER FOR (IND),Y MODE. C86C A8
18    ;
19    ;
20    ;CALCULATE BAL FOR (IND),Y ADDRESSING MODE.
21    LDA #00                C86D A9 00
22    STA +IAL+1             ;CLEAR IAL+1 LOCATION.          C86F 85 FC
23    LDA #$F8                C871 A9 F8
24    AND +YPOS               C873 25 FF
25    ASL A                  ;MULTIPLY BY 8 WITH THREE     C875 0A
26    ROL +IAL+1             ;SHIFTS AND ROTATES.          C876 26 FC
27    ASL A                  C878 0A
28    ROL +IAL+1             C879 26 FC
29    ASL A                  C87B 0A
30    ROL +IAL+1             C87C 26 FC
31    STA +IAL                ;CALCULATION OF BAL IS COMPLETE. C87E 85 FB
32    ;
33    ;
34    ;CALCULATE BAH FOR (IND),Y ADDRESSING MODE.
35    LDA +YPOS               C880 A5 FF
36    LSR A                  ;DIVIDE BY 8 WITH 3 SHIFTS LEFT. C882 4A
37    LSR A                  C883 4A
38    LSR A                  C884 4A
39    CLC                   ;ADD HIGH BYTE OF X.          C885 18
40    ADC +XPOS+1            C886 65 FE
41    ADC #$A0                ;ADD BASE ADDRESS OF HIRES SCREEN. C888 69 A0
42    ADC +IAL+1              ;ADD PREVIOUS RESULT.          C88A 65 FC
43    STA +IAL+1              ;CALCULATION OF BAH IS COMPLETE. C88C 85 FC
44    RTS                   C88E 60

```

D. Switching out the ROM

Recall that we have placed bit-mapped memory in the same location as the BASIC interpreter in ROM. There is no problem with the 6510 *writing* to the R/W memory that occupies this 8K block of memory, but when it attempts to *read* codes from this part of memory it will read the codes in ROM. On the other hand, the VIC always reads the R/W memory, not the ROM.

A problem arises when we wish to turn on a pixel without affecting the other pixels controlled by the bits in that byte. This will require that we first perform an ORA instruction that involves a byte in the bit-mapped memory. An ORA instruction requires a *read* operation on the part of the 6510. Thus, we must switch out the BASIC ROM before we can read a location in bit-mapped memory.

The Commodore 64 system provides this option. Bits zero and one of the I/O port (IOP) at location \$0001 allow either the BASIC ROM, the operating system ROM, or both to be switched out of the memory space. These facts are summarized in Table 11-6. In the particular case we are considering, we wish to replace the BASIC ROM with R/W memory, so we must switch bit zero of the IOP at location \$0001 to zero before we attempt to read bit-mapped memory. This may be accomplished with the following sequence of instructions:

```
LDA IOP
AND #$FE
STA IOP
```

The ROM must be switched back in before returning to BASIC. This may be accomplished with the following sequence of instructions:

```
LDA IOP
ORA #$01
STA IOP
```

It is also good practice to disable interrupts when replacing either BASIC or the operating system ROM with R/W memory. SEI and CLI instructions will accomplish this.

Table 11-6. Switching ROM and R/W memory.

IOP Bit #	Name	Value	Function
0	LORAM	0	R/W memory from \$A000 to \$BFFF
		1	BASIC ROM from \$A000 to \$BFFF
1	HIRAM	0	R/W memory from \$E000 to \$FFFF
		1	Operating system ROM from \$E000 to \$FFFF

It should be clear that we could have put bit-mapped memory in the 8K block from \$E000 to \$FFFF. In that case, we would have to switch the operating system ROM out of the memory space rather than the BASIC ROM. Refer to Table 11-6 to see that bit one must be switched to zero to accomplish this.

E. Plotting points in the bit-mapped mode

To plot a point on the screen, we need to set one bit in the byte of bit-mapped memory identified by ADH:ADL, calculated above. The bit number of the bit we must set was also calculated above. It is $(7 - x \text{ AND } 7)$. Adding one to this number gives the number of right shifts to move the carry flag into the correct bit. Thus, our technique involves setting the carry flag and then rotating the carry into the appropriate bit.

All of this is accomplished with our PLOT subroutine listed in Example 11-6. This subroutine does three things. It switches the BASIC ROM out of the system and disable interrupts. Given the x-coordinate, it places a one in the bit position appropriate to that coordinate in the accumulator. Finally, it performs an OR operation involving the number in A and the appropriate byte in the bit-mapped memory, and stores the result in bit-mapped memory, turning on the pixel identified by (x, y) . It does this by first calling the subroutine in Example 11-5 to identify ADH:ADL. Study this program and the comments in connection with our discussion.

Example 11-6. The Plot Subroutine

Object: Turn on the pixel at (x, y) without affecting the status of any other pixel.

10	PLOT	JSR ADHADL	; TURN X AND Y INTO AN ADDRESS.	C890	20	60	C8
11		SEI	; DISABLE OPERATING SYSTEM INTERRUPTS.	C893	78		
12		LDA +\$01	; CLEAR BIT ZERO OF 6510 IOP	C894	A5	01	
13		AND #\$FE	; TO SWITCH BASIC ROM OUT	C896	29	FE	
14		STA +\$01	; OF SYSTEM TO ACCESS R/W MEMORY.	C898	85	01	
15		LDA +XPOS	; IDENTIFY BIT POSITION IN BYTE.	C89A	A5	FD	
16		AND #7	; MASK ALL BUT LOW THREE BITS.	C89C	29	07	
17		TAX	; PUT IN X FOR A COUNTER.	C89E	AA		
18		INX	; INCREMENT COUNTER.	C89F	E8		
19		LDA #0	; CLEAR A.	C8A0	A9	00	
20		SEC	; CARRY WILL BE ROTATED INTO A.	C8A2	38		
21	BIT	ROR A	; ROTATE RIGHT X TIMES.	C8A3	6A		
22		DEX	; NEXT X.	C8A4	CA		
23		BNE BIT		C8A5	D0	FC	
24		ORA (IAL),Y	; COMBINE WITH OTHER BITS IN	C8A7	11	FB	
25		STA (IAL),Y	; THE SAME BYTE IN MEMORY.	C8A9	91	FB	
26		LDA +\$01	; SWITCH BASIC ROM BACK INTO	C8AB	A5	01	
27		ORA #\$01	; COMMODORE SYSTEM.	C8AD	09	01	
28		STA +\$01		C8AF	85	01	
29		CLI	; ENABLE INTERRUPTS.	C8B1	58		
30		RTS		C8B2	60		

A subroutine to complement the routine in Example 11-6 is given in Example 11-7. The program in Example 11-7 turns off a pixel identified by (x, y) . That is, it *clears* the bit in bit-mapped memory that corresponds to the pixel whose position is specified by (x, y) . Sometimes it is desirable simply to move a dot around on the screen rather than to plot a curve. In this case the pixel is first turned on, and after a short delay it is turned off. This gives the appearance of motion. The routine in Example 11-7 works exactly like the routine in Example 11-6, except that a bit is cleared rather than set.

Example 11-7. The Unplot Subroutine

Object: Turn off the pixel at (x,y) without affecting the status of any other pixel.

10 UNPLOT	JSR ADHADL	;TURN X AND Y INTO AN ADDRESS.	C8C0 20 60 C8
11 SEI		;DISABLE OPERATING SYSTEM INTERRUPTS.	C8C3 78
12 LDA +\$01		;CLEAR BIT ZERO OF 6510 IOP	C8C4 A5 01
13 AND #\$FE		;TO SWITCH BASIC ROM OUT	C8C6 29 FE
14 STA +\$01		;OF SYSTEM TO ACCESS R/W MEMORY.	C8C8 85 01
15 LDA +XPOS		;IDENTIFY BIT POSITION IN BYTE.	C8CA A5 FD
16 AND #7		MASK ALL BUT LOW THREE BITS.	C8CC 29 07
17 TAX		PUT IN X FOR A COUNTER.	C8CE AA
18 INX		INCREMENT COUNTER.	C8CF E8
19 LDA #0		CLEAR A.	C8D0 A9 00
20 SEC		CARRY WILL BE ROTATED INTO A.	C8D2 38
21 BIT ROR A		ROTATE RIGHT X TIMES.	C8D3 6A
22 DEX		NEXT X.	C8D4 CA
23 BNE BIT			C8D5 D0 FC
24 EOR #\$FF		;COMPLEMENT THIS BYTE.	C8D7 49 FF
25 AND (IAL),Y			C8D9 31 FB
26 STA (IAL),Y			C8DB 91 FB
27 LDA +\$01		;SWITCH BASIC ROM BACK INTO	C8DD A5 01
28 ORA #\$01		COMMODORE SYSTEM.	C8DF 09 01
29 STA +\$01			C8E1 85 01
30 CLI		ENABLE INTERRUPTS.	C8E3 58
31 RTS			C8E4 60

F. A bit-mapped graphics demonstration routine

The program in Example 11-8 will demonstrate the subroutines that we have just been discussing. It plots a polar graph known as a five-leaved rose. It also demonstrates how the bit-mapped graphics routines are called from a BASIC program. On line 1, we call two of our subroutines: the first sets up the bit-mapped mode and the second clears bit-mapped memory. On line 2, we POKE the desired colors into memory location \$0002, and then call the routine to load these colors into screen memory. We chose black dots (color code \$0) on a light red background (color code \$A). The next line calculates a function R of an angle ANG. Line 4 can be replaced with almost any function that expresses the radius R as a function of an angle. On lines 5 and 6, the x- and y-coordinates corresponding to R and ANG are calculated and rounded to the nearest integer. The coordinate system is also shifted so that its origin (the pole) is at (150, 100), near the center of the screen.

It is important to understand lines 7 and 8. This is where the coordinates x and y are POKEd into memory for use by our assembly-language routine. First the LSB of x is POKEd into location 253, then the MSB of x is POKEd into 254. Since y is a single-byte number, it is simply POKEd into location 255. The PLOT subroutine in Example 11-6 is called next and the point is plotted. The angle is incremented by about one degree, and the program loops back to calculate and plot another point. Load all of the bit-mapped graphics subroutines, then load and run this BASIC program.

Example 11-8. A BASIC Program to Demonstrate the Bit-Mapped Graphics Routines

```
0 REM EXAMPLE 11-8
1 SYS 51200:SYS 51232
2 POKE 2,10:SYS 51264
3 ANG=0
4 R=90*SIN(5*ANG)
5 X=150+R*COS(ANG):X=INT(X+.5)
6 Y=100-R*SIN(ANG):Y=INT(Y+.5)
7 POKE 253,XAND255:POKE 254,X/256
8 POKE 255,Y
9 SYS 51344
10 ANG=ANG+3.14159/180
11 GO TO 4
```

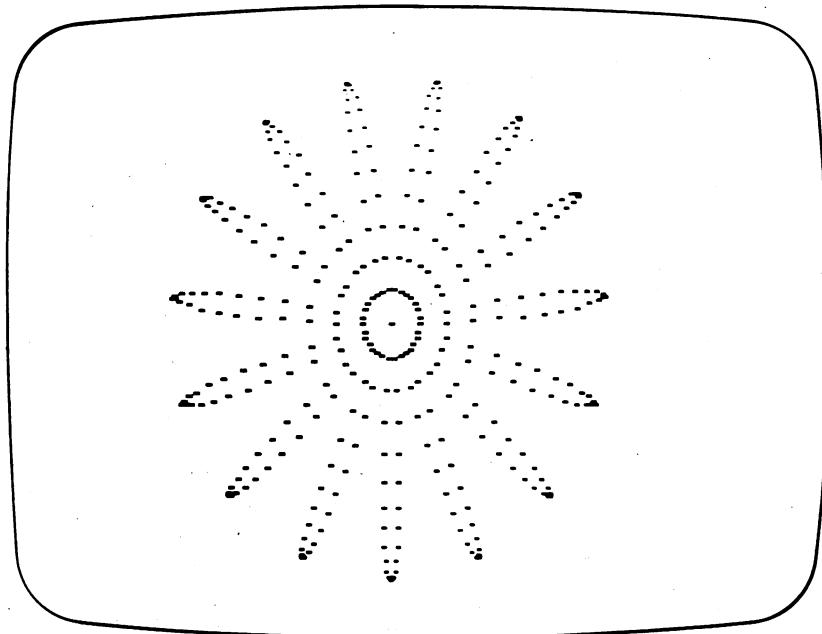


Figure 11-3. Photograph showing the output of the program in Example 11-8 with $R = 90 \cdot \text{SIN}(5 \cdot \text{ANG})$ on line 4.

The routine in Example 11-9 can be used to reduce the number of SYS commands required to call the bit-mapped graphics subroutines. If the color codes are loaded into location \$0002, a single call to the routine in Example 11-9 will set up the bit-mapped memory so that the PLOT subroutine can be called. In other words, this routine replaces the three SYS commands in Example 11-8 with one SYS 51440 command.

Example 11-9. A Routine to Call the Bit-Mapped Graphics Subroutines

Object: Reduce the three SYS commands required to use bit-mapped graphics to one SYS 51440 command.

10	Hires	JSR	MEMSET	; SET MEMORY POINTERS IN VIC.	C8F0	20	00	C8
11		JSR	CLEAR	;CLEAR HIRES MEMORY.	C8F3	20	20	C8
12		JSR	SETCLR	;PLACE COLORS IN SCREEN MEMORY.	C8F6	20	40	C8
13		RTS			C8F9	60		

This completes our discussion of bit-mapped graphics and the VIC. Of course, there are many topics that we could not pursue. As in the case of the SID, the VIC is sufficiently powerful that several chapters, perhaps even a book, would not do it justice. We made a more or less arbitrary decision to introduce its bit-mapped graphics capability. Exploring its MOB (movable object blocks) or sprite capabilities would have been equally exciting.

At the very least, you should have seen a programming style emerge during the last two chapters. This style includes reducing a programming problem to a set of smaller problems, each of which can be handled with a relatively simple subroutine.

IV. Summary

The 6567 video interface chip (VIC) translates information in memory into video information on the screen. The VIC can be made to use any of four 16K banks of memory in the 64K memory space of the Commodore 64. This 16K block contains the information needed to display characters, colors, and movable object blocks or to do bit-mapped graphics.

Screen codes in a 1,000-byte block of screen memory are combined with character display information in a 2K block of character memory to display text or graphics characters on the screen. The colors of these characters are determined by the color codes in color memory. The location of screen memory and character memory is easily changed, allowing multiple screens and character sets to be stored at one time.

In the bit-mapped mode, each bit in a 8,000-byte block of memory is mapped into one of the 320-by-200 (64,000) pixels on the screen. If the bit has a value of one, the pixel has one color; if it has a value of zero, it has another color. These color codes are stored in screen memory. The main goal of any bit-mapping program is to identify the location of the bit in memory that controls the status of the pixel located on the screen at the point (x,y). This capability allows you to graph mathematical functions such as sine waves, ellipses, and parabolas. You can plot circuit diagrams, contour maps, or play Etch-a-Sketch. You can simulate a satellite circling the earth, with a death ray streaking out to destroy it.

V. Exercises

1. Execute the subroutine CLEAR in Example 11-3 with a SYS 51232 command. How long do you think it takes to execute this subroutine? Now write a BASIC program to perform the same function. How long does it take?
2. Think of a way to test subroutine UNPLOT in Example 11-7, then test it.
3. Write a BASIC program that uses the bit-mapped graphics subroutines to draw the axes of an x,y coordinate system on the screen. Put the origin of the coordinate system at the screen point (160,100).
4. Consult your *Programmer's Reference Guide* to see how the VIC is disabled. Write one routine to disable it, and another routine to enable it. All programs run more quickly with the VIC disabled because it stops the 6510 microprocessor to obtain display information.
5. Modify the programs in the bit-mapped graphics section to use the 8K block of memory from \$E000 to \$FFFF for bit-mapped memory.
6. Consult your *Programmer's Reference Guide* for a description of the multi-color bit-mapped mode. Modify the routines in this chapter to work in the MCM (multi-color mode).

This mode is enabled by setting bit five in location \$D011, the BMM enable bit, and by setting bit four in location \$D016. In this mode, the colors on the screen are determined by *pairs* of bits. Each *pair* of bits in the bit-mapped memory controls the color of two *adjacent* pixels. Thus, the horizontal resolution is cut in half, giving a display of 160 pixels wide by 200 pixels down. In return for a reduction in resolution, you obtain a greater variety of colors. In the MCM/BMM mode, four pairs of bits in a byte in the bit-mapped memory determine the color of the two-pixel-wide dot on the screen in this manner:

Bit Values	Color Code Comes From
0 0	Background color #0 register (\$D021)
0 1	High nibble in screen memory
1 0	Low nibble in screen memory
1 1	Low nibble in color memory

0 0	Background color #0 register (\$D021)
0 1	High nibble in screen memory
1 0	Low nibble in screen memory
1 1	Low nibble in color memory

12

Input/Output: The 6526 Complex Interface Adapter

I. Introduction

In this chapter, you will learn how to connect your computer to the outside world. The Commodore 64 has two *game control ports* and a *user port* that allow you to connect your computer to external devices. It is these ports that we will describe and use in this chapter.

To make effective use of these ports, you must either have or acquire a minimal amount of electronics experience. After all, every connection you make to the computer is an electrical connection. For some programs, all you will have to do is plug in a paddle or joystick; for others, you may have to make several solder connections. We will document our work with photographs and diagrams to help you feel comfortable with your wiring.

If you decide to do some of the electronic projects, you can obtain many of the parts from a Radio Shack store. If you cannot find the parts there, try one of the many electronic parts mail-order stores that advertise on the back pages of many computer magazines. We obtained most of our parts from:

Priority One Electronics
9161 Deering Ave.
Chatsworth, CA 91311
(213) 709-5464

Most firms require a minimum order, so read the entire chapter to see what parts you may want to order before placing an order.

The Commodore 64 comes with two 6526 CIA (complex interface adapter) integrated circuits. These two chips are used to interface the computer to the keyboard, the two game control ports, and the user port. You do not make any connections directly to the 6510 microprocessor: all I/O signals go through an interface adapter. The 6526 CIA contains two I/O ports, two counter/timers,

a time-of-day clock, and a serial port. The focus of this chapter will be the I/O ports and the counter/timers. We begin with a discussion of some fundamental I/O concepts.

II. Input/Output Fundamentals

A. Input ports

In an eight-bit microcomputer such as the Commodore 64, information is stored, transferred, and manipulated in the form of eight-bit binary codes. Even if you could peer inside a memory chip, the 6510 microprocessor, or another integrated circuit inside the computer, you would not see eight-bit binary codes lying about waiting to be read or modified. The ones and zeros are merely a convenient way for *human beings* to view the information in the computer. The computer deals with *voltage levels*, not numbers. A binary *one* corresponds to a voltage level between two and five volts, while a binary *zero* corresponds to a voltage level between zero and 0.8 volt. Voltages between 0.8 and two volts are *indeterminate* as far as their binary value is concerned. The integrated circuits in the microcomputer usually switch very rapidly from one binary level to another, passing through the indeterminate state as quickly as possible. In short, a voltage near five volts is interpreted as a one, while a voltage near zero volts is interpreted as a zero.

It should be obvious that to input information to the computer, an external device of some kind must control the voltage levels on one or more *input pins* connected to the microcomputer. External devices include mechanical switches, phototransistors (in a light pen, for example), or other integrated circuits outside of the computer. An *input port* consists of one to eight input pins. Like a memory location, an input port is identified by *one* of the 65,536 addresses that the 6510 is capable of producing. Thus, one way of reading the information available at an input port is with an

LDA PORT

instruction, where PORT is a symbol for the address of the input port. Any 6510 instruction that involves a read operation may use the information that is available at the input port.

The pins of an input port are numbered in the same way as the bits in a memory location. Thus, an input port consisting of eight pins would have the pins numbered

P7, P6, P5, ..., P0

The pin numbers correspond to the bit numbers of the eight-bit code that is obtained when the input port is read. If the voltage on pin P0 is near five volts, then when the port is read, bit zero will be one. If the voltage on pin P0 is near zero volts, then when the port is read, bit zero will be zero. The same is true for the other seven pin/bit combinations.

B. Output ports

To control an external device, the computer must control the voltage levels on one or more *output pins*. External devices include lights, relays, or other integrated circuits, such as in a printer. An *output port* consists of one to eight output pins. Like a memory location, an output port is identified by *one* of the 65,536 addresses that the 6510 is capable of placing on the address bus. One way of writing to an output port is with an

STA PORT

instruction, although any 6510 instruction that involves a write operation may be used to modify the information at the output port.

The pins of an output port are numbered in the same way as the bits in a memory location. Thus, an output port consisting of eight pins would have the pins designated as

P7, P6, P5, ..., P0

The pin numbers correspond to the bit numbers of the byte that is written to the output port. If this byte has a one in bit seven, then pin P7 will have a voltage level near five volts. If this byte has a zero in bit seven, then pin P7 will have a voltage level near zero volts. The same is true for the other seven bit/pin combinations.

It is important to realize that the *electrical power* available at an output pin is very limited. Do not expect to light even a small flashlight bulb with the five-volt signal on an output pin. It is capable of supplying (sourcing) much less than one milliamp (0.001 ampere) of current. Needless to say, you are not going to use your computer to supply the power to start your car or heat your coffee.

Nor does the output pin produce much of a *short* (or current sink) when it is at zero volts. It will draw (sink) only about one or two millamps. If you attempt to supply more current to it, the voltage will rise and it will no longer correspond to a bit value of zero.

What all of this means is that it is usually necessary to have one or more integrated circuits *between* the output pin and the device you wish to control.

C. Memory-mapped I/O and interfacing

To summarize what we have just explained, an input/output port (I/O port) is a location in the memory map of the computer that can be used to transfer information either from an external device to the computer or from the computer to an external device. The type of input/output operations that use ports in the memory map of the computer system is called *memory-mapped I/O*. These ports may be accessed with any of the instructions in the 6510 instruction set that involve a read or write operation.

The type of I/O in which from one to eight bits of information are transmitted simultaneously is called *parallel I/O*. If only one bit can be output at a time, then the I/O operations are described as *serial I/O*. This chapter will deal exclusively with parallel I/O.

The process of designing and connecting devices that connect the computer to the outside world is called *interfacing*. We will keep our interfacing projects as simple and interesting as possible.

III. Input/Output with the 6526 CIA

A. Data-direction registers and I/O ports

The Commodore 64 has two 6526 CIAs, called CIA #1 and CIA #2. Each CIA has two eight-bit I/O ports, Port A and Port B. A diagram of the I/O ports associated with CIA #1 is shown in Figure 12-1. CIA #1 I/O ports are used for keyboard and game control functions. The operating system makes extensive use of this CIA, and some care must be taken not to disturb the operating system when using CIA #1. Port A of CIA #2 is also used by the operating system. This leaves Port B of CIA #2 to interface with the user port. The addresses of the I/O ports are given in Table 12-1.

Table 12-1. Memory locations of the complex interface adapter registers.

<i>Symbol</i>	<i>Address</i>	<i>Description</i>	<i>CIA</i>
PRA1	\$DC00	Port A data register I/O pins PA7 PA6 PA5 PA4 PA3 PA2 PA1 PA0	#1
PRB1	\$DC01	Port B data register I/O pins PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0	#1
DDRA1	\$DC02	Port A data-direction register	#1
DDRB1	\$DC03	Port B data-direction register	#1
PRA2	\$DD00	Port A data register I/O pins PA7 PA6 PA5 PA4 PA3 PA2 PA1 PA0	#2
PRB2	\$DD01	Port B data register I/O Pins PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0	#2
DDRA2	\$DD02	Port A data-direction register	#2
DDRB2	\$DD03	Port B data-direction register	#2

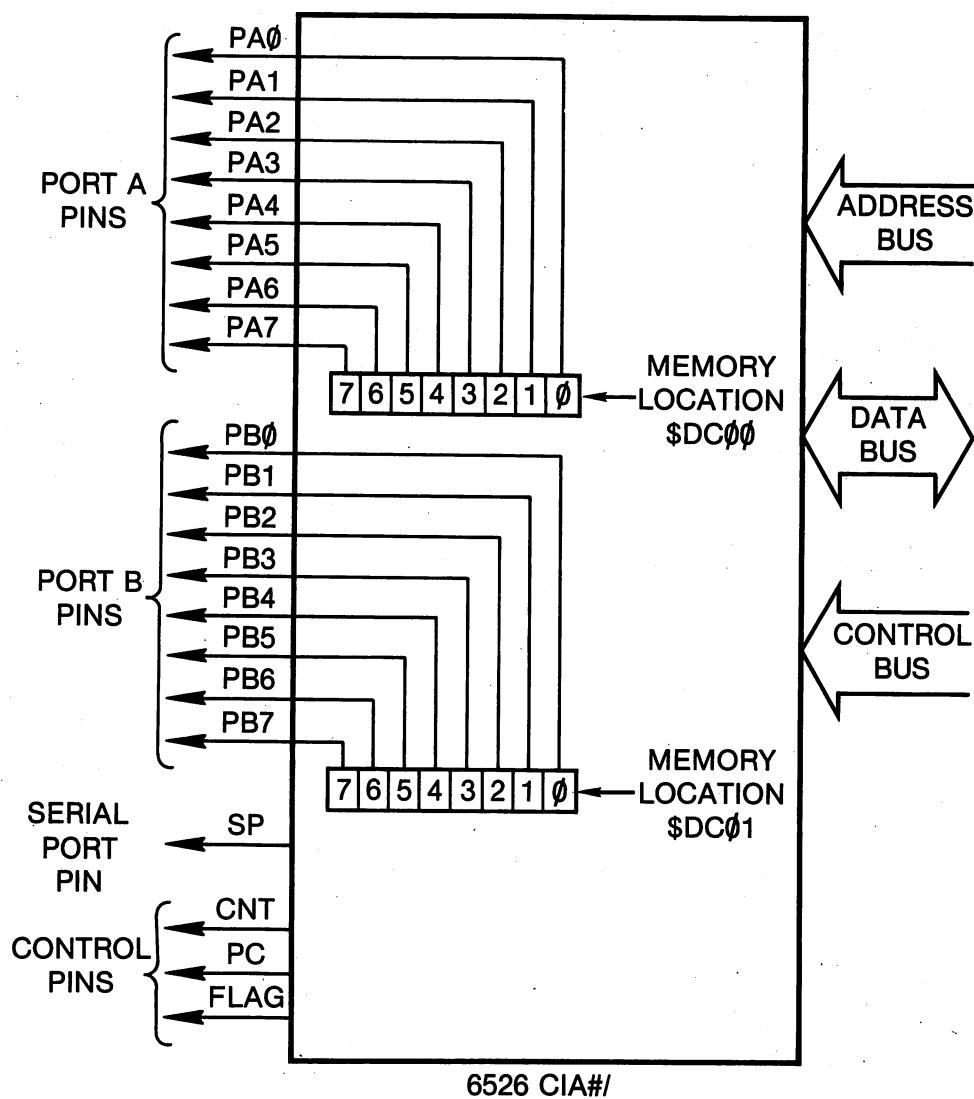


Figure 12-1. The I/O port pins and memory locations on the 6526 CIA #1.

Associated with each port is a *data-direction register*. It should be clear that the information or data moves in one direction, from an external device to the computer, if the port is an input port, while it moves in the other direction, from the computer to an external device, if the port is an output port. The code stored in a port's data-direction register determines whether the port is an input or output port.

- A port is an input port if each bit in its data-direction register is zero.
- A port is an output port if each bit in its data-direction register is one.

Thus, a port may be *programmed* to be either an input port or an output port, depending on the code stored in its DDR (data-direction register). The following sequence of four instructions make Port A an output port and Port B an input port:

