Osborne McGraw-Hill

APPLE® ROOTS

ASSEMBLY LANGUAGE PROGRAMMING FOR APPLE® IIe AND APPLE® IIc

MARK ANDREWS

Apple® Roots:

Assembly Language Programming For the Apple® IIe & IIc

Apple® Roots:
Assembly Language Programming For the Apple® IIe & IIc

Mark Andrews

Osborne **McGraw-Hill**
Berkeley, California

Published by
Osborne **McGraw-Hill**
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations and book distributors
outside of the U.S.A., please write to Osborne **McGraw-Hill**
at the above address.

Apple® Roots:
Assembly Language Programming
For the Apple® IIe & IIc

1234567890 DODO 898765

ISBN 0-07-881130-9

Jonathan Erickson, Acquisitions Editor
Paul Jensen, Technical Editor
Michael Fischer, Technical Reviewer
Jessica Bernard, Copy Editor
Judy Wohlfrom, Text Design
Yashi Okita, Cover Design

To Muriel

Contents

Introduction

If your Apple doesn't understand you, maybe it's because you don't speak its language. Now we're going to break that language barrier.

This book will teach you how to write programs in assembly language—the fastest-running and most memory-efficient of all programming languages. It will give you a working knowledge of machine language, your computer's native tongue. It will enable you to create programs that would be impossible to write in BASIC or other less advanced languages. It also will prove to you that programming in assembly language is not nearly as difficult as you may think.

Many books have been written about assembly language, but this is the first assembly-language book to deal specifically with the Apple IIc and the Apple IIe, the two newest computers in the Apple II line. It is also the first book that explains how to write assembly-language programs using ProDOS, the Apple IIc/Apple IIe disk operating system that has now replaced its predecessor, DOS 3.3. The book also covers the advanced features of the 65C02 microprocessor, the new chip built into the Apple IIc that can also be installed in the Apple IIe.

In addition, this is the first assembly-language book that explains how to use three of the most popular assemblers for the Apple IIc and the Apple IIe: the *ProDOS Assembler Tools* package from Apple, the *Merlin Pro* assembler-editor system from Roger Wagner Publishing, Inc., and the ORCA/M assembler from The Byte Works of Albuquerque, New Mexico.

The Apple IIc and the Apple IIe offer a number of brand-new features that are of great importance to Apple programmers and potential Apple programmers. These features include an 80-column text display, double high-resolution graphics, and 64K of extra memory (all built into the Apple IIc and optional on the Apple IIe). Both the IIc and the IIe have expanded keyboards, including new function keys (OPEN APPLE and CLOSED APPLE keys) and new special-character keys. In addition, the Apple IIc has a built-in set of special characters designed for use with a mouse, and the same special characters are available on any Apple IIe equipped with plug-in mouse cards.

Both the Apple IIc and IIe are now being shipped with ProDOS, the new Apple II disk operating system that succeeds DOS 3.3. ProDOS is not just another revision of DOS 3.3; it is a completely new disk operating system that was designed specifically for the Apple IIc, the Apple IIe, and future computers in the Apple II line. ProDOS handles disk files and disk drives very differently from the way they were handled under DOS 3.3.

A point-by-point comparison between DOS 3.3 and ProDOS is beyond the scope of this introduction. However, please note that *there are so many differences between ProDOS and the systems it replaces that most assembly-language programs written under earlier disk operating systems will not work in a ProDOS environment.* This is the first book about writing assembly-language programs for the new ProDOS-equipped Apple IIc and Apple IIe computers.

Both the Apple IIc and the newest versions of the Apple IIe are now equipped with an advanced 8-bit microprocessor called the 65C02. The 65C02, a new member of the 6502 series of