```
LDA # $FF  
STA DDRA  
LDA # $00  
STA DDRB
```

In fact, any *individual pin* of a port may be programmed to be either an input pin or an output pin. There is a one-to-one correspondence between the bit numbers of the bits in the DDR and the I/O port pin numbers.

- If a bit in the DDR is zero, then the corresponding I/O pin is an input pin.
- If a bit in the DDR is one, then the corresponding I/O pin is an output pin.

Assume we want PA3 through PA0 to be output pins and PA7 through PA4 to be input pins. The following two instructions suffice:

```
LDA # $0F  
STA DDRA
```

B. A joystick as an input device

We will begin our examples with a program that illustrates how to read the five input pins associated with the Commodore 64 Game Control Port 2. This port is used to read either a joystick with a fire button or a game paddle with a fire button. In Example 12-1, we illustrate how to read the port with a joystick attached.

A joystick consists of five switches, four of which are used to control the direction in which something on the screen moves; the other is the fire button, used to destroy something on the screen, such as a sprite. There is one switch for each direction, up, down, left, and right. In certain positions of the joystick, two switches are closed at one time. You can have combinations such as up-left, down-right, up-right, and up-left. Refer to Figure 12-2 if you are interested in the circuit diagram.

The first line in Example 12-1 disables the interrupts from the operating system. This is important since the same port that reads the joystick is used by the operating system to read the keyboard. The next three lines clear bits zero to four of the Port A data-direction register (DDRA), ensuring that the corresponding pins of Port A act as input pins.

The function of lines 16 through 18 in Example 12-1 is to wait for the fire button to be pressed. If it is left unconnected, an input pin on the CIA will "float" to five volts, corresponding to a binary one. When the fire button is pressed, it connects the input pin to zero volts (ground), corresponding to a binary zero. Thus, the program waits in the loop until the fire button connected to pin PA4 of the CIA is pressed.

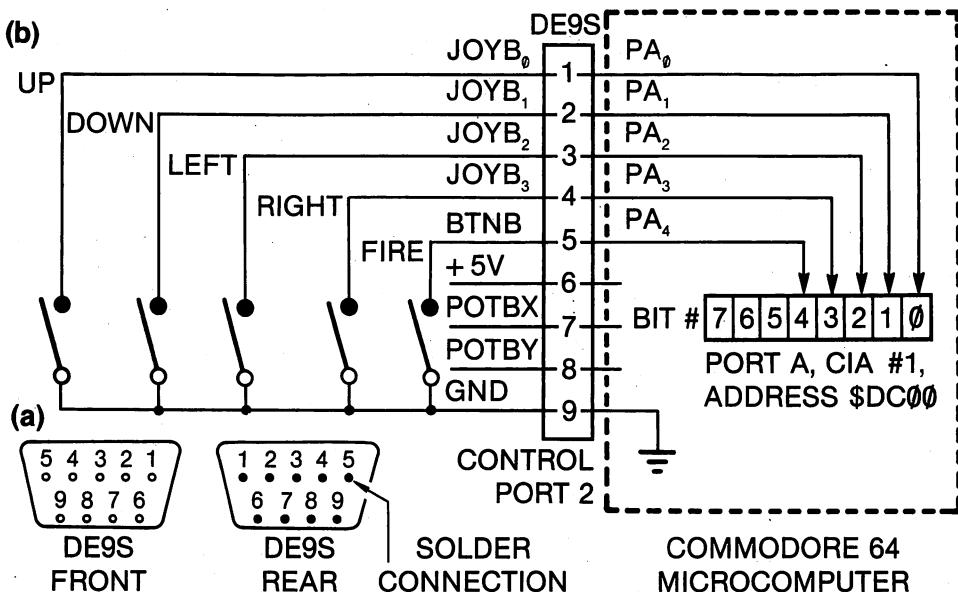


Figure 12-2. Circuit diagram of a joystick. (a) Pin diagram of the DE9S connector. (b) Schematic diagram of the joystick circuit. The switches SW can be either pushbutton, toggle or slide switches, or they can be the switches in a commercial joystick.

After the fire button is pressed, the program reads Port A on line 20 of the program, masks the bits that are not connected to Control Port 2 on the Commodore 64, and stores the result in location \$0002. This completes all operations involved in configuring and reading Port A. It is illustrative, however, to display the binary values obtained by reading the four least-significant bits of Port A, the values of which are determined by the setting of the joystick. The remainder of the program, lines 24 through 32, is used to display the partially masked byte obtained from Port A. Remember, the most-significant nibble was masked, so these four bits will be zero.

Example 12-1. Reading the Joystick Inputs

Object: Read and display the bit values of Port A, pins PA0 through PA3, determined by the setting of the joystick in Control Port 2.

```

10 START    SEI      ;DISABLE SYSTEM INTERRUPTS.          C000 78
11      ;                                 C001 A9 E0
12      LDA #$E0   ;CONFIGURE PORT A TO HAVE PINS          C003 2D 02 DC
13      AND DDRA   ;PA0-PA4 AS INPUT PINS.                  C006 8D 02 DC
14      STA DDRA
15      ;
16      LDA #$10   ;MASK FOR FIRE BUTTON BIT.            C009 A9 10
17 LOAF     BIT PRA   ;IS FIRE BUTTON PRESSED?           C00B 2C 00 DC
18      BNE LOAF   ;NO, SO WAIT FOR IT.                 C00E D0 FB
19      ;
20      LDA PRA    ;YES, READ THE PORT.                C010 AD 00 DC
21      AND #$0F   ;MASK MOST SIGNIFICANT NIBBLE.       C013 29 0F
22      STA +$02   ;STORE JOYSTICK CODE HERE.        C015 85 02

```

```

23      ;
24      LDX #$08      ;DISPLAY EIGHT BITS.          C017 A2 08
25 BACK   ASL +$02      ;SHIFT LEFT INTO CARRY.    C019 06 02
26      LDA #00      ;CLEAR A.                  C01B A9 00
27      ADC #$30      ;CONVERT TO ASCII.        C01D 69 30
28      JSR CHROUT    ;OUTPUT THE BIT VALUE.    C01F 20 D2 FF
29      DEX
30      BNE BACK      ;LOOP BACK TO DISPLAY EIGHT BITS. C022 CA
31      LDA #$0D      ;OUTPUT CARRIAGE RETURN. C023 D0 F4
32      JSR CHROUT    ;OUTPUT CARRIAGE RETURN. C025 A9 0D
33      JMP START     ;LOOP FOREVER.           C027 20 D2 FF
                                         C02A 4C 00 C0

```

Load the program into memory, then call it with a SYS 49152 command. Insert the joystick control into Control Port 2. Press the fire button, then move the joystick. Do the values you get for PA0 through PA3 agree with those given in Table 12-2? Write and test a program that reads Control Port 1. This port is connected to pins PB0 through PB3 with the fire button connected to PB4.

Table 12-2. Bit values determined by the joystick settings.

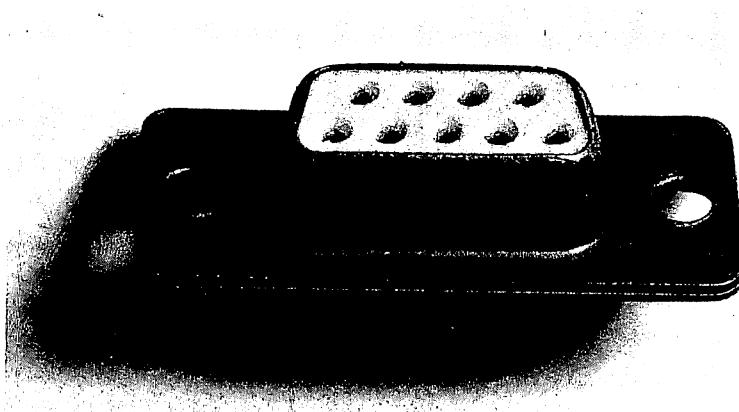
<i>Bit Value</i>				<i>Joystick Position</i>
PA3	PA2	PA1	PA0	
1	1	1	1	Control port 2
1	1	1	0	Neutral
1	1	0	1	Up
1	0	1	1	Down
0	1	1	1	Left
				Right

Perhaps you are interested in controlling these bit values with an ordinary switch rather than a joystick. A circuit diagram is given in Figure 12-2. You will need a DE9S (female) connector to attach to the DE9P (male) connector used for each of the game-control ports on the Commodore 64. The DE9 connectors are subminiature D-type connectors: one is shown in Figure 12-3.

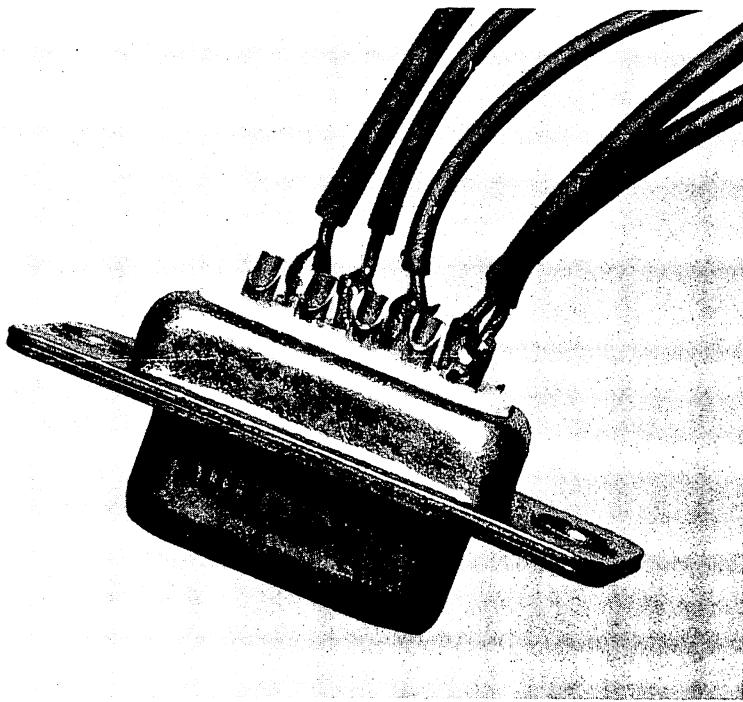
There are at least three approaches that you can use to connect your electronic components, resistors, capacitors, switches, and so on, to the control port connectors. You can solder wires to the pins on the DE9S connectors, and the other ends of these wires can be connected to the electronic components mounted on a breadboard or a more permanent circuit board. See Figure 12-3 for photographs that illustrate this approach.

The second approach involves soldering the nine pins of a DE9S connector to nine of the leads of a 14- or 16-conductor DIP jumper with a 14- or 16-pin DIP socket at the other end. This approach is shown in Figure 12-4. Although the soldering is tedious, this approach allows great flexibility in testing and prototyping various circuits. Breadboards for prototyping circuits will be shown in Figures 12-9 and 12-11.

(a)



(b)



**Figure 12-3. Photographs of the DE9S connector. (a) The connector.
(b) The connector with wires attached.**

A third approach is to dismantle a joystick and use its DE9S connector and cable. It is not difficult to solder other leads to the joystick cable leads and connect these to a breadboard. You are left to your own ingenuity if none of these approaches suits you.

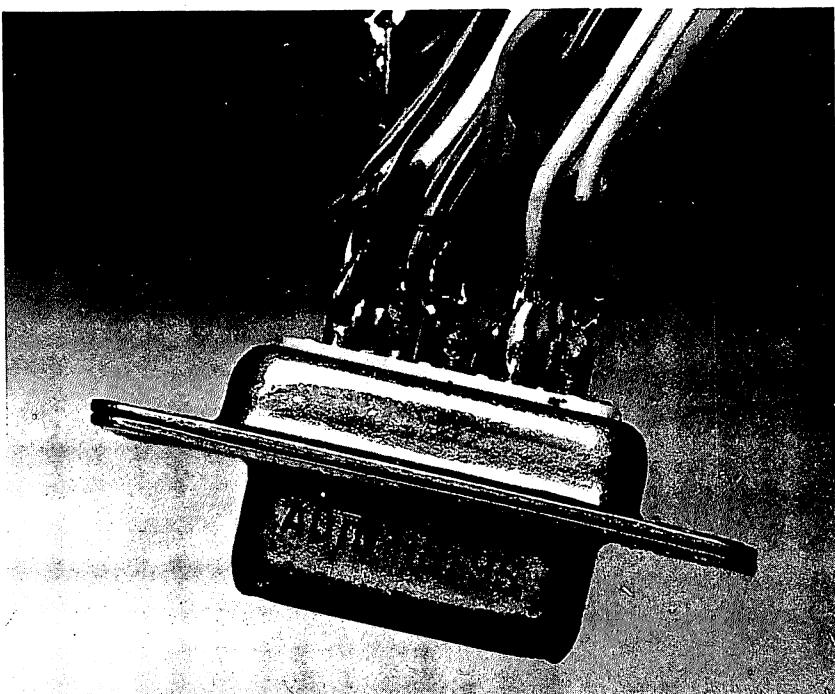


Figure 12-4. A DIP jumper connected to a DE9S connector.

C. A Commodore 64 control port as an output port

Now we will make use of an output port. Our next programming example allows us to select the so-called POT inputs on the SID and make analog-to-digital conversions. Pins PA7 and PA6 of Port A on CIA #1 control an integrated circuit known as an *analog switch*. This integrated circuit is identified by its number, 4066, on the Commodore 64 circuit diagram. The 4066 selects which of the paddle inputs connected to the control ports will be connected to the POTX and POTY inputs on the 6581 SID. Refer to Table 12-3 to see which paddle inputs are selected by PA7 and PA6. Notice that four different paddles may be selected, two from each control port.

Table 12-3. Using PA7 and PA6 to select a paddle.

<i>PA7</i>	<i>PA6</i>	<i>Control Port #</i>	<i>Paddles</i>	
1	0	2	POTBX	POTBY
0	1	1	POTAX	POTAY

To select which control port is going to be used, we must first make pins PA7 and PA6 output pins. One way to accomplish this is with the sequence:

```
LDA #$C0
STA DDRA
```

After that is accomplished, storing the number \$80 in Port A selects Control Port 2, while the number \$40 selects Control Port 1.

A program to select Control Port 2 for the paddle input and read POTX is given in Example 12-2. The first three lines of this program disable the interrupts from the operating system and save the current code in data-direction register A (DDRA) in memory. Upon returning to BASIC it will be important to leave DDRA unaffected, since Port A is also used to read the keyboard. Lines 13 and 14 store \$80 in Port A, selecting Control Port 2 for the paddle inputs. Note that pins PA7 and PA6 may not yet be output pins. They become output pins after the instructions on lines 15 and 16 of the program in Example 12-2 are executed.

The POT inputs on the SID require a minimum of 512 clock cycles before they provide valid information. Thus, in lines 18 through 20 the program waits in a delay loop. The POTBX input on the SID is read on line 22 of the program, and its value is stored in location \$0002 on line 23. The POTBY input would be read at location \$D41A. The value of the code in DDRA is restored on lines 24 and 25, and the program returns to the calling program on line 27.

Example 12-2. Making PA7 and PA6 Output Pins—Reading the POTBX Input

Object: Configure Port A on CIA #1 to have PA7 and PA6 as output pins, selecting Control Port 2 for paddle inputs. Read the POTBX input and store it.

10	ATOD	SEI	;DISABLE INTERRUPTS.	C000 78
11		LDA DDRA	;SAVE VALUE IN DDRA.	C001 AD 02 DC
12		STA \$0313		C004 8D 13 03
13		LDA #\$80	;IDENTIFY GAME PORT 2.	C007 A9 80
14		STA PRA	;SELECT THE ANALOG SWITCH.	C009 8D 00 DC
15		LDA #\$C0	;PINS PA7 AND PA6 WILL BE OUTPUT	C00C A9 C0
16		STA DDRA	;PINS TO ANALOG SWITCH.	C00E 8D 02 DC
17	;			
18		LDX #\$E0	;DELAY FOR AT LEAST 512 CYCLES.	C011 A2 E0
19	LOOP	DEX	;SHORT DELAY LOOP.	C013 CA
20		BNE LOOP		C014 D0 FD
21	;			
22		LDA POTX	;READ PADDLE X.	C016 AD 19 D4
23		STA +\$02	;STORE IT HERE.	C019 85 02
24		LDA \$0313	;GET DDRA BACK AGAIN.	C01B AD 13 03
25		STA DDRA	;RESTORE DDRA.	C01E 8D 02 DC
26		CLI	;ENABLE INTERRUPTS.	C021 58
27		RTS	;RETURN TO CALLING PROGRAM.	C022 60

In a moment we will describe how to use this program, but first we will briefly describe how the POT inputs work. Refer to Figure 12-5 for a schematic diagram of a POT input. You may also wish to remove the back from one of your paddle inputs to see the potentiometer inside. Rotating the knob on the potentiometer varies the *resistance* between the +5V connection and the POT input. This varies the rate at which the 1,000 pf capacitor *charges*. This capacitor is in the circuitry of the Commodore 64. You do not need to wire it in the

circuit. The higher the resistance, the longer it takes the capacitor to charge. Circuitry inside the SID measures how long it takes to charge to a certain voltage and it stores this number in one of its registers. The two POT registers are read at addresses \$D419 (POTX) and \$D41A (POTY).

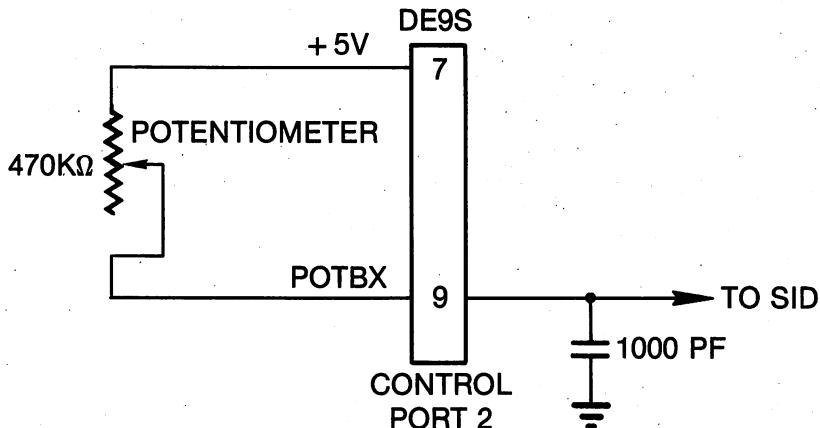


Figure 12-5. Circuit diagram of a game paddle input. There is an identical circuit for POTBY input.

Inside each of your game paddles is a 470K ohm potentiometer. Rotating it varies the number in the corresponding POT register from zero to 255. Resistance is an *analog* quantity which, by means of a POT input, can be represented with a number, a *digital* quantity. Thus, the POT inputs perform *analog-to-digital* conversions. This is an extremely useful thing to be able to do.

To demonstrate the program in Example 12-2, we will call it from the BASIC program in Example 12-3. This program also makes use of the assembly-language bit-mapped graphics routines, Examples 11-2 to 11-9, so they must be loaded into memory. The program in Example 12-3 graphs the paddle input along the y-axis, while the x-axis graphs time, making a graph of the potentiometer resistance as a function of time.

Example 12-3. A BASIC Program to Read and Display Paddle Input

```

5 REM EXAMPLE 12-3
10 POKE 2,10 :REM SET COLOR.
20 SYS 51440 :REM INITIALIZE BIT MAPPED GRAPHICS.
30 K=199/255
40 FOR X=0 TO 319 :REM COLLECT 320 POINTS.
50 SYS 49152:REM READ PADDLE
60 Y=PEEK(2):Y=INT(K*Y+.5)
70 POKE 253,XAND255:POKE 254,X/256
75 POKE 255,Y
80 SYS 51344
90 NEXT X
100 GO TO 10

```

Load the bit-mapped graphics machine-language routines, load the program in Example 12-2, and RUN the BASIC program in Example 12-3. Now rotate the paddle knob. You should see the graph go up and down on the screen. We looped a string through a pendulum bob and looped it around a game paddle knob. Then we started the pendulum swinging and started the program. Our results are shown in Figure 12-6. Notice that the amplitude of the swing decreases with time, an effect called damping. Also observe that the graph of the position of the pendulum as a function of time has a sine waveform.

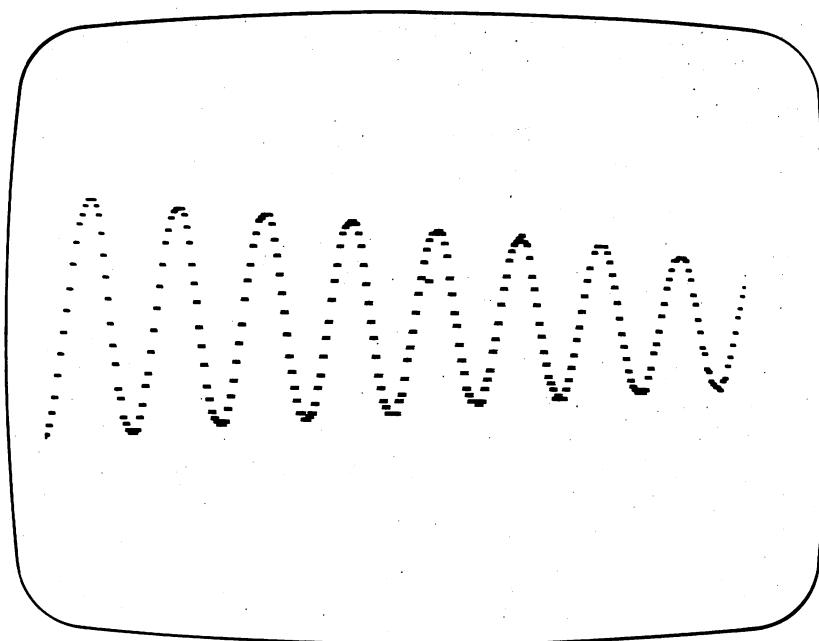


Figure 12-6. Output of Example 12-3 with a pendulum attached to the game-paddle potentiometer.

There are other devices that you can connect to the paddle inputs. Figure 12-7 shows that the potentiometer can be replaced by either a thermistor or a phototransistor. Using the same programs with the thermistor replacing the potentiometer gives a graph on the video monitor that shows how temperature varies with time. Phototransistors are sensitive to light, so replacing the thermistor with a phototransistor gives a graph on the video monitor that shows how the light intensity varies with time. Only one of the devices shown in Figure 12-7 should be in the circuit at one time. The phototransistor is available from Radio Shack stores, while thermistors can be obtained from:

Newark Electronics
500 N. Pulaski Road
Chicago, IL 60624
(312) 638-4411

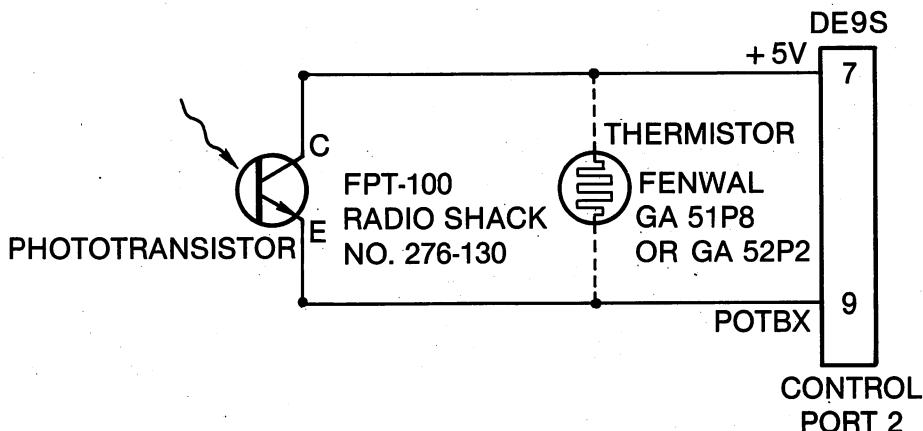


Figure 12-7. Two game paddle input circuits. The phototransistor is sensitive to light; the thermistor is sensitive to temperature.

The rate at which data is acquired from the POT input and plotted on the screen may be controlled by a FOR...NEXT loop inserted on line 55 of Example 12-3.

D. More input/output circuits

We conclude this section with two additional programming examples that illustrate simple I/O operations. The first program reads pin PA4, accessed at the fire button on Control Port 2, and outputs this bit to pin PA0, also available at Control Port 2. We connected the PA0 pin to a 7406 inverter in order to drive a light-emitting diode (LED), as shown in Figure 12-8. A photograph of this circuit appears in Figure 12-9. A breadboard similar to the one in the photograph is available at Radio Shack stores.

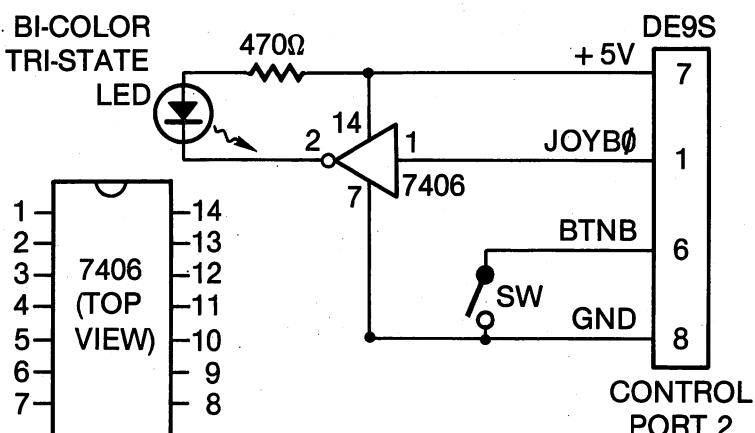


Figure 12-8. Circuit diagram of a one-bit output port. The one-bit output makes a light-emitting diode (LED) glow.

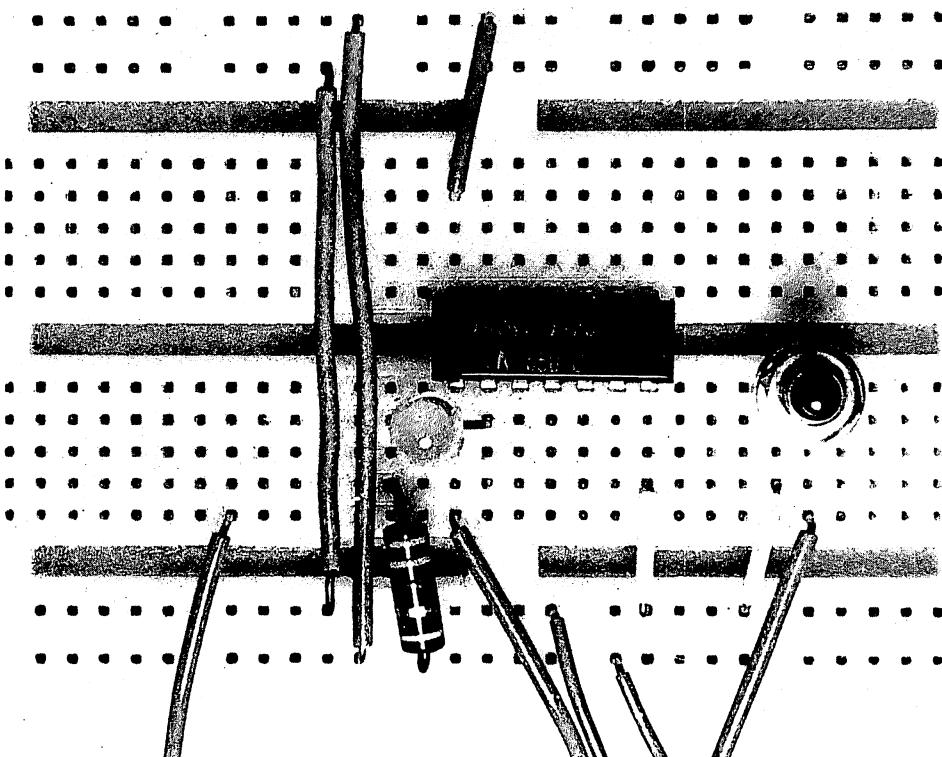


Figure 12-9. Photograph of the one-bit I/O port.

Writing a one to bit zero of Port A turns on the light-emitting diode (LED). Writing a zero to bit PA0 turns off the LED. We suggest using a Tri Color light-emitting diode available from Radio Shack stores, since the polarity of such an LED is of no consequence. It changes from a red glow to a green glow when the polarity is reversed.

A program to demonstrate how a computer can control an output device, such as the LED in the circuit of Figure 12-8, is given in Example 12-4. Lines 11 through 14 of this program are used to configure the Port A data-direction register (DDRA) so that pin PA0 is an output pin and pin PA4 is an input pin. The rest of the program merely transfers the value of bit four to bit zero. Pressing the push button switch turns off the LED: the LED glows when the switch is not touched. Wire the circuit and then load and execute the program. The LED should respond to changes in the switch condition. You have just created a very expensive switch.

Our last demonstration program and circuit elaborate on the previous ones. In Example 12-5 we have a program that reads the four least-significant bits of Port A on CIA #1 and transfers these values to the four least-significant bits of Port B on the same CIA. Port A is accessed at Control Port 2, and these bits may be controlled by a joystick plugged into this port or with the circuit shown

Example 12-4. One-Bit I/O Operations

Object: Transfer bit four, an input bit, to bit zero, an output bit, of Port A.

10	SEI	;DISABLE INTERRUPTS.	C000 78
11	LDA #\$EF	;CLEAR BIT 4 OF DDRA TO	C001 A9 EF
12	AND DDRA	;MAKE PIN PA4 AN INPUT PIN.	C003 2D 02 DC
13	ORA #\$01	;MAKE PA0 AN OUTPUT PIN.	C006 09 01
14	STA DDRA		C008 8D 02 DC
15	LOOP LDA PRA	;READ PORT A.	C00B AD 00 DC
16	AND #\$10	;MASK ALL EXCEPT BIT 4.	C00E 29 10
17	LSR A	;SHIFT BIT 4 INTO BIT 0.	C010 4A
18	LSR A		C011 4A
19	LSR A		C012 4A
20	LSR A		C013 4A
21	STA PRA	;STORE IT IN PORT A.	C014 8D 00 DC
22	JMP LOOP	;LOOP FOREVER.	C017 4C 0B C0

in Figure 12-2. Port B is accessed at Control Port 1, and a circuit to drive four light-emitting diodes (LEDs) from this port is shown in Figure 12-10. A photograph of this circuit is shown in Figure 12-11.

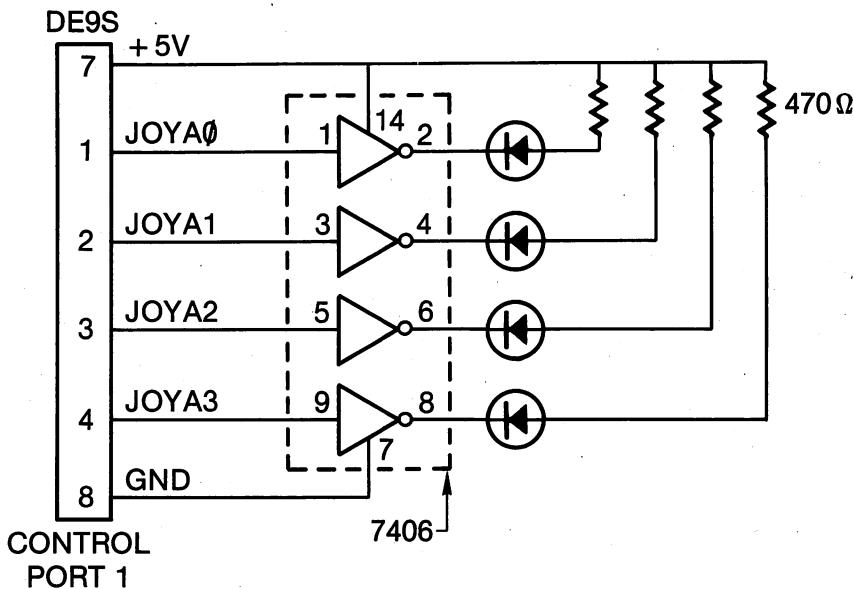


Figure 12-10. A four-bit output port driving light-emitting diodes.

The program in Example 12-5 configures the four low bits of Port B to be output bits and the four low bits of Port A to be input bits. The remainder of the program is a loop in which the data from Port A is transferred to Port B. Wire the circuit, plug a joystick into Control Port 2, and execute the program in Example 12-5. Move the joystick around and the glowing LEDs will tell you which joystick switches are closed.

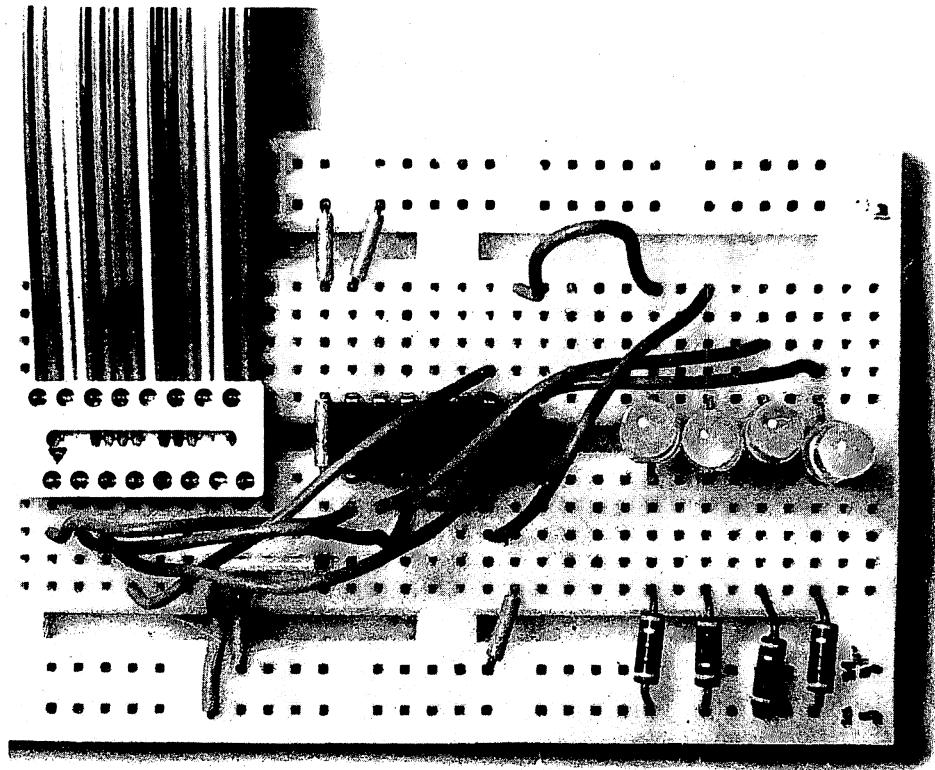


Figure 12-11. Photograph of the four-bit output circuit.

Example 12-5. Transferring Data From One Port to Another

Object: Transfer the four least-significant bits from Port A to Port B on CIA #1.

10	SEI	;DISABLE INTERRUPTS.	C000 78
11	LDA #\$F0	;CLEAR BITS 0-3 OF DDRA TO	C001 A9 F0
12	AND DDRA	;MAKE PINS PA0-PA3 INPUT PINS.	C003 2D 02 DC
13	STA DDRA		C006 8D 02 DC
14	LDA #\$0F	;SET BIT 0-3 OF DDRB TO	C009 A9 0F
15	ORA DDRB	;MAKE PINS PB0-PB3 OUTPUT PINS.	C00B 0D 03 DC
16	STA DDRB		C00E 8D 03 DC
17 LOOP	LDA PRA	;READ PORT A.	C011 AD 00 DC
18	STA PRB	;OUTPUT THIS RESULT TO PORT B.	C014 8D 01 DC
19	JMP LOOP	;STAY IN INFINITE LOOP.	C017 4C 11 C0

This concludes our brief introduction to input/output operations. Of course, there are more exciting and useful things to control than light-emitting diodes, for example, relays and solid-state switches. At the end of this chapter we provide references that provide more information about devices that you can connect to your computer. Our purpose has been fulfilled, since you now know how to perform input and output operations in assembly language.

It might be worth adding that a much more convenient but expensive way of experimenting with input switches and output LEDs is with the LR-25 outboard pictured in Figure 12-12. The outputs from a CIA can be connected directly to the LEDs on the LR-25 outboard with no other driving circuits required. The board has a set of switches that can be connected directly to input pins on the CIA. This equipment is available from:

E & L Instruments
61 First Street
Derby, CT 06418

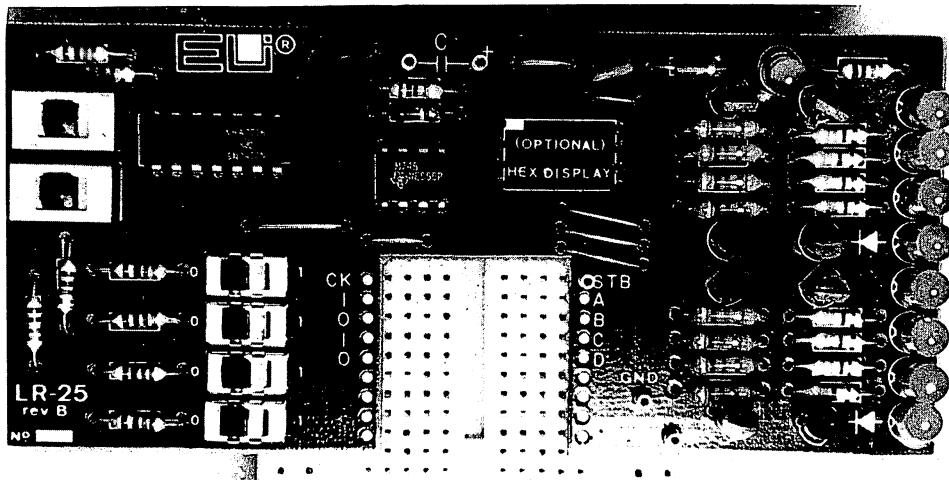


Figure 12-12. Photograph of the E & L Instruments LR-25 outboard connected to a Super Strip.

Now we will turn our attention to the timing abilities of the 6526 CIA.

IV. Counting and Timing with the 6526 CIA

The focus of this section will be a precision timing program that you can use either to measure time intervals between events or to measure the frequency of a pulse train. We begin with fundamental concepts.

A. Counting/timing fundamentals

In its simplest form, a *counter/timer* consists of a register in an integrated circuit. The register occupies one or more memory locations in the memory map of the microcomputer. The counter/timer is *loaded* by storing a number in the register. STA or POKE instructions may be used to load a counter/timer. When the counter/timer is *started*, the number initially loaded into the register is *decremented* at a regular rate by the microcomputer system clock or by an external source of pulses applied to a pin on the counter/timer integrated

circuit. When the number decrements *through zero*, a bit in another register is *set* or an interrupt is requested. The event "counting through zero" is called an *underflow*. The bit that signals the underflow is commonly called a *flag*.

A counter/timer is, among other things, capable of precisely controlling the length of the time interval between the instant the counter/timer is started and the instant it underflows. In assembly-language programming, the counter/timer frequently replaces delay loops produced by software and the considerable difficulty associated with producing a specified delay. Moreover, while the counter/timer is counting down toward underflow, the microprocessor can be doing a variety of other tasks, whereas in a delay loop it can only perform the instructions in the loop. The Commodore 64 operating system uses a counter/timer to produce an interrupt every 60th of a second. At this time, the keyboard is scanned to see if new information is present. A number of other tasks are also performed. In other words, almost all of the computing takes place between interrupts, while a counter/timer is performing its countdown. You can, in fact, speed up the execution of any program by disabling these interrupts. Use either an SEI instruction or stop the counter/timer that is producing the interrupts using a technique that we will describe shortly.

We will now look specifically at the counter/timers on the 6526 CIAs. A short memory map of the registers in the CIA #2 that will be of importance to us in counting/timing operations is given in Figure 12-13. Operating a

ADDRESS	READ	WRITE
\$DD04	TIMER A LATCH LOW	TIMER A COUNTER LOW
\$DD05	TIMER A LATCH HIGH	TIMER A COUNTER HIGH
\$DD06	TIMER B LATCH LOW	TIMER B COUNTER LOW
\$DD07	TIMER B LATCH HIGH	TIMER B COUNTER HIGH
\$DD0D	INTERRUPT CONTROL REGISTER FLAG BITS	INTERRUPT CONTROL REGISTER MASK BITS
\$DD0E	CONTROL REGISTER A	CONTROL REGISTER A
\$DD0F	CONTROL REGISTER B	CONTROL REGISTER B

Figure 12-13. Memory map of the counter/timer registers in complex interface adapter (CIA) #2.

counter/timer involves the use of two registers besides the counter/timer registers. Each of the two counter/timers has a *control* register that is used to select one of the several modes of operation that are available. The control registers are diagrammed in more detail in Figures 12-14 and 12-15. Bit zero in the control register is used to start the counter/timer.

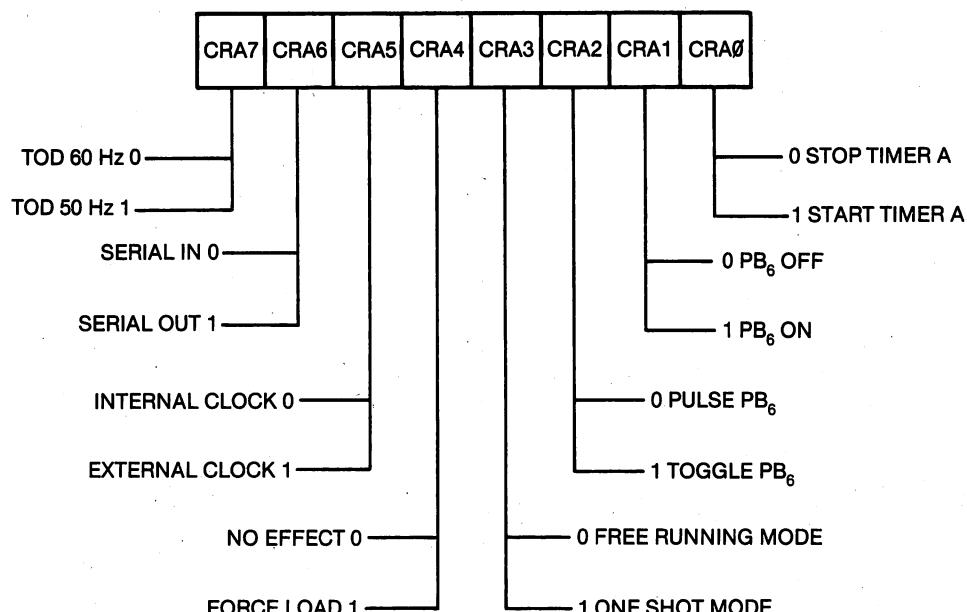


Figure 12-14. Bit structure of control register A (CRA) of the complex interface adapter (CIA).

A bit in another register, called the interrupt control register (ICR), is used to signal the underflow of a counter/timer. Each bit in the ICR is called a flag. The significance of each bit in the ICR, when it is *read*, is shown in Figure 12-16. Bit seven of the ICR is set when any of the other bits are set. All the flags are cleared when the ICR is read.

The CIA can be programmed so that an interrupt is requested with any flag setting. Refer to Figure 12-17. Writing to the interrupt control register (ICR) either enables or disables this interrupt request feature, using the following convention:

- If bit seven (ICR7) of the byte written to the ICR is one, any mask bit that has the value of one will *enable* the interrupt request feature.
- If bit seven of the byte written to the ICR is zero, any mask bit that has the value of one will *disable* the interrupt request feature.

It is important to realize that the interrupt request pin of CIA #2 is connected to the NMI pin on the 6510. Thus, an interrupt from CIA #2 will be a non-maskable interrupt (NMI). The interrupt request pin of CIA #1 is connected to the IRQ pin on the 6510. Interrupts were described in Chapter 9.

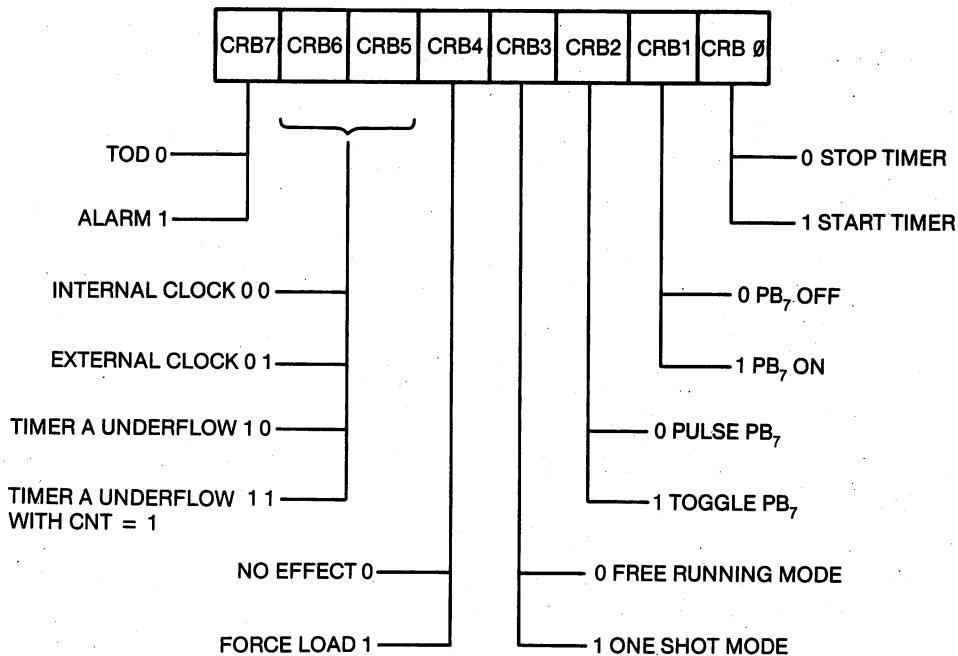


Figure 12-15. Bit structure of control register B (CRB) of the complex interface adapter (CIA).

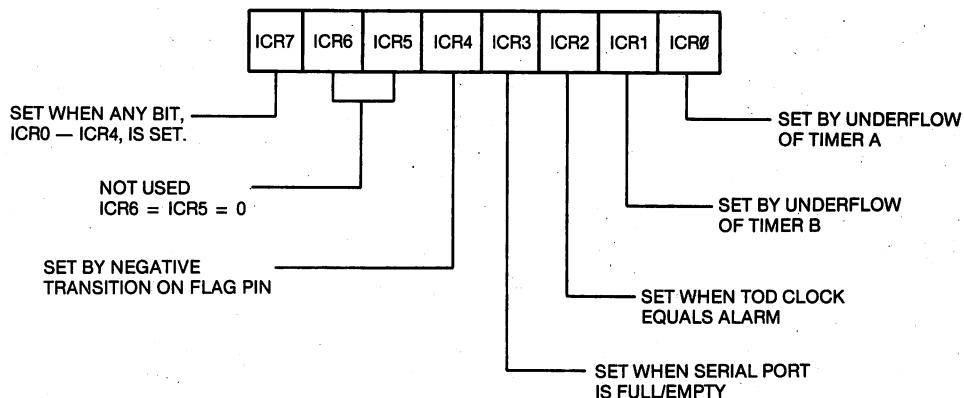


Figure 12-16. The flags in the interrupt control register (ICR). All the flags are cleared when the ICR is read.

Finally, there are the counter/timer registers themselves. Refer again to Figure 12-13. The 6526 CIA has two counter/timers called Timer A and Timer B. Just as in the case of the ICR, these registers have a different function, depending on whether the operation involves reading or writing. Each timer consists of a 16-bit register, called the timer *latch* or *prescaler*. The 16-bit latch consists of two eight-bit memory locations that can be loaded with two STA instructions. You can only write to the latch: you cannot read it. When a timer

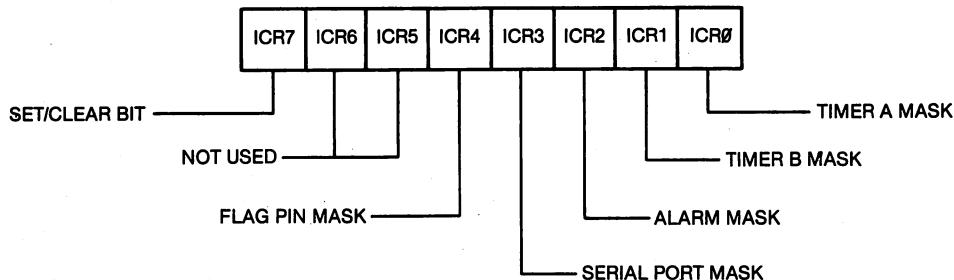


Figure 12-17. The mask bits in the interrupt control register (ICR). If bit seven (ICR7) of the byte written to the ICR is one, then any mask bit written with a one will enable an interrupt request on the IRQ pin. If bit seven (ICR7) of the byte written to the ICR is zero, then any mask bit written with a one will disable the interrupt request (IRQ).

is started, by setting bit zero of its control register, the number in the latch is automatically transferred to the 16-bit counter and the number in the counter begins to decrement.

When the memory locations corresponding to the latch are *read*, the number obtained is the value of the counter at that instant, *not* the number you wrote to the latch. In a sense, by reading the counters you can "watch" the timer/counter decrement. In fact, the counters are not usually read: the most common applications only require the reading of the flags in the interrupt control register (ICR) to see whether an underflow has occurred.

B. Counting/timing programming examples

It is time to illustrate some of these concepts with some simple programming examples. The program in Example 12-6 produces a delay of 50,000 clock cycles. We calculate the delay from the instant the timer is started on line 18 of the program to the instant the Timer A flag is set. If N is the number loaded into the timer, then this time interval T is given by the expression

$$T = (N + 1) * TC$$

where TC is the period of the system clock. In the case of the Commodore 64 clock, $TC = 0.977778$ microsecond.

Example 12-6. Producing a Delay of 50,000 Clock Cycles

Object: Configure Timer A to produce a one-shot delay of 50,000 clock cycles.

10	SEI	;DISABLE SYSTEM INTERRUPTS.	C000	78
11	LDA #\$7F	;DISABLE INTERRUPTS FROM CIA #2.	C001	A9 7F
12	STA ICR		C003	8D 0D DD
13	LDA #\$4F	;DELAY FOR 50000 CLOCK CYCLES.	C006	A9 4F
14	STA TALO	;SC34F + 1 = 50000.	C008	8D 04 DD
15	LDA #\$C3	;LOAD THE MSB OF THE TIMER.	C00B	A9 C3
16	STA TAHI		C00D	8D 05 DD
17	LDA #\$09	;PUT TIMER A IN ONE-SHOT MODE	C010	A9 09
18	STA CRA	;AND START TIMER A.	C012	8D 0E DD

19	LDA #\\$01	; MASK ALL BUT TIMER A FLAG.	C015 A9 01
20	WAIT	BIT ICR ; LOGICAL AND WITH ICR.	C017 2C 0D DD
21		BEQ WAIT ; WAIT FOR TIMER FLAG TO BE SET.	C01A F0 FB
22		CLI ; ENABLE INTERRUPTS.	C01C 58
23	RTS		C01D 60

Study the program in Example 12-6 in connection with Figures 12-13 to 12-16 and our line-by-line description that follows. On line 10 we disable the operating system interrupts. The next two lines write the byte \$7F to the interrupt control register (ICR). Study Figure 12-17 and observe that the effect of writing this byte to the ICR is to disable all interrupt requests from it. The instructions on lines 13 through 16 of the program in Example 12-6 load the latch of Timer A with the 16-bit number \$C37F. \$C37F + 1 = 50,000.

Lines 17 and 18 load Control Register A with the code \$09. Study Figure 12-14 and observe that this will start the timer in its *one-shot mode*. In the one-shot mode, the counter/timer counts down just *once*. In the *free-running mode*, the counters are automatically reloaded from the latches. We have selected the internal clock as our source of pulses to decrement the counter/timer. It is also possible to count the pulses from an external pulse source if CRA5 is set and a source of pulses is applied to the CNT pin on the CIA. Other operating modes for Timer A that may be selected by programming the CRA will be explored later.

An underflow of Timer A sets bit ICR0. On line 19 of the program in Example 12-6, we make a mask to isolate bit zero of the ICR. Lines 20 and 21 cause the program to wait in a loop until the underflow occurs, after which interrupts are enabled and the control returns to the calling program on line 23. This completes our line-by-line description of Example 12-6. The program will take slightly more than 50,000 clock cycles to execute; during most of that time, it is simply waiting for the Timer A flag, ICR0, to be set.

Our next programming example illustrates how the two timers can be coupled to generate long delays. In this case, we program Timer B to count underflows from Timer A that is operated in its free-running mode. Refer to Figure 12-15 and observe that Timer B can be programmed to count internal clock pulses, external clock pulses on the CNT pin, Timer A underflows, and Timer A underflows when the CNT pin is at logic one. In the program in Example 12-7, Timer B is programmed to count Timer A underflows. Assuming that Timer B has been started, the time T between the instant when Timer A is started and Timer B underflows is given by the formula

$$T = (NA + 1) * (NB + 1) * TC$$

where NA is the 16-bit number stored in the latch of Timer A and NB is the 16-bit number stored in the latch of timer B. The maximum time interval that can be produced in this way is 4,212 seconds, which is slightly more than one hour.

The program is listed in Example 12-7. We have chosen to produce underflows from Timer A every 50,000 clock cycles. We count a total of 200 of these underflows on Timer B before ending the program. This will produce

a delay of ten million clock cycles, which is approximately 10 seconds. Study this program and its comments in connection with Figures 12-14 to 12-16. In particular, notice that the loop in which we wait for Timer B to underflow tests bit one (ICR1), the Timer B flag, in the ICR.

Load the program and type in the SYS 49152 command. When you hit return, check your watch. Approximately 10 seconds later the Commodore 64 returns to BASIC with the READY prompt.

Example 12-7. Operating the Timers Together

Object: Produce a delay of 10 million clock cycles.

13	SEI	;DISABLE SYSTEM INTERRUPTS.	C000 78
14	LDA #\$7F	;DISABLE INTERRUPTS FROM CIA #2.	C001 A9 7F
15	STA ICR		C003 8D 0D DD
16	LDA #\$4F	;DELAY FOR 50000 CLOCK CYCLES.	C006 A9 4F
17	STA TALO	;\$C34F + 1 = 50000.	C008 8D 04 DD
18	LDA #\$C3	;LOAD THE MSB OF THE TIMER.	C00B A9 C3
19	STA TAHI		C00D 8D 05 DD
20	LDA #199	;TB COUNTS 200 TA UNDERFLOWS.	C010 A9 C7
21	STA TBLO		C012 8D 06 DD
22	LDA #00		C015 A9 00
23	STA TBHI		C017 8D 07 DD
24	LDA #49	;SET UP TB AND START IT.	C01A A9 49
25	STA CRB		C01C 8D 0F DD
26	LDA #01	;TIMER A IN FREE RUNNING MODE.	C01F A9 01
27	STA CRA	;START TIMER A.	C021 8D 0E DD
28	LDA #02	;MASK ALL BUT TIMER B FLAG.	C024 A9 02
29 WAIT	BIT ICR	;LOGICAL AND WITH ICR.	C026 2C 0D DD
30	BEQ WAIT	;WAIT FOR TIMER FLAG TO BE SET.	C029 F0 FB
31	CLI	;ENABLE INTERRUPTS.	C02B 58
32	RTS		C02C 60

Our next example involves counting external pulses on the CNT pin. The first requirement is a source of pulses, and the next requirement is a means for connecting the source of pulses to the CNT pin. This pin is available at the Commodore 64 user port. A pin diagram of this port is given in Figure 12-18. A photograph of the 12/24 edgeboard connector used to access the user port is shown in Figure 12-19. This connector is available from Priority One Electronics, which we mentioned above. A circuit diagram showing how to use a mechanical switch as a source of pulses is shown in Figure 12-20. (This circuit is duplicated on the E&L LR-25 outboard, mentioned earlier.) A positive pulse occurs on pin six of the 7400 (or 74LS00) whenever the double-pole single-throw (DPST) switch is changed from its NC (normally-closed) position to its NO (normally-open) position.

The program in Example 12-8 waits in a loop until nine positive pulses are applied to the CNT pin, pin six, on the user port. Study the program in connection with the diagram of the CRB and the ICR. Connect the circuit shown in Figure 12-20. Load and execute the program. Press the button that produces a positive pulse *nine* times, and program control will return to BASIC.

12/24 PIN EDGE CONNECTOR

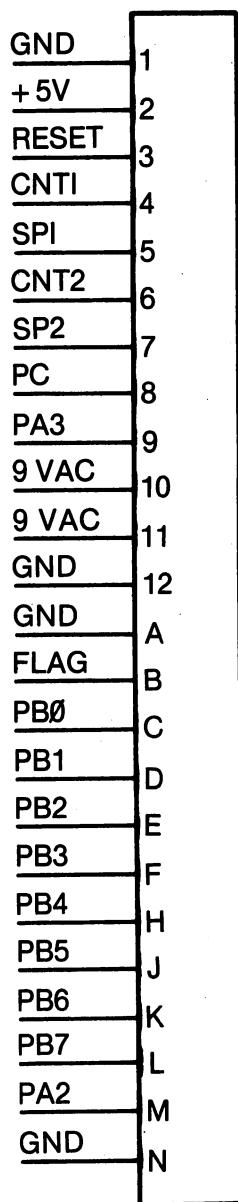


Figure 12-18. Pin diagram of the Commodore 64 user port.

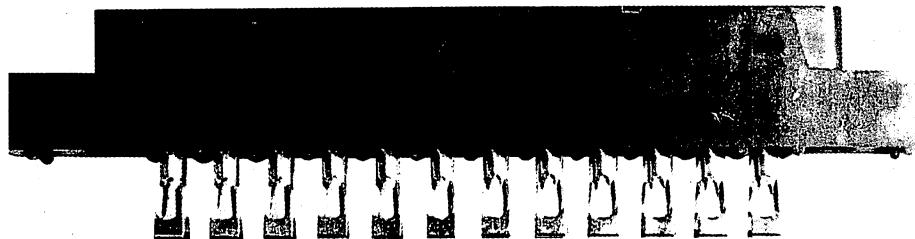


Figure 12-19. Photograph of the edge connector to access the user port.

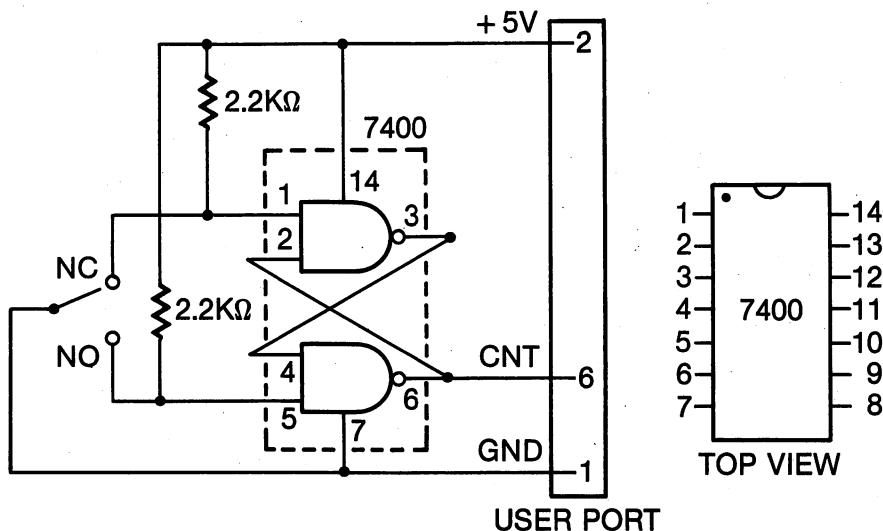


Figure 12-20. Diagram of a circuit to produce pulses using a mechanical switch. The switch is a STDP (single-throw, double-pole) type.

Example 12-8. Counting External Pulses

Object: Wait in a loop until nine positive pulses are applied to the CNT pin on CIA #2.

<pre> 10 SEI ;DISABLE SYSTEM INTERRUPTS. 11 LDA #\$7F ;DISABLE INTERRUPTS FROM CIA #2. 12 STA ICR 13 LDA #08 ;TB COUNTS 8 PULSES ON 14 STA TBLO ;THE CNT PIN. 15 LDA #00 16 STA TBHI 17 LDA #\$29 ;SET UP TB AND START IT. 18 STA CRB 19 LDA #\$02 ;MASK ALL BUT TIMER B FLAG. 20 WAIT BIT ICR ;LOGICAL AND WITH ICR. 21 BEQ WAIT ;WAIT FOR TIMER FLAG TO BE SET. 22 CLI ;ENABLE INTERRUPTS. 23 RTS </pre>	<pre> C000 78 C001 A9 7F C003 8D 0D DD C006 A9 08 C008 8D 06 DD C00B A9 00 C00D 8D 07 DD C010 A9 29 C012 8D 0F DD C015 A9 02 C017 2C 0D DD C01A F0 FB C01C 58 C01D 60 </pre>
--	--

C. A counting/timing application

We conclude this section on the 6526 CIA counter/timers with a versatile counting/timing program. The program can be used to measure the time T it takes for N pulses to occur on the CNT pin of the CIA. Refer to Figure 12-21 for a diagram of the T and N parameters. The pulse train pictured in Figure 12-21 can have either a square waveform or a rectangular waveform. Although we have pictured periodic waveforms, the pulses can arrive at random intervals, as they do in the case of counting radioactive events or people moving through turnstiles. In the case of a periodic waveform, the parameter of interest is the *frequency* of the pulses, that is, at what rate do they occur. If the number of pulses occurring in a time T is N , then the frequency f is given by the formula

$$f = N/T$$

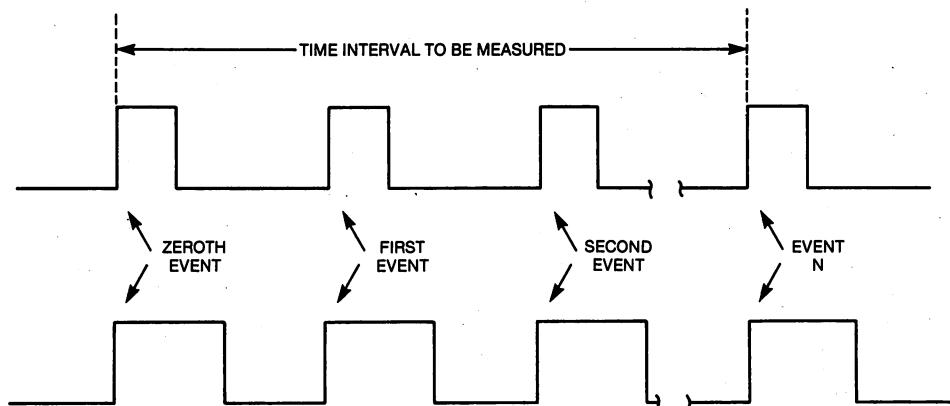


Figure 12-21. Timing diagram for the versatile counting/timing program in Example 12-9.

Frequency counting has many scientific and engineering applications. With voltage-to-frequency (V/F) and temperature (T/F) converters, a frequency-counting program can be used to measure voltage and temperature.

Refer again to Figure 12-21. If $N = 1$, then the time T between *two* pulses is obtained. The switch and the circuit shown in Figure 12-21 produce a positive pulse each time the switch is changed from its NC to its NO position. Thus, the switch and the program can be used to make a stopwatch.

Another circuit that may be used with this program is shown in Figure 12-22. A positive pulse on the CNT pin occurs whenever the light to the phototransistor is interrupted. Thus, you can measure the time between successive interruptions of the light shining on the phototransistor. For example, if a pendulum is swinging back and forth between the light source and the phototransistor, then, with $N = 2$, the program will measure the period of the pendulum. For another example, if two flags are mounted a distance L apart on

a moving object, then successive interruption of the light beam by the flags allow you to measure the time T for the object to move a distance L . Since the speed of the object is L/T , the circuit in Figure 12-22 can be used to measure the speed of an object.

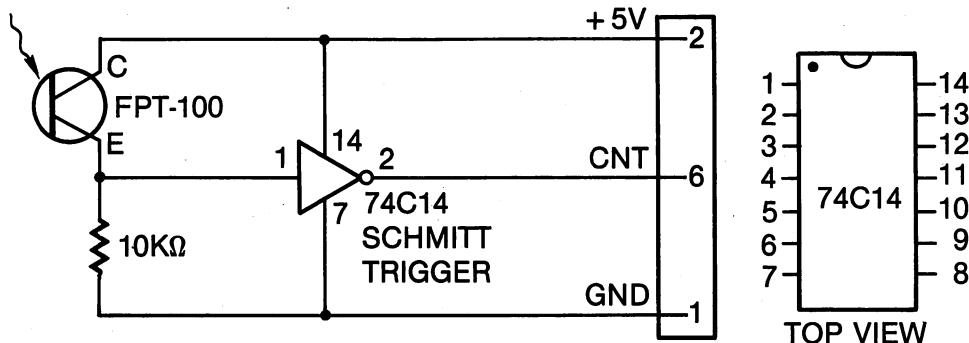


Figure 12-22. Diagram of a circuit that produces a positive pulse each time the light to the phototransistor is interrupted. The FPT-100 phototransistor is available from Radio Shack stores.

The assembly-language program to make these measurements is listed in Example 12-9. Here is how the program works. \$0001 is stored in Timer B, and it is operated in its external pulse-counting mode. In this mode, it counts pulses on the CNT pin accessed at the user port. The program loops until the first pulse occurs and the counter decrements to \$0000.

At this instant, Timer A is started in its free-running mode. The IRQ interrupts from Timer A are also enabled. Timer A counts system clock pulses that occur at a precise and constant rate, namely, 1.022727 MHz. An interrupt occurs every 65,536 clock cycles. Each time an interrupt occurs, a two-byte counter is incremented in the interrupt routine. Thus, the number of interrupts, plus the information in the Timer A registers, determine the time that has elapsed since the first pulse occurred on the CNT pin.

The number of pulses to be counted, N , is loaded into Timer B. When it decrements to zero, N pulses have been counted and both timers are stopped. The number of clock cycles since Timer A was last started is stored in memory, as well as the total number of interrupts.

Study the program and the comments in Example 12-9 in connection with this discussion and you will see how the program works.

The best way to use the program in Example 12-9 is to call it from a BASIC program that performs some of the arithmetic needed to convert clock cycles into a time expressed in seconds. The program in Example 12-10 illustrates how this is accomplished. On line 20 you enter the number of events to be counted. If you are performing a stopwatch-type experiment, then $N = 1$. The timer is started with the zero-eth event and stopped with the first event. If you wish to measure the time interval associated with 2,000 pulses, then enter 2,000 for N . The number N is converted into two bytes and POKE'd into

memory locations for the assembly language subroutine. The routine in Example 12-9 is called on line 50 of Example 12-10. The time calculations are accomplished by lines 60 through 100, and the result is printed on line 110.

Example 12-9. A Precision Timing/Counting Program

Object: Use Timer A to measure the time T it takes for N pulses to occur on Timer B.

```

1      ;EXAMPLE 12-9
2  CRA    EQU $DC0E          DC0E
3  TALO   EQU $DC04          DC04
4  TAHI   EQU $DC05          DC05
5  ICR1   EQU $DC0D          DC0D
6  ICR2   EQU $DD0D          DD0D
7  CRB    EQU $DD0F          DD0F
8  TBLO   EQU $DD06          DD06
9  TBHI   EQU $DD07          DD07
10 TIME   EQU $FD            00FD
11 NBLO   EQU $FB            00FB
12 NBHI   EQU $FC            00FC
13 EQU $C000          C000

14      ;
15      ;IRQ INTERRUPT ROUTINE.
16 IRQRTN INC +TIME          C000 E6 FD
17      BNE BR1             C002 D0 02
18      INC +TIME+1          C004 E6 FE
19 BR1    LDA ICR1          ;CLEAR IRQ FLAG.
20      PLA                ;RESTORE REGISTERS.
21      TAY                C006 AD 0D DC
22      PLA                C009 68
23      TAX                C00A A8
24      PLA                C00B 68
25      RTI                C00C AA
26      ;
27      ;INITIALIZATION ROUTINE.
28      SEI                ;DISABLE SYSTEM INTERRUPTS.
29      LDA #$00            ;STOP TIMERS.
30      STA CRA             C00F 78
31      STA CRB             C010 A9 00
32      STA $0314           ;SET UP IRQ VECTOR.
33      LDA #$C0             C012 8D 0E DC
34      STA $0315           C015 8D 0F DD
35      LDA #$81            ;ENABLE INTERRUPTS FROM TIMER A.
36      STA ICR1             C018 8D 14 03
37      LDA #$FF            ;PROVIDE INTERRUPTS EVERY
38      STA TALO             C01B A9 C0
39      STA TAHI             C01D 8D 15 03
40      LDA #00              C020 A9 81
41      STA TBHI             C022 8D 0D DC
42      STA +TIME             C025 A9 FF
43      STA +TIME+1          ;CLEAR TWO BYTE INTERRUPT
44      LDX #$01            ;COUNTER.
45      STX TBLO             C027 8D 04 DC
46      LDY #$31            ;BYTE FOR CRA TO START TIMER A.
47      CLI                C02A 8D 05 DC
48      ;
49      STY CRB             ;START TIMER B.
50 WAIT   LDA TBLO           C02D A9 00
51      BNE WAIT            ;HAS FIRST PULSE OCCURRED?
52      ;NO, WAIT FOR THE FIRST COUNT.
53      STX CRA             C02F 8D 07 DD
54      STA CRB             C032 85 FD
55      STA TBLO             C034 85 FE
56      LDA +NBLO            C036 A2 01
57      STA TBHI             C038 8E 06 DD
58      CLI                C03B A0 31
59      ;
60      STY CRB             ;RELOAD TIMER B.
61      ;RESTART TIMER B.
62      ;RESTART TIMER B.
63      ;RESTART TIMER B.
64      ;RESTART TIMER B.
65      ;RESTART TIMER B.
66      ;RESTART TIMER B.
67      ;RESTART TIMER B.
68      ;RESTART TIMER B.
69      ;RESTART TIMER B.
70      ;RESTART TIMER B.
71      ;RESTART TIMER B.
72      ;RESTART TIMER B.
73      ;RESTART TIMER B.
74      ;RESTART TIMER B.
75      ;RESTART TIMER B.
76      ;RESTART TIMER B.
77      ;RESTART TIMER B.
78      ;RESTART TIMER B.
79      ;RESTART TIMER B.
80      ;RESTART TIMER B.
81      ;RESTART TIMER B.
82      ;RESTART TIMER B.
83      ;RESTART TIMER B.
84      ;RESTART TIMER B.
85      ;RESTART TIMER B.
86      ;RESTART TIMER B.
87      ;RESTART TIMER B.
88      ;RESTART TIMER B.
89      ;RESTART TIMER B.
90      ;RESTART TIMER B.
91      ;RESTART TIMER B.
92      ;RESTART TIMER B.
93      ;RESTART TIMER B.
94      ;RESTART TIMER B.
95      ;RESTART TIMER B.
96      ;RESTART TIMER B.
97      ;RESTART TIMER B.
98      ;RESTART TIMER B.
99      ;RESTART TIMER B.
100     ;RESTART TIMER B.
101     ;RESTART TIMER B.
102     ;RESTART TIMER B.
103     ;RESTART TIMER B.
104     ;RESTART TIMER B.
105     ;RESTART TIMER B.
106     ;RESTART TIMER B.
107     ;RESTART TIMER B.
108     ;RESTART TIMER B.
109     ;RESTART TIMER B.
110     ;RESTART TIMER B.

```

```

59 ;
60 BR2    LDA TBHI      ;WAIT FOR TIMER TO REACH ZERO.      C059 AD 07 DD
61       BNE BR2
62 BR3    LDA TBLO      ;WAIT FOR TIMER TO REACH ZERO.      C05C D0 FB
63       BNE BR3
64       STA CRA        ;STOP TIMER A.                  C05E AD 06 DD
65       STA CRB        ;STOP TIMER B.                  C061 D0 FB
66       LDA TALO        ;READ TIMER A LOW BYTE.      C063 8D 0E DC
67       EOR #$FF        ;SUBTRACT FROM $FF.      C066 8D 0F DD
68       STA +NBLO        ;STORE IT.                  C069 AD 04 DC
69       LDA TAHI        ;READ TIMER A HIGH BYTE.      C06C 49 FF
70       EOR #$FF        ;SUBTRACT FROM $FF.      C06E 85 FB
71       STA +NBHI        ;STORE IT.                  C070 AD 05 DC
72       RTS

```

Example 12-10. A BASIC Routine to Call the Subroutine in Example 12-9

```

0 REM EXAMPLE 12-10
5 REM PRECISION COUNTING/TIMING PROGRAM
10 PRINT "INPUT THE NUMBER OF EVENTS."
15 PRINT "THIS NUMBER CANNOT EXCEED 65535."
20 INPUT N
30 NHI=N/256:NLO=NAND255
40 POKE 251,NLO:POKE 252,NHI
50 SYS 49167
60 A=PEEK(251)+256*PEEK(252)
70 B=65536*PEEK(253)+A
80 B=B/1022727
90 D=16777216/1022727*PEEK(254)
100 T=D+B
110 PRINT "THE TIME IS ";T;" SECONDS."
120 GO TO 40

```

If you are measuring frequency, the program in Example 12-10 should be modified slightly. We have illustrated this in Example 12-11. Suppose you are measuring a frequency of about 1,000 Hz. If you enter a number near 1,000, then you will get a reading about once every second. The program will print the frequency and return to measure it again.

Example 12-11. A BASIC Program to Measure Frequency Using the Routine in Example 12-9

```

0 REM EXAMPLE 12-11
5 REM FREQUENCY COUNTING PROGRAM
10 PRINT "INPUT THE NUMBER OF CYCLES"
20 PRINT "NOT TO EXCEED 65535."
30 INPUT N
40 NHI=N/256:NLO=N AND 255
50 POKE 251, NLO:POKE 252, NHI
60 SYS 49167
70 A=PEEK(251)+256*PEEK(252)
80 B=65536*PEEK(253) + A
85 B=B/1022727
90 D=16777216/1022727*PEEK(254)
100 F=N/(B+D)
110 PRINT "THE FREQUENCY IS ";F;" HZ."
120 GO TO 50

```

Use the stopwatch circuit in Figure 12-20 to test Examples 12-9 and 12-10. To test the programs in Examples 12-9 and 12-11, use a square-wave function generator connected between a GND pin and the CNT pin on the user port. If you do not have a function generator, you can construct the simple square wave generator whose circuit is shown in Figure 12-23.

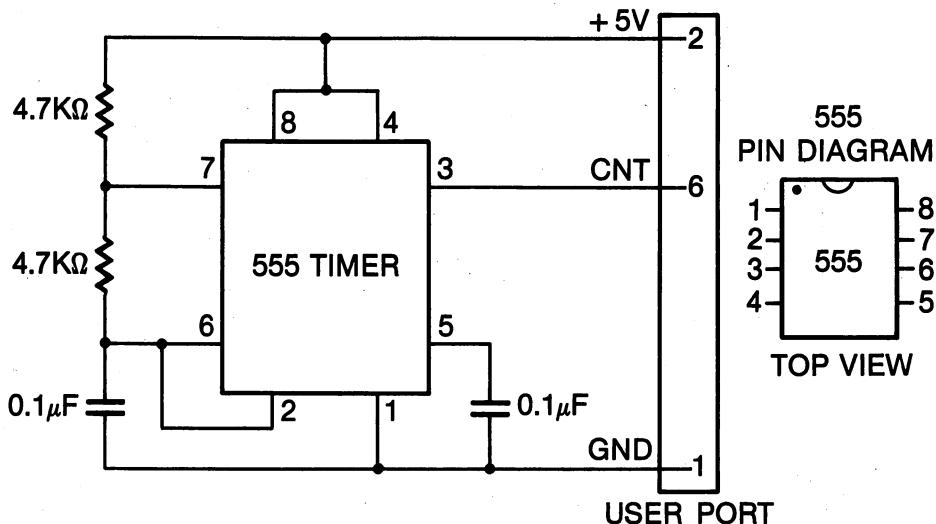


Figure 12-23. Diagram of a circuit to produce a square wave output at a frequency of approximately 1,000 Hz.

The maximum time interval that can be measured is approximately one hour. The uncertainty of any time measurement less than 65,535 clock cycles is approximately 10 clock cycles, while the uncertainty of the measurement is less than 100 clock cycles for time intervals longer than 65,535 clock cycles. The reason for the decreased precision for the longer intervals is that it is impossible to tell where in the interrupt routine the last pulse occurred, and the interrupt routine takes between 50 and 75 clock cycles. Unless you are working for the National Bureau of Standards, this program offers sufficient precision for most of your measurements. The maximum pulse rate that can be measured is approximately 50,000 Hz.

This completes our formal discussion of the 6526 CIA. We have already briefly discussed the time-of-day (TOD) clock in Chapter 7, and we will discuss it in more detail in the exercises.

If you are interested in expanding your knowledge of input/output functions and interfacing your computer to the outside world, here are some useful references. Although they do not refer specifically to the Commodore 64, these are useful books for anyone who has an interest in computer hardware. We have listed them in the order of their usefulness, according to our judgment.

Programming & Interfacing the 6502, With Experiments. Marvin L. De Jong. Indianapolis, Indiana: Howard W. Sams & Co., Inc., 1980.

Interfacing Microcomputers to the Real World. Murray Sargent III and Richard L. Shoemaker. Reading, Massachusetts: Addison-Wesley, 1981.

Programming a Microcomputer. Caxton C. Foster. Reading, Massachusetts: Addison-Wesley, 1978.

TRS-80 Interfacing Book 2. J.A. Titus, C.A. Titus, D.G. Larsen. Indianapolis, Indiana: Howard W. Sams & Co., Inc., 1980.

Microcomputer Analog Converter Software & Hardware Interfacing. J.A. Titus, C.A. Titus, D.G. Larsen. Indianapolis, Indiana: Howard W. Sams & Co., Inc., 1978.

Microcomputer Interfacing. Bruce A. Artwick. Englewood Cliffs, New Jersey: Prentice-Hall, 1980.

V. Summary

The 6526 complex interface adapter has two I/O ports, two counter/timers, a serial port, and a time-of-day timer. An input port is used to transfer information from the outside world to the microcomputer. An output port transfers information from the microcomputer to the outside world. These I/O ports typically consist of eight pins, allowing eight bits of information to be transferred simultaneously. As far as the programmer is concerned, an I/O port is simply another location in memory. This location is read for an input operation, and information is written to this location for an output operation.

Associated with each I/O port on the CIA is a data-direction register. The DDR is used to determine whether an I/O pin is going to be used for input or for output purposes. A one in a bit of the DDR makes the corresponding I/O pin an output pin; a zero in a bit of the DDR makes the corresponding I/O pin an input pin.

In almost all I/O operations, it will be necessary to have some knowledge about the hardware and electronic circuitry that is connected to the I/O port. Consult some of the references provided in the chapter to acquire this knowledge.

The 6526 CIA has two counter/timers. These integrated circuits are most frequently used to provide programmable delays without having to write software delay loops. A number is written to the counter/timer, after which it is decremented at a rate determined by the system clock or an external clock. When the counter/timer decrements through zero (underflows) a flag is set in a register to signal this event. The counter/timer can also be programmed to produce an interrupt when an underflow occurs. The counter/timers are also used to make precision time and frequency measurements.

VI. Exercises

The programming exercises will focus on the TOD clock on the 6526 CIA. Refer to Table 12-4 for the memory locations of the TOD clock. The time in

each register is a binary-coded decimal (BCD) number. Thus, in the tenths-of-seconds register it is only the low nibble that contains any information, which will be a number from zero to nine.

Table 12-4. Complex interface adapters time-of-day clock registers.

Register	Bits								Address	
	7	6	5	4	3	2	1	0	CIA #1	CIA#2
Tenths-of-seconds	0	0	0	0	T3	T2	T1	T0	\$DC08	\$DD08
Seconds	0	S6	S5	S4	S3	S2	S1	S0	\$DC09	\$DD09
Minutes	0	M6	M5	M4	M3	M2	M1	M0	\$DC0A	\$DD0A
Hours	PM	0	0	H4	H3	H2	H1	H0	\$DC0B	\$DD0B

When the number in the tenths-of-seconds register increments from nine to zero, the "carry" moves into the seconds register. Bits S0 through S3 identify the *units* of seconds, that is, the number of seconds from zero to nine. Bits S4 through S3 identify the *tens* of seconds, a digit from zero to five.

When the seconds register increments from 59 to 00, the carry moves into the minutes register. The minutes register works exactly like the seconds register. It counts to 59 and then a carry moves into the hours register.

The hours register counts from zero to 12. The PM flag, bit seven in the hours register, is set when the time moves from 11 59 59.9 to 00 00 00.0.

Two other bits in two other registers are important in controlling the TOD clock. To set the time, you write to the TOD registers described in Table 12-4 with CRB7 zero. Refer to Figure 12-12 for a diagram of the CRB register. To set the TOD clock alarm, you write to the TOD registers described in Table 12-4 with CRB7 one. The alarm "sounds" by setting ICR2, bit two in the interrupt control register (refer to Figure 12-16). Of course, you can also enable an interrupt with this alarm bit.

The TOD clock is started by writing to the tenths-of-seconds register. The approach for setting and starting the TOD clock was given in Example 7-7. Write the desired numbers to the hours, minutes, seconds, and tenths-of-seconds registers, in that order, with CRB7 = 0. To set the alarm, write the desired time to the hours, minutes, seconds, and tenths-of-seconds registers with, CRB7 = 1.

To read the clock, read the hours, minutes, seconds, and tenths-of-seconds, in that order. The clock latches when the hours register is read so that you do not miss a carry. The clock remains latched until the tenths-of-seconds register is read. The clock continues to run even when it is latched.

1. Begin by writing a subroutine to start the clock with zeros in all of the registers.
2. Next, write a subroutine to read the time and output it to the video monitor. Refer to Example 7-7 if you have difficulty. Extend your program to display the time continuously.

3. Write a routine to set the alarm. Remember, CRB7 must be one to write to the alarm registers, which have the same addresses as the clock registers. Use the CRB in the CIA you have chosen to use for the TOD clock.
4. Use the routines you have written to start the clock at 00 00 00.0 and set the alarm for 00 01 30.0 (that is, a minute and 30 seconds after the clock starts). Add a program segment that waits in a loop for the alarm to go off, and then returns to BASIC. You need to test the ICR2 bit in the interrupt-control register in the CIA you have chosen to use for the TOD clock. Test your programs by starting the clock and waiting for a minute and a half until the setting of the alarm bit allows your program to return to BASIC via an RTS instruction.
5. Here is how you can make a stopwatch using the TOD clock. Write a program segment that waits in a loop until one of the joystick bits is switched to zero with a joystick. Refer to the examples in this chapter to see how to read the joystick input bits.

When this joystick bit is zero, the program should continue, and it should set the TOD clock to 00 00 00.0 and start it.

Next, wait in another loop until another joystick input bit switches to zero, then read the clock and output the time using either the subroutine you wrote or Example 7-7.

6. Write a program to output the TOD clock time each time the joystick switches a bit to zero.
7. Your last exercise before you can call yourself an assembly-language programmer is to discover the only instruction we have not yet mentioned in this book and learn how to use it. We have left the simplest instruction until last.

A

Decimal, Binary, and Hexadecimal Number Systems

I. Introduction

The concept of *number* is as old as human culture, and different cultures have found different ways to express numbers. A specific scheme for using symbols to represent numbers is called a *numeration system*. Even within a culture, different numeration systems may be useful to express the same number. That is the case in our culture where we use a *base-ten place value* (decimal) numeration system for most applications, but we use a *base-two place value* (binary) numeration system for many computer applications.

By holding up various numbers of fingers, a human being can easily symbolize ten unique *states*. Each state symbolizes a counting number from one to ten. This fact is very likely the reason for our familiar base-ten number system. (With more complex symbolism, these same ten fingers can be used to symbolize many more numbers than ten.) The evolution of numeration has also given us ten unique numerals (or digits):

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

These ten unique symbols can be grouped together to express many more than ten numbers. Although you are familiar with using numbers in this way, it will be important to understand exactly *how* this scheme works. We will describe it in detail below.

The electronic components in a microcomputer are inherently *two-state* rather than ten-state devices. When a two-state device is used to express a number, only two numerals are available, and the numeration system is called a *base-two system* or a *binary numeration system*. When writing numbers in this system, it is customary to use the numerals "0" and "1" for zero and one, respectively.

Within the microcomputer itself, the numerals 0 and 1 have no meaning. The microprocessor and the other components of the microcomputer system cannot read numerals. However, engineers and computer scientists who design computer systems agree that a "0" will be represented by a *voltage* of zero volts and a "1" will be represented by a *voltage* of five volts. As far as the microcomputer is concerned, these voltages are numerals. Regardless of whether numbers are represented with two voltages or two written symbols ("0" and "1"), the numeration system is a base-two (binary) system. Since the microcomputer is built around a binary concept, and since you want to program it in its native binary language, it will be important for you to be proficient with binary numbers, and to have the ability to translate from decimal numbers to binary numbers.

The smallest unit of information in the microcomputer is the *binary digit*, which will represent either zero or one. The word "*bit*" is a contraction of the words "binary" and "digit." Thus, the basic unit of information in a computer is a bit. Here is an example of a bit:

1

Each memory location in the microcomputer stores eight bits. Thus, information is stored in the form of eight-bit binary codes (or numbers). An eight-bit code is called a *byte*. Here is an example of a byte:

00111010

Information is not only *stored* in eight-bit codes, essentially all of the operations of the microprocessor are *performed* eight bits at a time. The Commodore 64 is, therefore, an *eight-bit* microcomputer.

Eight-bit codes are difficult to remember and write. A much more convenient scheme is to use a *base-sixteen place value* numeration system, called a *hexadecimal* system. A base-sixteen system requires 16 numerals. We will use the numerals 0 through 9 for the first ten hexadecimal numerals, and the letters A, B, C, D, E, and F for the remaining six hexadecimal numerals. As you will see, in the hexadecimal system each hexadecimal numeral represents four bits of binary data. Thus, each byte of information stored in the computer can be represented with two hexadecimal numerals. (Throughout this appendix, we will refer to these as hexadecimal digits rather than hexadecimal numerals). The byte we illustrated above in binary form is expressed as

\$3A

in hexadecimal. The "\$" prefix is traditionally used to indicate a hexadecimal number.

The Commodore 64 microcomputer and many other microcomputers are designed so that each memory location is identified by a binary number with 16 binary digits (a 16-bit binary number). This number is called the *address* of the memory location. Here is an example of a 16-bit address:

1001101010111100

It is inefficient to express a number in its 16-bit binary form. It is much more efficient and convenient to use four hexadecimal digits to identify the address of a memory location. The 16-bit address listed above is represented by

\$9ABC

in hexadecimal. In decimal, this address is the number

39612

It will be important for you to learn to move easily between the three representations of a number, decimal (base ten), binary (base two), and hexadecimal (base sixteen). The purpose of this appendix is to help you on your way.

Before studying numbers in more detail, it is important to realize that numbers are used in at least three different ways:

- Numbers are used to indicate *quantity*.
- Numbers are used to indicate *order*.
- Numbers are used as *codes* for purposes of identification.

The use of a number to express *quantity* is familiar to all of us. How many pages are there in this book? The number 65,536 indicates the quantity of memory locations in your microcomputer. The houses in a block are *ordered* by their addresses. The memory locations in a microcomputer are also ordered by their addresses. You are identified by your social security number, a nine-digit *code*. Postal zip codes and telephone numbers are also used as identification codes. In some cases, a number serves several purposes. Your house address not only identifies your house, it orders it on the street where you live. Microcomputers use numbers in each of the three ways just described.

II. Decimal Numbers

Decimal numbers are symbolized by a sequence of digits. It will be useful to dissect a decimal number. Such a dissection is shown in Figure A-1. Each digit has a *face value* and a *place value*. Face values run from zero to nine, and these are learned at an early age.

The place value of a digit is determined by the *position* of the digit in the sequence of digits. The rightmost digit, called the least-significant digit, has a place value of one. The next digit to the left has a place value of ten, the next has a place value of one hundred, and so on. Thus, the place values show the following pattern:

10000, 1000, 100, 10, 1

which may also be written as

$10^4, 10^3, 10^2, 10^1, 10^0$

If each digit in the sequence is given a *digit number*, starting with zero for the digit on the right and counting upward, then the place value is the *base* raised to a power (exponent) equal to the digit number. Thus, the least-significant

digit, the 0th digit, has a place value of 10^0 . Digit one has a place value of 10^1 , and so on. Thus, the number 2,598 may be written in an expanded form as follows:

$$2598 = 2 \cdot 10^3 + 5 \cdot 10^2 + 9 \cdot 10^1 + 8 \cdot 10^0$$

This shows that the value of the number is found by multiplying the face value of each digit by its place value and then adding these results. How would you expand the decimal number 54231?

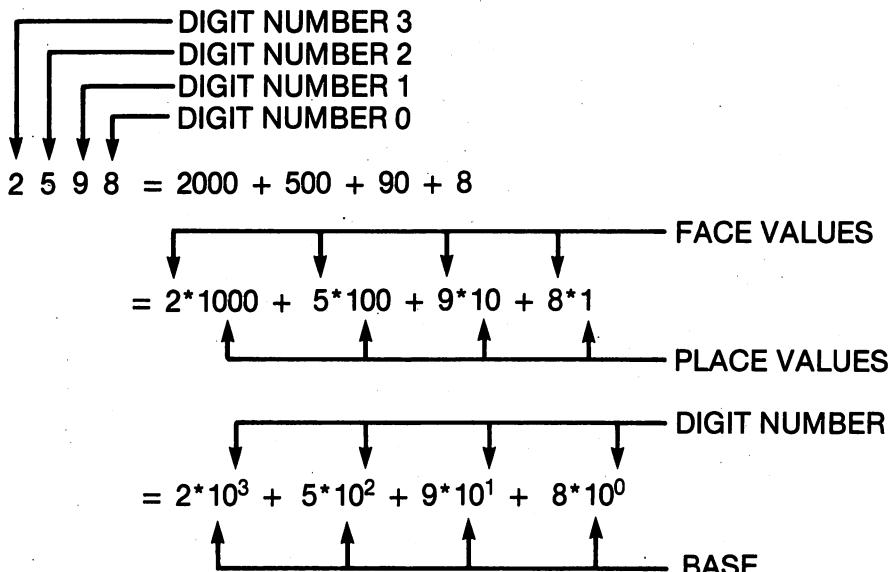


Figure A-1. Dissection of the decimal number 2,598.

III. Binary Numbers

The essential difference between binary numbers and decimal numbers is in the base. Binary numbers have a base of two. Only two face values are allowed, namely, zero and one. The base of a number other than a decimal number is sometimes indicated by a subscript. Thus,

$$1011_{\text{two}}$$

is a base two or binary number. This particular binary number is dissected in Figure A-2. In the binary numeration system, the place values are $2^0, 2^1, 2^2, 2^3$ and so on, corresponding to bit numbers 0, 1, 2, and 3. Compare Figures A-1 and A-2. A brief table of powers of two is useful; one is given in Table A-1.

A binary number can also be written in an expanded form. Thus,

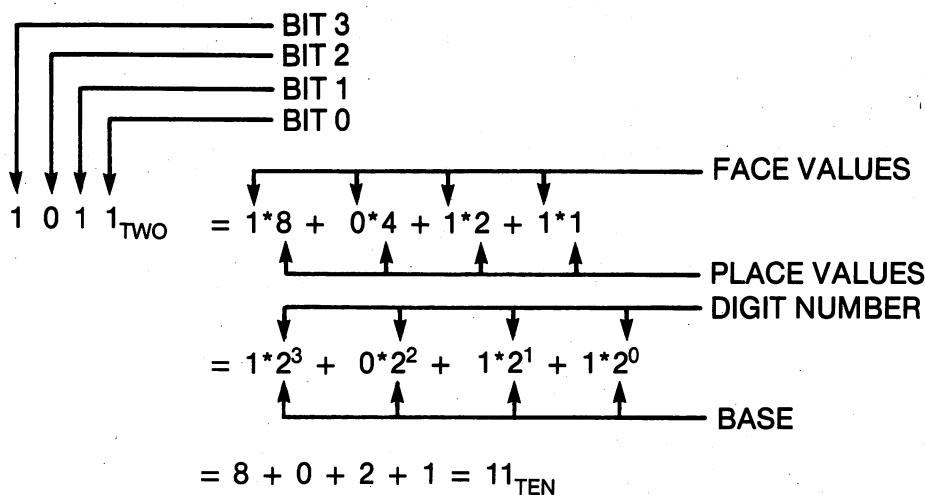
$$1011_{\text{two}} = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

or

$$1011_{\text{two}} = 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 11_{\text{ten}}$$

Table A-1. Powers of 2.

n	2^n	n	2^n
0	1	9	512
1	2	10	1024
2	4	11	2048
3	8	12	4096
4	16	13	8192
5	32	14	16384
6	64	15	32768
7	128	16	65536
8	256	17	131072

**Figure A-2. Dissection of the binary number 1011.**

If it is clear from the context of the discussion that the base is two, then the subscript "two" is dropped. Assembly-language programmers frequently use a percent sign (%) rather than a subscript to designate binary numbers. Thus,

$$1101_{\text{two}} = \%1101$$

The expanded form of a binary number that we have just described and the diagram in Figure A-2 suggest a technique for converting a binary number to decimal. Multiply the face value (0 or 1) of each bit by its place value and add the results. Thus,

$$\begin{aligned} \%11000110 &= 1 * 128 + 1 * 64 + 0 * 32 + 0 * 16 + 0 * 8 + 0 * 4 + 1 * 2 + 0 * 1 \\ &= 128 + 64 + 4 + 2 \\ &= 198_{\text{ten}} \end{aligned}$$

It is simpler to organize this work starting with the least-significant bit. Thus,

$$\%11000110 = 0 * 1 + 1 * 2 + 1 * 4 + 0 * 8 + 0 * 16 + 0 * 32 + 1 * 64 + 1 * 128$$

There are several techniques for converting a decimal number to a binary number. Here is the one that I like:

- Use Table A-1 to find the largest power of two that will divide the given decimal number. Place a one in the corresponding bit position.
- Divide the decimal number by this power of two and note the remainder.
- Divide the remainder of the first result by the next largest power of two. Put the dividend, 0 or 1, in the next lower bit position.
- Continue dividing by successively smaller powers of two until you finish by dividing by one.

Example A-1 illustrates this process.

Example A-1. Convert 207 to a Binary Number

Solution: Refer to Table A-1 to find that the largest power of two that will divide 207 is 2^7 , or 128. Thus, bit seven will have a one in it. Now perform successive divisions as follows:

$$\begin{array}{r} \overline{1} & \overline{1} & \overline{0} & \overline{0} & \overline{1} & \overline{1} \\ 128 \overline{)207} & 64 \overline{)79} & 32 \overline{)15} & 16 \overline{)15} & 8 \overline{)15} & 4 \overline{)7} \\ \underline{128} & \underline{64} & \underline{0} & \underline{0} & \underline{8} & \underline{4} \\ 79 & 15 & 15 & 15 & 7 & 3 \\ \end{array}$$

Thus, $207 = \%11001111$.

When numbers are used to indicate *quantity*, they do not ordinarily have leading zeros. You would not say you bought 0025 pounds of nails; instead, you would drop the leading zeros and say you bought 25 pounds of nails. When numbers are used as *codes*, they may have leading zeros. Postal zip codes, for example, have leading zeros. All of the codes in a microcomputer are of a fixed length, eight bits, and they may have leading zeros. If an eight-bit code is interpreted as a number, representing a quantity, then the leading zeros are disregarded.

As we have pointed out, the fixed length of the codes used by the microcomputer is eight bits or one byte. To identify memory locations, the microcomputer uses a fixed-length 16-bit code called the *address* of the memory location. The 16-bit address is frequently described in terms of two bytes. The least-significant byte is called the low-order byte of the address, or *address low* (ADL). The most-significant byte is called the high-order byte of the address, or *address high* (ADH). For example, in the 16-bit address

1100001100111100

the byte

11000011

is the ADH of the address, and the byte

00111100

is the ADL of the address.

A four-bit binary number is sometimes called a *nibble*. This is a useful idea when translating from binary to hexadecimal. A byte may be divided into a most-significant nibble (high-order nibble) and a least-significant nibble (low-order nibble). The byte

11110000

has a high-order nibble that is

1111

and a low-order nibble which is

0000

Some writers prefer to spell nibble as nybble.

IV. Hexadecimal Numbers

The sixteen hexadecimal numerals and their binary and decimal equivalents are listed in Table A-2. With continued use of hexadecimal and binary numbers, this table will eventually be committed to your memory. The important thing to notice in Table A-2 is that four bits of information are represented with one hexadecimal numeral. Thus, a byte is described with two hexadecimal numerals and a 16-bit address is described with four hexadecimal numerals.

Table A-2. Hexadecimal, binary, and decimal equivalents.

<i>Hexadecimal Number (Base Sixteen)</i>	<i>Binary Number (Base Two)</i>	<i>Decimal Number (Base Ten)</i>
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15
10	10000	16

Converting from binary to hexadecimal proceeds as follows:

- Group the binary number into nibbles.
- Using the table or your memory, convert each nibble into a hexadecimal digit.

For example,

$$01011010 = 0101\ 1010 = \$5A$$

and

$$1111111011011100 = 1111\ 1110\ 1101\ 1100 = \$FEDC$$

To convert from hexadecimal to binary, reverse the process. Thus,

$$\$6E = 0110\ 1110 = 01101110$$

and

$$\$89AB = 1000\ 1001\ 1010\ 1011 = 1000100110101011$$

To convert from hexadecimal to decimal, expand the hexadecimal number as follows:

$$\$6E = 6*16^1 + 14*16^0 = 6*16 + 14*1 = 110$$

or

$$\$89AB = 8*16^3 + 9*16^2 + 10*16^1 + 11*16^0$$

giving

$$\$89AB = 8*4096 + 9*256 + 10*16 + 11*1 = 35243$$

To convert from decimal to hexadecimal, divide the decimal number by 16 and note the remainder. Divide the quotient of that result by 16 and note the remainder. Continue until the quotient is zero. The first remainder becomes the least-significant hexadecimal digit, the last remainder becomes the most-significant hexadecimal digit. Example A-2 illustrates the process.

Example A-2. Convert 45678 to Hexadecimal

Solution:

$$\begin{array}{r} 2854 \\ 16 \overline{)45678} \\ 4564 \\ \hline 14 \ (\$E) \end{array} \quad \begin{array}{r} 178 \\ 16 \overline{)2854} \\ 2848 \\ \hline 6 \end{array} \quad \begin{array}{r} 11 \\ 16 \overline{)178} \\ 176 \\ \hline 2 \end{array} \quad \begin{array}{r} 0 \\ 16 \overline{)11} \\ 0 \\ \hline 11 \ (\$B) \end{array}$$

Therefore, 45678 = \$B26E

Some useful patterns between binary, hexadecimal, and decimal numbers may emerge in your mind if you study Table A-3. In this table, we have listed a variety of binary numbers and their hexadecimal and decimal equivalents.

Table A-3. Some binary, hexadecimal and decimal numbers.

<i>Binary Number</i>		<i>Hexadecimal Number</i>	<i>Decimal Number</i>
0000	Nibble	\$00	0
0001	Nibble	\$01	1
0010	Nibble	\$02	2
0011	Nibble	\$03	3
1000	Nibble	\$08	8
1001	Nibble	\$09	9
1010	Nibble	\$0A	10
1011	Nibble	\$0B	11
1110	Nibble	\$0E	14
1111	Nibble	\$0F	15
0001 0000	Byte	\$10	16
0010 0000	Byte	\$20	32
0100 0000	Byte	\$40	64
1000 0000	Byte	\$80	128
1000 1000	Byte	\$88	136
1010 0000	Byte	\$A0	160
1100 0000	Byte	\$C0	192
1110 0000	Byte	\$E0	224
1111 1111	Byte	\$FF	255
0001 0000 0000	3 nibbles	\$0100	256
0010 0000 0000	3 nibbles	\$0200	512
0011 0000 0000	3 nibbles	\$0300	768
0100 0000 0000	3 nibbles	\$0400	1024
1000 0000 0000	3 nibbles	\$0800	2048
0001 0000 0000 0000	2 bytes	\$1000	4096
0001 1111 1111 1111	2 bytes	\$1FFF	8191
0010 0000 0000 0000	2 bytes	\$2000	8192
0111 1111 1111 1111	2 bytes	\$7FFF	32767
1000 0000 0000 0000	2 bytes	\$8000	32768
1100 0000 0000 0000	2 bytes	\$C000	49152
1111 1111 1111 1110	2 bytes	\$FFFE	65534
1111 1111 1111 1111	2 bytes	\$FFFF	65535

V. Problems

1. Convert these binary numbers to hexadecimal and decimal numbers, then convert your decimal result back into hexadecimal and binary to check both calculations:

00001001
10000000
01000000
00100000
00010000
00001000
11000000
11000011
00111100
11111111

2. Convert these hexadecimal numbers to binary numbers:

\$01
\$05
\$0F
\$50
\$5F
\$A7
\$6F
\$FF

3. Convert these hexadecimal addresses to binary and decimal:

\$0123
\$0700
\$ABCD
\$FFFF

4. Convert these decimal numbers to hexadecimal numbers:

1024
32768
49152
57343



The Computer Assisted Instruction Program

This appendix is a listing of the computer-assisted instruction (CAI) program used for the exercises in Chapters 4, 5, and 7.

1 OPA EQU \$FB	;STORAGE FOR ONE OPERAND.	00FB
2 OPB EQU \$FC	;STORAGE FOR 2ND OPERAND.	00FC
3 TEMP EQU \$02	;TEMPORARY STORAGE.	0002
4 IOINIT EQU \$FF84		FF84
5 CHRROUT EQU \$FFD2		FFD2
6 GETIN EQU \$FFE4		FFE4
7 GETTWO EQU \$C100		C100
8 ;		
9 ;		
10 ;SUBROUTINE GETTWO - GETS TWO BYTES.		
11 GETTWO JSR IOINIT	;INITIALIZE I/O DEVICES.	C100 20 84 FF
12 LDA #\$0D	;CARRIAGE RETURN.	C103 A9 0D
13 JSR CHRROUT		C105 20 D2 FF
14 JSR RDBYTE	;GET A HEX NUMBER.	C108 20 13 C1
15 STA +OPA	;STORE IN OPERAND A.	C10B 85 FB
16 GETONE JSR RDBYTE	;GET A SECOND HEX NUMBER.	C10D 20 13 C1
17 STA +OPB	;STORE IN OPERAND B.	C110 85 FC
18 RTS		C112 60
19 ;		
20 ;SUBROUTINE RDBYTE.		
21 RDBYTE JSR GETIN	;GET A KEY CODE.	C113 20 E4 FF
22 CMP #00	;IS CODE ZERO?	C116 C9 00
23 BEQ RDBYTE	;YES, TRY AGAIN.	C118 F0 F9
24 STA +TEMP	;STORE CODE HERE.	C11A 85 02
25 JSR CHRROUT	;OUTPUT IT TO THE SCREEN.	C11C 20 D2 FF
26 LDA +TEMP	;GET KEY CODE AGAIN.	C11F A5 02
27 JSR ASHEX	;CONVERT IT TO HEX.	C121 20 61 C1
28 ASL A	;SHIFT TO HIGH ORDER NIBBLE.	C124 0A
29 ASL A		C125 0A
30 ASL A		C126 0A
31 ASL A		C127 0A
32 STA +TEMP	;SAVE THE HIGH NIBBLE.	C128 85 02
33 WAIT JSR GETIN	;GET 2ND KEY CODE.	C12A 20 E4 FF
34 CMP #00	;IS IT ZERO?	C12D C9 00
35 BEQ WAIT	;YES, TRY AGAIN.	C12F F0 F9
36 PHA	;SAVE CODE ON THE STACK.	C131 48
37 JSR CHRROUT	;OUTPUT IT TO THE SCREEN.	C132 20 D2 FF
38 PLA	;GET CODE FROM STACK.	C135 68
39 JSR ASHEX	;CONVERT IT TO HEX.	C136 20 61 C1
40 ORA +TEMP	;COMBINE BOTH NIBBLES.	C139 05 02
41 STA +TEMP	;STORE THE HEX NUMBER.	C13B 85 02

42	LDX #06	; INSERT 6 SPACES.	C13D A2 06
43	LDA #\$20	; ASCII SPACE.	C13F A9 20
44	JSR CHRROUT	; OUTPUT SPACE.	C141 20 D2 FF
45	DEX		C144 CA
46	BNE BACK		C145 D0 F8
47	LDX #08	; BIT COUNTER.	C147 A2 08
48	LDA +TEMP	; GET NUMBER.	C149 A5 02
49	STA +\$97	; STORE NUMBER HERE.	C14B 85 97
50	BR3 ASL +\$97	; SHIFT LEFT INTO CARRY.	C14D 06 97
51	LDA #00		C14F A9 00
52	ADC #\$30	; CONVERT TO ASCII.	C151 69 30
53	JSR CHRROUT	; OUTPUT IT.	C153 20 D2 FF
54	DEX		C156 CA
55	BNE BR3		C157 D0 F4
56	LDA #\$0D	; OUTPUT A	C159 A9 0D
57	JSR CHRROUT	; CARRIAGE RETURN.	C15B 20 D2 FF
58	LDA +TEMP	; GET THE NUMBER.	C15E A5 02
59	RTS	; AND RETURN.	C160 60
60	,		
61	; SUBROUTINE ASCII TO HEXADECIMAL.		
62	ASHEX CMP #\$40	; DIGIT OR LETTER?	C161 C9 40
63	BCS BR1		C163 B0 04
64	AND #\$0F	; DIGIT, MASK HIGH NIBBLE.	C165 29 0F
65	BPL BR2	; BRANCH PAST LETTER.	C167 10 02
66	BR1 SBC #\$37	; LETTER, SUBTRACT \$37.	C169 E9 37
67	BR2 RTS	; RETURN WITH DIGIT IN A.	C16B 60
68	,		
69	,		
70	CHRROUT EQU \$FFD2		FFD2
71	DISPLAY EQU SC16C		C16C
72	DISPLAY PHP	; SAVE P ON THE STACK.	C16C 08
73	PHA	; PUT A ON THE STACK.	C16D 48
74	PHP	; SAVE P AGAIN.	C16E 08
75	PLA	; GET P INTO A.	C16F 68
76	LDX #08		C170 A2 08
77	STA +\$FE		C172 85 FE
78	PLA	; GET A BACK FROM THE STACK.	C174 68
79	PHA	; AND PUT IT BACK FOR SAVE KEEPING.	C175 48
80	STA +\$02		C176 85 02
81	BR4 LDA #\$20	; ASCII SPACE.	C178 A9 20
82	JSR CHRROUT		C17A 20 D2 FF
83	DEX		C17D CA
84	BNE BR4		C17E D0 F8
85	LDX #08	; COUNT EIGHT BITS.	C180 A2 08
86	BR5 ASL +\$02	; SHIFT BIT INTO CARRY.	C182 06 02
87	LDA #00	; CLEAR A.	C184 A9 00
88	ADC #\$30	; CONVERT BIT TO ASCII.	C186 69 30
89	JSR CHRROUT	; OUTPUT IT.	C188 20 D2 FF
90	DEX		C18B CA
91	BNE BR5		C18C D0 F4
92	LDA #\$0D	; CARRIAGE RETURN.	C18E A9 0D
93	JSR CHRROUT		C190 20 D2 FF
94	LDA #\$0D		C193 A9 0D
95	JSR CHRROUT	; ANOTHER RETURN.	C195 20 D2 FF
96	LDX #8		C198 A2 08
97	BR6 LDA #\$20		C19A A9 20
98	JSR CHRROUT		C19C 20 D2 FF
99	DEX		C19F CA
100	BNE BR6		C1A0 D0 F8
101	LDX #8		C1A2 A2 08
102	BR7 ASL +\$FE		C1A4 06 FE
103	LDA #00	; CONVERT BIT TO ASCII.	C1A6 A9 00
104	ADC #\$30		C1A8 69 30
105	JSR CHRROUT	; OUTPUT IT.	C1AA 20 D2 FF
106	DEX		C1AD CA
107	BNE BR7		C1AE D0 F4
108	LDA #\$0D		C1B0 A9 0D
109	JSR CHRROUT		C1B2 20 D2 FF
110	LDX #08		C1B5 A2 08
111	BR8 LDA #\$20		C1B7 A9 20
112	JSR CHRROUT		C1B9 20 D2 FF
113	DEX		C1BC CA

114	BNE BR8	C1BD D0 F8
115	LDA #'N	C1BF A9 4E
116	JSR CHRROUT	C1C1 20 D2 FF
117	LDA #'V	C1C4 A9 56
118	JSR CHRROUT	C1C6 20 D2 FF
119	LDA #\$20	C1C9 A9 20
120	JSR CHRROUT	C1CB 20 D2 FF
121	LDA #'B	C1CE A9 42
122	JSR CHRROUT	C1D0 20 D2 FF
123	LDA #'D	C1D3 A9 44
124	JSR CHRROUT	C1D5 20 D2 FF
125	LDA #'I	C1D8 A9 49
126	JSR CHRROUT	C1DA 20 D2 FF
127	LDA #'Z	C1DD A9 5A
128	JSR CHRROUT	C1DF 20 D2 FF
129	LDA #'C	C1E2 A9 43
130	JSR CHRROUT	C1E4 20 D2 FF
131	LDA #\$0D	C1E7 A9 0D
132	JSR CHRROUT	C1E9 20 D2 FF
133	PLA	C1EC 68
134	PLP	C1ED 28
135	RTS	C1EE 60
136 WAITKY	PHP	C1EF 08
137	PHA	C1F0 48
138 LOAF	JSR \$FFE4	C1F1 20 E4 FF
139	BEQ LOAF	C1F4 F0 FB
140	PLA	C1F6 68
141	PLP	C1F7 28
142	RTS	C1F8 60

C

The 6510 Instruction Set Summary

Table C-1. The 6510 instruction set summary.
(Reprinted with permission of Semiconductor Products Division of Rockwell International)

INSTRUCTIONS		IMMEDIATE			ABSOLUTE			ZERO PAGE			ACCUM			IMPLIED			(IND, X)			(IND, Y)				
MNEMONIC	OPERATION	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#		
A D C	A + M + C → A (4) (1)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2		
A N D	A & M → A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2		
A S L	C - [] 0 - 0				0E	6	3	06	5	2	0A	2	1											
B C C	BRANCH ON C = 0 (2)																							
B C S	BRANCH ON C = 1 (2)																							
B E Q	BRANCH ON Z = 1 (2)																							
B I T	A & M				2C	4	3	24	3	2														
B M I	BRANCH ON N = 1 (2)																							
B N E	BRANCH ON Z = 0 (2)																							
B P L	BRANCH ON N = 0 (2)																							
B R K	BREAK															00	7	1						
B V C	BRANCH ON V = 0 (2)																							
B V S	BRANCH ON V = 1 (2)															18	2	1						
C L C	0 → C															18	2	1						
C L D	0 → D															08	2	1						
C L I	0 → I															58	2	1						
C L V	0 → V															BB	2	1						
C M P	A - M	C9	2	2	CD	4	3	C5	3	2									C1	6	2	D1	5	2
C P X	X - M	E0	2	2	EC	4	3	E4	3	2														
C P Y	Y - M	C0	2	2	CC	4	3	C4	3	2														
D E C	M - 1 → M				CE	6	3	C6	5	2														
D E X	X - 1 → X															CA	2	1						
D E Y	Y - 1 → Y															88	2	1						
E O R	A ∨ M → A (1)	49	2	2	4D	4	3	45	3	2									41	6	2	51	5	2
I N C	M + 1 → M				EE	6	3	E6	5	2														
I N X	X + 1 → X															E8	2	1						
I N Y	Y + 1 → Y															C8	2	1						
J M P	JUMP TO NEW LOC				4C	3	3																	
J S R	JUMP SUB				20	6	3																	
L D A	M → A (1)	A9	2	2	AD	4	3	A5	3	2									A1	6	2	B1	5	2

Table C-1. The 6510 instruction set summary (continued).
(Reprinted with permission of Semiconductor Products Division of Rockwell International)

Z PAGE. X			ABS X			ABS. Y			RELATIVE			INDIRECT			Z PAGE. Y			PROCESSOR STATUS CODES									
OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	7	6	5	4	3	2	1	0	MNEMONIC	
																		N	V	*	B	D	I	Z	C		
75	4	2	7D	4	3	79	4	3										N	V	ADC	
35	4	2	3D	4	3	39	4	3										N	Z	.	.	AND	
16	6	2	1E	7	3							90	2	2				N	.	.	.	Z	C	A S L			
												B0	2	2				BCC	
												F0	2	2				BCS	
												30	2	2				M ₇	M ₆	.	.	.	Z	.	.	.	BEQ
												D0	2	2				BIT	
												F0	2	2				BMI	
												50	2	2				BNE	
												70	2	2				BPL	
																	.	.	.	1	.	1	.	.	BRK		
																	BVC	
																	BVS	
																	0	.	.	CLC	
																	.	.	.	0	CLD	
																	0	CLI	
D5	4	2	DD	4	3	D9	4	3									.	0	CLV	
																	N	.	.	.	Z	C	C M P				
																	N	.	.	.	Z	C	C P X				
																	N	.	.	.	Z	C	C P Y				
D6	6	2	DE	7	3													N	.	.	.	Z	.	DEC			
																	N	.	.	.	Z	.	DEX				
																	N	.	.	.	Z	.	DEY				
55	4	2	5D	4	3	59	4	3									N	.	.	.	Z	.	E O R				
F6	6	2	FE	7	3												N	.	.	.	Z	.	I N C				
																										IN X	
																										IN Y	
																										J M P	
																										J S R	
B5	4	2	BD	4	3	B9	4	3									6C	5	3							L D A	

Table C-1. The 6510 instruction set summary (continued).
(Reprinted with permission of Semiconductor Products Division of
Rockwell International)

Table C-1. The 6510 instruction set summary (continued).
(Reprinted with permission of Semiconductor Products Division of
Rockwell International)

Z PAGE, X			ABS. X		ABS. Y		RELATIVE		INDIRECT		Z. PAGE. Y		PROCESSOR STATUS CODES								MNEMONIC				
OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	7	6	5	4	3	2	1	0			
						BE	4	3				B6	4	2	N	Z	.	.	LDX		
B4	4	2	BC	4	3										N	Z	.	.	LDY		
56	6	2	5E	7	3										O	Z	C	.	LSR		
															NOP		
15	4	2	1D	4	3	19	4	3							N	Z	.	.	ORA		
															PHA		
															PHP		
															N	Z	.	.	PLA		
															(RESTORED)								PLP		
36	6	2	3E	7	3										N	Z	C	.	ROL		
76	6	2	7E	7	3										N	.	.	.	Z	C			ROR		
															(RESTORED)								RTI		
															RTS		
F5	4	2	FD	4	3	F9	4	3							N	V	.	.	Z	(3)			SBC		
															1	.	.	SEC		
															.	.	.	1	•	•	.	.	SED		
															1	•	•	.	.	SEI	
95	4	2	9D	5	3	99	5	3																STA	
																							STX		
94	4	2													96	4	2								STY
															TAX		
															N	.	.	.	Z	.	.	.	TAY		
															N	.	.	.	Z	.	.	.	TSX		
															N	.	.	.	Z	.	.	.	TXA		
															TXS		
															N	.	.	.	Z	.	.	.	TYA		
X	INDEX X													+	ADD		M ₇	MEMORY BIT 7							
Y	INDEX Y													-	SUBTRACT		M ₆	MEMORY BIT 6							
A	ACCUMULATOR													Λ	AND		n	NO. CYCLES							
M	MEMORY PER EFFECTIVE ADDRESS													V	OR		#	NO. BYTES							
Ms	MEMORY PER STACK POINTER													▼	EXCLUSIVE OR										

Glossary

Absolute addressing mode—Addressing mode in which the second and third bytes of the instruction specify the 16-bit address of the operand.

Absolute indexed addressing mode—The address of the operand of the instruction is determined by adding the number in an index register (X or Y) to the second two bytes of the instruction.

Accumulator—Eight-bit register in the 6510 microprocessor used to transfer eight-bit codes or numbers between the microprocessor and memory. It is also used whenever the microprocessor combines *two* numbers in addition, subtraction, or logical operations.

Accumulator addressing mode—The operand of the instruction is the eight-bit code in the accumulator.

Address—A number used to identify one of the microcomputer's 65,536 memory locations.

Address bus—16-conductor bus connecting the microprocessor to each memory location or I/O device. Each conductor represents one bit of a 16-bit binary number called the address.

Addressing mode—The way a 6510 instruction identifies the register or address of the operand. The addressing mode is the way a 6510 identifies a destination when a jump or branch instruction is executed.

ADH—The most-significant byte of a 16-bit address.

ADL—The least-significant byte of a 16-bit address.

ADSR envelope—The waveshape that defines the attack, decay, sustain, and release parameters associated with a single musical note.

Analog-to-digital converter—Device used to represent an analog quantity with a number.

ASCII—American Standard Code for Information Interchange. A set of eight-bit codes used to represent alphanumeric characters, punctuation marks, and control codes.

Assembler—Computer program that converts an assembly-language program to a machine-language program.

BASIC interpreter—A machine-language program stored in ROM that carries out commands written in BASIC.

Binary coded decimal (BCD)—A set of ten binary codes that represent the decimal digits 0 through 9.

Bit—Contraction of the words "binary digit." One binary digit is the smallest quantity of information used in a computer.

Bit-mapped graphics—A technique of displaying information on the video monitor. Each bit in an 8,000-byte block of memory maps into one pixel on the screen.

Bit-mapped memory—The 8,000-byte block of memory used for bit-mapped graphics.

Branch—Modification of the 16-bit program counter by an eight-bit signed number.

Branch instruction—A 6510 instruction that tests the condition of a flag in the P register. If the condition is met, an eight-bit signed number is added to the program counter. Otherwise the program counter increments by one.

Break flag (B)—The break flag in the P register. It is set when a BRK instruction is executed. It is cleared when an IRQ-type interrupt occurs.

Bus—A set of electrical conductors that interconnects all of the components of a microcomputer system.

Byte—Eight binary digits of information. An eight-bit code.

Carry flag (C)—Flag in the P register that is modified by arithmetic, compare, rotate, and shift instructions.

Character memory—A 2K block of memory that contains codes that determine the shape of the characters displayed on the screen.

Character ROM—See "Character memory."

Clear—Give one or more bits a value of zero.

Code—A set of binary digits that convey information.

Color memory—A 1,000-byte block of memory containing the codes that determine the colors of the characters on the screen. Color memory is from \$D800 to \$DBE7.

Compare instruction—An instruction that subtracts a number in memory from a number in a 6510 register to determine whether the number in the register is greater than, equal to, or lesser than the number in memory.

Complement—To change each of the bit values of a number.

Complex interface adapter (CIA)—An integrated circuit used to interface the 6510 microprocessor to various input/output devices.

Computer programming language—A means by which human beings instruct a computer to perform tasks.

Condition code—Another name for a flag in the P register.

Control bus—One of three important buses in the microcomputer system. Includes the read/write line and the clock line.

Counter/timer—An integrated circuit that decrements a number in a register each time an electrical pulse occurs on a particular pin.

Data—Information represented by eight-bit codes in computer.

Data bus—Eight-conductor bus used to transfer eight-bit codes between the microprocessor and memory. Each conductor represents one bit of an eight-bit code or number.

Decimal mode flag (D)—A flag in the P register which, when set, causes all arithmetic operations to be performed in BCD.

Delay loop—A programming structure designed to take a specific amount of time.

Disassemble—Change machine-language into assembly-language mnemonics and hexadecimal operand fields.

Dot matrix—A rectangular pattern of dots that conveys information.

Editor/assembler—Computer program to enter, modify, and assemble assembly-language programs.

Eight-bit code—Eight binary digits of information arranged in a particular order.

Filter—Circuitry on the SID that can be programmed to reject certain frequencies.

Flag—A bit used to signal a particular event. Each of the bits in the P register is a flag.

Flag-modification instruction—Instruction that sets or clears a flag in the P register.

Frequency—The rate (number of times per second) at which a waveform repeats itself.

Fundamental frequency—The dominant frequency of a particular sound waveform.

Harmonic—A frequency component of a waveform. This component has a frequency that is a whole-number multiple of the fundamental frequency.

Hexadecimal number—Base-sixteen number using the numerals 0 through 9 and A through F and a "\$" prefix.

High-level language—A programming language that has a structure and syntax similar to English.

Immediate addressing mode—The operand is the second byte of the instruction.

Implied addressing mode—The location of the operand is defined by the instruction. No address is needed to locate the operand.

Indexed indirect addressing mode—Seldom used mode that adds the second byte of the instruction and the number in the X register to identify the first of two sequential page-zero memory locations that contain the address of the operand.

Indirect addressing mode—In this mode, the second two bytes of the instruction specify the address of the first of two sequential memory locations that contains the destination address of the JMP instruction.

Indirect indexed addressing mode—The second byte of the instruction identifies the first of two sequential page-zero memory locations that contain a 16-bit number which, when added to the number in the Y register, is the address of the operand.

Input port—A set of one or more pins whose voltage levels determine the bit values in a specific memory location.

Instruction—A set of one, two, or three eight-bit codes that cause the 6510 microprocessor to carry out one of the 56 operations in its instruction set.

Instruction set—The group of 56 instructions capable of being performed by the 6510 microprocessor.

Interfacing—The process of connecting the computer to external devices.

Interrupt—A design feature of the 6510 microprocessor that allows an external signal (interrupt request) to halt the program currently being executed, execute another program, and then return to execute the original program.

Interrupt flag (I)—A flag in the P register that is set whenever the microprocessor is processing an interrupt. IRQ-type interrupts are not recognized when this flag is set.

Interrupt request—An external signal on either the IRQ or NMI pins on the 6510 that signals the need for an interrupt.

Interrupt routine—The program that is executed when an interrupt request is recognized.

IRQ-type interrupt—Interrupt requested on the IRQ pin on the 6510.

IRQ vector—The 16-bit number stored in locations \$FFFE and \$FFFF that identifies the starting address of the IRQ-type interrupt routine.

Label—Labels are names used in assembly-language programs to identify the addresses of certain instructions.

Logic operation—An operation accomplished with an AND, EOR, or ORA instruction.

Loop—Programming structure that provides for the repetitive performance of a programming task.

Low-level language—Programming language which, unlike English, is similar to machine language.

Machine-language instruction—See "Instruction."

Machine-language program—A set of instructions stored in sequential memory locations which, when read by the microprocessor, will perform a specific task.

Mask—One of the two bytes involved in an AND instruction. The mask byte has zeros in the bits that are not being tested and ones in the bits being tested.

Memory map—Diagram showing how memory locations in the microcomputer system are allocated.

Memory-mapped I/O—Technique used in 6510-based microcomputer systems in which I/O ports are part of the memory organization of the microcomputer.

Microcomputer clock—Electrical signal (square waveform) that repeats itself 1,022,727 times a second and is used to synchronize all of the microprocessor's read and write operations.

Microprocessor—The 6510 40-pin integrated circuit that performs all of the operations described in the instruction set.

Mnemonic—One of the 56 three-letter words used by human beings to identify a 6510 instruction. The part of an assembly language instruction that identifies which of the 56 operations in the instruction set is to be executed.

Multiple-byte arithmetic—Arithmetic operations in which the numbers must be represented by more than one byte.

Negative flag (N)—Flag in the P register that is set whenever the result of an operation has a one in bit seven; otherwise, it is cleared.

Nibble—Four bits of binary information.

NMI-type interrupt—Interrupt requested with a signal on the NMI pin of the 6510. This interrupt is nonmaskable; it cannot be disabled by the I flag.

Non-volatile memory—Memory in which information is not lost when power is removed.

Object program—The machine-language program that is produced when an assembly-language program is assembled.

Offset—The second byte of a branch instruction. An eight-bit signed number that is added to the program counter when the branch is taken.

Op code—The first byte of any instruction used to identify one of the 56 operations and one of the 13 addressing modes available to the 6510.

Operand—The eight-bit code that is the object of an instruction. The location of the operand is either implied by the instruction or identified by the second and/or third bytes of the instruction.

Operand field—The part of an assembly-language instruction that symbolizes the location of the operand. The operand field of a jump, subroutine call, or branch instruction contains a label for the address of the destination.

Operating system—A program that provides the microcomputer with essential input and output capabilities.

Output port—A set of one or more pins whose voltage levels are determined by the binary values of corresponding bits in some memory location.

Overflow flag (V)—Flag in the P register that is set when the sum or difference of two eight-bit signed numbers either exceeds +127 or is less than -128.

Page—A set of 256 (\$FF + 1) memory locations beginning at one of the following page boundaries: \$0000, \$0100, \$0200, \$0300, ..., \$FF00.

Page one—The set of 256 memory locations with addresses from \$0100 to \$01FF. This is also known as the stack.

Page zero—The set of 256 memory locations with addresses from \$0000 to \$00FF.

Parallel I/O—Input/output operations that involve several bits simultaneously.

Pixel—Contraction of the words "picture element." One dot on the screen of the video monitor.

Processor status register—Also known as the P register. The register in the 6510 that holds the N, V, B, D, I, Z, and C flags.

Program counter—16-bit register in the 6510 that contains the address of the memory location of the next byte of the machine-language program being executed.

Pseudo-code—An informal combination of assembly language and English that a programmer may use to describe what a program is to do before writing it.

RAM—See "Read/write memory."

Read only memory (ROM)—Memory that can be read but cannot be modified with a 6510 write operation. ROM is usually non-volatile memory.

Read operation—The 6510 copies a code in memory into one of its registers.

Read/write (R/W) line—Control bus line that informs the components of a microcomputer whether a read or write operation is taking place.

Read/write (R/W) memory—Memory that functions with both read and write operations.

Relative addressing mode—Used exclusively by branch instructions. The second byte of the instruction, called the offset, is added to the program counter to calculate the destination of the branch.

Relocatable program—A program that does not need to be modified if its location in memory is changed.

Resolution—A term related to the number of pixels on the screen.

ROM—See "Read only memory."

Screen—The face of the video monitor or TV set.

Screen character codes—Eight-bit codes which when placed in screen memory, cause a specific character to appear on the screen.

Screen memory—A bank of 1,000 memory locations whose contents determine which characters appear on the screen.

Serial I/O—A form of I/O using a one-bit I/O port in which binary codes are input or output timewise.

Set—Give one or more bits a binary value of one.

Single-step mode—A process of executing a program one instruction at a time.

Sound interface device (SID)—The integrated circuit used to produce sound effects.

Source program—The assembly-language program that is used to produce the corresponding machine-language program called the object program.

Sprite—A pattern of dots on the screen whose position, color, and shape can be programmed.

Stack—256 memory locations with addresses from \$0100 to \$01FF.

Stack operation instructions—6510 instructions that save (push) codes on the stack or read (pull) codes from the stack. Also includes TSX and TXS instructions.

Stack pointer (S)—Eight-bit 6510 register that identifies the least-significant byte of the address of the next available stack location.

Status bit—Same as flags or condition codes.

Symbol—A name for an address in the operand field of an assembly-language instruction.

Test instructions—The compare and BIT instructions.

Toggle—Alternately switch on and off.

Two's complement—Number obtained by forming the complement and adding one.

Underflow—Term used in timing and counting applications to indicate when a register has decremented through zero.

Vector—A 16-bit number that identifies the address of the destination of a jump instruction or the starting address of an interrupt routine.

Video interface chip (VIC)—The integrated circuit, actually known as the VIC-II, used by the Commodore 64 to provide output video information to the screen.

Volatile memory—Loses stored information when power is removed.

Write-only registers—Registers that function only with write operations. These registers cannot be read.

Write operation—The 6510 copies a code from one of its register into a memory location.

X register—Eight-bit register in the 6510 used for data transfers and as an index in the indexed addressing modes.

Y register—Eight-bit register in the 6510 used for data transfers and as an index in the indexed addressing modes.

Zero flag (Z)—Flag in the P register that is set when the result of an operation has zeros in all eight bits; otherwise, the flag is cleared.

Zero page—See “Page zero.”

Zero-page addressing mode—The operand is in the page-zero location whose least-significant byte is the second byte of the instruction.

Zero-page indexed addressing mode—The operand is in the page-zero location whose least-significant byte is the sum of the second byte of the instruction and the number in an index register.

Index

- Accumulator, 16, 18, 19
ADC instruction, 58, 60, 61
Address, 6, 260
Address bus, 6
Addressing modes, 25
 absolute, 30
 absolute indexed, 134
 accumulator, 118
 immediate, 30
 implied, 31
 indirect, 157
 indirect indexed, 140
 indexed indirect, 146
 relative, 94
 zero-page, 30
 zero-page indexed, 138
ADH, 25, 264
ADL, 15, 264
ADSR envelope, 183
 attack, 138
 decay, 183
 release, 183
 sustain, 183
Analog-to-digital conversion, 236
AND instruction, 77, 78, 79
ASCII, 111
ASCII-to-hexadecimal, 114
ASL instruction, 117, 118
Assembler, 24, 48

Backward branch, 94, 98
Base address, 135
Base two, 259
BASIC interpreter, 5

BASIC music interpreter, 199
BCC instruction, 91, 102
BCD (see binary-coded decimal)
BCD-to-binary routine, 130
BCS instruction, 91, 102
BEQ instruction, 91, 102
Binary-coded decimal (BCD), 67, 84
Binary numbers, 259, 262
Binary-to-BCD routine, 129
Bit instruction, 93, 102
Bit-mapped graphics, 210
 demonstration program, 221
 graphing, 220
Bit-mapped memory, 210
Bit-mapped mode, 210
BMI instruction, 91
BNE instruction, 91, 102
Borrow, 64
BPL instruction, 91
Branch calculation, 96
Branch destination, 96
Branch forced, 98
Branch instructions, 94
Branch unconditional, 98
Break flag (B), 58, 169
BRK instruction, 154, 169
Buses, 6
BVC instruction, 91
BVS instruction, 91
Byte, 4, 260
Byte-input routine, 123
Byte-print routine, 122

Carry flag (C), 58, 59, 93

- Character memory, 207
Character plot routine, 148
Character ROM, 208
CIA, 225
CLC instruction, 58, 59, 60
CLD instruction, 58, 59
Clear, 58, 81
Clearing, 48, 81
Clearing bit-mapped memory, 144
CLI instruction, 154, 167
CLV instruction, 58, 59, 71
CMP instruction, 93, 101
Code-display routine, 121
Code-shift routine, 119
Color memory, 147, 206
Comparison instructions, 101
Complement, 64, 65, 79
Computer, 57
Computer-assisted instruction program, 73, 87, 131, 269
Computer programming language, 1
Condition code, 58
Control bus, 9
Controller, 57
Counting/timing, 242
CPX instruction, 93, 101
CPY instruction, 93, 101
- Data bus, 8
Data processor, 57
Data transfer instructions, 43
Decimal-mode arithmetic, 67
DEC instruction, 93, 99
Decimal-mode flag (D), 58, 68
Decimal numbers, 259
Delay loop, 100
Delay routine, 246, 248
Development system, 48
DEX instruction, 93, 99
DEY instruction, 93, 99
Directive, 49, 140
Disassembled, 156
Division, 129
Dot matrix, 207
- Editor/assembler, 35, 48
Eight-bit code, 4, 260
- Eight-bit multiplication routine, 128
EOR instruction, 77, 78, 79
Exclusive OR, 77
- Face value, 261
Filter, 181
FIFO buffer, 172
Flag, 58
Flag-modification instructions, 59
Forward branch, 94
Four-bit multiplication routine, 127
Frequency, 180
Frequency-counting routine, 253
Fundamental, 180
- Harmonic structure, 180
Harmonics, 180
Hexadecimal numbers, 265
Hexadecimal-to-ASCII, 111
High-level language, 1
- Image, 184
INC instruction, 93, 99
Indexed addressing modes, 133
ABS,X, 134, 135
ABS,Y, 134
(IND,X), 146
(IND,Y), 140, 141
Z Page,X, 138, 139
Z Page,Y, 138
Indirect addressing, 157
Indirect jump vector, 158
Input/output, 5, 20, 226
Input port, 226
Instruction, 10, 23
Instruction set, 10, 26, 272
Interfacing, 228
Interrupt, 166
Interrupt-driven keyboard routine, 171
Interrupt flag (I), 58, 167
Interrupt routine, 166
INX instruction, 93, 99
INY instruction, 93, 99
IRQ-type interrupt, 166
IRQ vector, 168

- JMP instruction, 154, 155
 JMP indirect, 157
 Joystick input routine, 231
 JSR instruction, 58, 71, 154, 159
 Jump table, 157
- Keyboard input routine, 123, 145, 155, 172, 175
- Label, 32, 72, 155
 LDA instruction, 13, 14, 41, 43
 LDX instruction, 41, 43
 LDY instruction, 41, 43
 Least-significant byte (LSB), 63
 Logic operations, 77
 Loop, 96, 98, 99, 109
 Low-level language, 1
 LSR instruction, 117, 118
- Machine-language program, 13, 17
 Mask, 83
 Memory map, 6, 7
 Memory-mapped I/O, 227
 Microcomputer, 2
 Microcomputer clock, 9, 10
 Microprocessor, 2, 4
 Mnemonic, 10, 24, 26
 Monitor, 35
 Most-significant byte (MSB), 63
 Multiple-byte addition, 63
 Multiple-byte subtraction, 66
 Multiplicand, 126
 Multiplication, 126, 127
 Multiplier, 126
- Negative flag (N), 58, 70, 81, 93
 Negative numbers, 70
 Nested subroutines, 162
 Nibble, 67, 265
 Nonmaskable interrupt (NMI), 175
 Nonvolatile memory, 5
 Note dynamics, 183
 Numeration, 259
- Object program, 33
 Octaves, 180, 191
 Offset, 94
 One-bit I/O routine, 238
Op code, see Operation code
 Operand, 25
 Operand field, 32
 Operating system, 5
 Operation code, 23, 26
 ORA instruction, 77, 78, 79
 Output port, 227
 Overflow flag (V), 58, 71, 93
- Page, 136
 Page boundary, 136
 Page crossing, 137
 Page one, 153
 Page zero, 30, 138
 Parallel I/O, 227
 Partial products, 127
 PHA instruction, 154, 163
 PHP instruction, 154, 163
 Pixel, 207, 211
 PLA instruction, 154, 163
 Place value, 261
 PLP instruction, 154, 163
 Potentiometer inputs, 235
 Processor status register, 19, 20, 58, 93
Processor status register flags, 58
 B flag, 169
 C flag, 59, 93
 D flag, 68
 I flag, 168
 N flag, 70, 81, 93
 V flag, 71, 81, 93
 Z flag, 81, 93
- Program counter, 19, 20, 92, 153
 Programming form, 35
 Pseudo-code, 52
 Pulse counting, 250
 Pulse width, 181, 182
 Push-down stack, 160
- RAM, *see* read/write memory
 Ramp waveform, 181, 182

Read only memory (ROM), 4
 Read operation, 4
 Read/write line (R/W line), 9
 Read/write (R/W) memory, 4
 Reading the POT inputs, 235
 Relocatable, 156
 Resolution, 210
 ROM, *see* read only memory
 ROM switching, 219
 ROL instruction, 117, 118
 ROR instruction, 117, 118
 RTI instruction, 154, 168
 RTS instruction, 34, 71, 154, 159, 162

SBC instruction, 58, 64
 Screen character code, 204
 Screen memory, 204
 SEC instruction, 58, 59
 SED instruction, 58, 59, 68
 SEI instruction, 154
 Serial I/O, 227
 Set, 58, 81
 Shift and rotate, 117
 SID routines, 45, 105, 136
 delay/tempo, 195
 frequency control, 193
 song, 197
 song table, 198
 potentiometer input, 235
 Signed-number arithmetic, 69
 Sine waveform, 181
 Single-step mode, 53
 6581 registers, 184, 185
 6567 VIC II, 203
 6510 microprocessor
 architecture, 19
 I/O ports, 20
 microprocessor, 4
 registers, 18, 19
 6526, CIA, 225
 counter/timers, 242
 data-direction registers, 228
 input pins, 226
 interrupts, 224, 245, 246
 output pins, 227
 time-of-day (TOD) clock, 67, 84, 104, 126

Source program, 33
 Sprite routines, 46, 105
 Square waveform, 181, 182
 STA instruction, 13, 15, 41
 Stack, 160
 Stack-operation instructions, 154, 162
 Stack pointer (S), 19, 20, 160, 162
 Status bit, 58
 Storing keyboard codes, 145
 STX instruction, 41, 43
 STY instruction, 41, 43
 Subroutine, 162
 Symbol, 32

Test instructions, 101
 TAX instruction, 41, 46, 47
 TAY instruction, 41, 46
 Timbre, 180
 Timing routines, 108, 246, 248, 250, 253
 Triangle waveform, 181
 Toggle, 82
 TSX instruction, 154, 166
 Two's complement, 65, 69
 TXA instruction, 41, 46, 47
 TXS instruction, 154, 166
 TYA instruction, 41, 46

Underflow, 243

Vector, 158
 VIC
 bank switching, 203
 background color register, 207
 character memory, 207
 character ROM, 208
 color codes, 206
 color memory, 206
 exterior color register, 207
 routines, 45, 81, 82, 83, 105, 137, 147, 149, 203
 screen memory, 204
 Volatile memory, 5
 Volume, 180

Waveform, 181

Write-only register, 106, 184

Write operation, 5

X register, 19, 134

Y register, 19, 141

Zero flag (Z), 58, 81, 93

Zero page, *see* page zero

Documentation for Diskette to Accompany Assembly Language Programming with the Commodore 64

This documentation is applicable only if you have purchased the kit or the diskette that includes a floppy disk containing all of the programs in *Assembly Language Programming with the Commodore 64*. All of the assembly language programs have been assembled and the machine language codes are stored in sequential (SEQ) files on the disk. The BASIC programs in the book are stored as PRG files. Refer to the disk directory on page 294. A description of how to use the disk follows.

To load a BASIC program from the disk type

```
LOAD "FILENAME",8
```

where FILENAME is one of the names given in quotes in the disk directory on page 294. To illustrate, Example 2-11 in the book is a BASIC program. In the disk directory it is listed as EX 2-11. To load it into the computer's memory, insert the disk into the disk drive and then type

```
LOAD "EX 2-11",8
```

The program will be loaded. When the disk drive stops, the program can be executed from BASIC with the RUN command.

294 Diskette Documentation

0	"DE*JONG/BRADY***" *84*2A		1	"EXS 6-18&19"	SEQ
4	"EX 2-10"	PRG	1	"EX 7-1"	SEQ
2	"EX 2-11"	PRG	1	"EX 7-2"	SEQ
2	"EX 2-8"	PRG	2	"EXS 7-3&7-4"	SEQ
3	"EX 2-9"	PRG	1	"EX 7-5"	SEQ
1	"EX 2-7"	PRG	1	"EX 7-6"	SEQ
1	"EX 2-6"	SEQ	1	"EX 7-7"	SEQ
1	"EX 3-1"	SEQ	1	"EX 7-8"	SEQ
1	"EX 3-2"	SEQ	1	"EX 7-9"	SEQ
1	"EX 3-3"	SEQ	1	"EX 7-10"	SEQ
1	"EX 3-4"	SEQ	1	"EX 7-11"	SEQ
1	"CLEAR SID"	PRG	1	"EX 7-12"	SEQ
1	"EX 3-5"	SEQ	1	"EX 7-13"	SEQ
1	"EX 3-6"	SEQ	1	"EX 8-2"	SEQ
1	"EX 3-7"	PRG	1	"EX 8-3"	SEQ
1	"EX 3-8"	SEQ	4	"EX 8-4 ETC."	SEQ
1	"EX 3-9"	SEQ	4	"EX 8-8 ETC."	SEQ
1	"EX 3-10"	PRG	1	"EX 8-6"	SEQ
5	"CAI PROGRAM"	SEQ	1	"EX 8-6 DATA"	SEQ
1	"EXERCISE 4-1"	PRG	1	"EX 9-2"	SEQ
1	"EX 4-13"	SEQ	1	"EX 9-1"	SEQ
1	"EX 4-2"	SEQ	1	"EX 9-3"	SEQ
1	"EX 4-4"	SEQ	1	"EX 9-4"	SEQ
5	"EX B-1"	SEQ	1	"EX 9-5A"	SEQ
1	"EX 4-6"	SEQ	1	"EX 9-5B"	SEQ
1	"EX 4-8"	SEQ	1	"EX 9-6A"	SEQ
1	"EX 4-10"	SEQ	1	"EX 9-6B"	SEQ
1	"EX 4-12"	SEQ	6	"SID SUBRTNS"	SEQ
1	"EX 5-1"	SEQ	6	"SONG TABLE"	SEQ
1	"EX 5-3"	SEQ	2	"EX 10-11"	SEQ
1	"EXS 5-4T05-6"	SEQ	3	"EX 10-13"	PRG
1	"EX 5-7"	SEQ	1	"FREQ TABLE"	SEQ
1	"EX 5-8 BASIC"	PRG	1	"EX 11-8"	PRG
1	"EX 5-8"	SEQ	6	"BMM GRAPHICS"	SEQ
1	"EX 5-9"	SEQ	2	"EX 11-1"	SEQ
1	"EX 5-10"	SEQ	2	"EX 12-1"	SEQ
1	"EX 5-10 BASIC"	PRG	2	"EX 12-3"	PRG
1	"EX 6-3"	SEQ	1	"EX 12-2"	SEQ
1	"EX 6-6"	SEQ	1	"EX 12-4"	SEQ
1	"EX 6-7"	SEQ	1	"EX 12-5"	SEQ
1	"EX 6-11"	SEQ	1	"EX 12-6"	SEQ
1	"EX 6-12"	SEQ	2	"EX 12-7"	SEQ
2	"EX 6-13"	SEQ	3	"EX 12-9"	SEQ
1	"MAKE SPRITE"	PRG	2	"EX 12-10"	PRG
2	"EX 6-14"	SEQ	2	"EX 12-11"	PRG
2	"EX 6-15"	SEQ	1	"EX 8-11"	SEQ
2	"EX 6-17"	PRG	518 BLOCKS FREE.		
2	"EX 6-16"	SEQ			

To load one of the machine language programs, stored as sequential (SEQ) files, you must first load the BASIC program EX 2-11. When EX 2-11 is loaded, RUN it. It will ask for a file name. Type in the file name shown in the disk directory. For example, if you wish to load the program identified in the book as Example 3-5 and in the disk directory as EX 3-5, then type

EX 3-5

when the file name is requested. Example 2-11 reads the program codes stored on the disk, prints the codes on the screen, and POKEs them into the memory locations identified in the book. Once the codes are loaded the machine language program can be executed from BASIC with a SYS xxxx command, where xxxx is the starting address of the program expressed in decimal. The program listed as EX 2-11 prints the starting and ending addresses of the machine language program it stores in memory.

What follows are a few comments about some programs shown on the disk directory.

"EX 3-5"—Load and RUN the "CLEAR SID" BASIC program before using the SYS 49152 command to execute EX 3-5.

"EX 3-6"—Load and RUN "EX 3-7" to create a sprite before executing EX 3-6.

"EX 3-10"—This program is only of use to owners of the French Silk "Develop-64" package.

"EXERCISE 4-1"—This is the BASIC program listed in the first exercise at the end of Chapter 4. To use it, first load both EX 4-13 and CAI PROGRAM.

"EX 4-13"—See the note on "EXERCISE 4-1".

"EX B-1"—This is a repeat of the CAI program. It is listed in Appendix B in the book. It contains the same codes as the file called "CAI PROGRAM".

"EX 5-8"—Call this program as a subroutine from the EX 5-8 BASIC program. Load EX 5-8, then load EX 5-8 BASIC.

"EX 5-10"—Be sure to load the CAI PROGRAM before running this program. Also, call EX 5-10 as a subroutine from the EX 5-10 BASIC program on the disk. Load EX 5-10, then CAI PROGRAM, then EX 5-10 BASIC.

"EX 6-14"—Load and run the MAKE SPRITE program before executing this program.

"EX 6-15"—Before executing this program, clear the SID registers by running the CLEAR SID program.

"EX 6-16"—Call this program as a subroutine from the BASIC program identified as EX 6-17. First load EX 6-16, then load and RUN EX 6-17. You will need a game paddle with a fire button plugged into Control Port 2.

"**EX 6-18&19**"—These programs are frequently used together, so we placed them in a single sequential file.

"**EXS 7-3&7-4**"—This file includes the routines in EX 6-18&19 which are called as subroutines. Test all of these routines with the program in EX 7-5.

"**EX 7-13**"—Load the CAI PROGRAM file before executing this program.

"**EX 8-4 ETC.**"—This file includes Examples 8-4, 8-10, 8-12, 8-13, and 8-14 in the book.

"**EX 8-8 ETC.**"—This file includes Examples 8-8, 8-9, 6-18, 6-19, 7-3, 7-4, and 9-2 in the book.

"**EX 8-6**"—Be sure to load the EX 8-6 DATA file before executing this program.

"**EX 9-5A**"—Also load EX 9-5B.

"**EX 9-6A**"—Also load EX 9-6B.

"**SID SUBRTNS**"—Includes Examples 10-1, 10-2, 10-3, 10-4, 10-5, 10-6, 10-7, 10-8, 10-9, 10-10, and 10-12. All of these subroutines are called from EX 10-11.

"**EX 10-11**"—Load SID SUBRTNS, SONG TABLE, and FREQ TABLE before executing this program.

"**BMM GRAPHICS**"—All of the bit-mapped graphics routines in Chapter 11 are stored in this file. Test them with EX 11-8, which calls them as subroutines.

"**EX 12-2**"—Load the BMM GRAPHICS routines before calling this program as a subroutine from the BASIC program titled EX 12-3. Connect a game paddle to Control Port 2.

"**EX 12-8**"—A source of pulses connected to the CNT pin, pin number 6 on the user port, is required to test this program.

"**EX 12-9**"—Like EX 12-8, a source of pulses must be connected to the CNT pin, pin 6 on the user port, to test this program. This program is called as a subroutine from EX 12-10 or EX 12-11.

Introducing —

Diskette to Accompany Assembly Language Programming with the Commodore 64

Marvin DeJong

Virtually *instant* access to more than 75 programs is within your grasp. Because instead of spending hours keyboarding code, all you have to do is boot a single diskette!

What's more, you'll be able to get right into such programs as . . .

- **Bit-Mapped Graphics**
- **Computer-Assisted Instruction:** gives you a window into how assembly language is performed.
- **Song Playing:** play the music already programmed or compose your own.

Here's How To Order Your Copy

Enclose a check or money order for \$25.00 plus local sales tax, slip in this handy order envelope and mail. No postage needed. Or charge it to your VISA or MasterCard.

YES! I want to talk to my C64 in its own language! Please rush me **Diskette to Accompany Assembly Language Programming with the Commodore 64/D3278-1**. I have enclosed payment of \$25.00 plus local sales tax.

Name _____

Charge my Credit Card Instead

VISA

MasterCard

Address _____

City _____ State _____ Zip _____

Account Number _____

Dept. Y

Expiration Date _____

Signature as it appears on Card _____

Brady

Brady Communications Co., Inc. • Bowie, MD 20715
A Prentice-Hall Publishing and Communications Company

Now you can

**TALK TO YOUR COMMODORE 64
IN ITS OWN LANGUAGE!**

See over for details . . .



**NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES**

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1976 BOWIE, MD

POSTAGE WILL BE PAID BY ADDRESSEE

Brady Communications Co., Inc.
A Prentice-Hall Publishing Company
Bowie, Maryland 20715

Prepublication reviewers say:

"...an assembly language guide that's truly dedicated to the Commodore 64. . .the writing style, exercises, and examples are excellent!"

"De Jong understands that it's essential to start at the beginning, but miraculously has found a way to do this without insulting the educated reader. . .this is the best written micro book I've come across in recent history!"

Assembly Language Programming with the Commodore 64

Marvin L. De Jong

Here is the comprehensive introduction to assembly language for beginning Commodore 64 programmers! This unique guide offers extensive coverage on how to write, debug, and execute assembly language programs—complete with numerous exercises and problems designed to familiarize you with the instruction set of the 6510 microprocessor inside the Commodore 64. You'll also find chapter reviews, exercise sets, and glossaries. . .plus these important assembly language topics:

- Data transfer instructions
- Logic operations
- Branches and loops
- Shift and rotate instructions
- Input/Output
- Programming the 6581 sound interface device
- Applications using the 6567 video interface chip, and much more!

Contents

The Commodore 64 Microcomputer System • Writing, Assembling, and Executing Programs • Data Transfer Instructions—Using An Editor/Assembler • Arithmetic Operations in Assembly Language • Logic Operations in Assembly Language • Branches And Loops • Shift And Rotate Routines • Indexed Addressing Modes • Jumps, Subroutine Calls, And Interrupts • Programming The 6581 Sound Interface Device • Applications Using the 6567 Video Interface Chip • Input/Output: The 6526 Complex Interface Adapter • Appendix A: Decimal, Binary, And Hexadecimal Number Systems • Appendix B: The Computer Assisted Instruction Program • Appendix C: The 6510 Instruction Set Summary • Glossary • Index • Documentation for Diskette to Accompany *Assembly Language Programming with the Commodore 64*

Also Available...Optional Diskette

This diskette contains all the programs from the text, designed to save you the time and trouble of keyboarding. See the insert inside the book for ordering information.



ISBN 0-89303-319-7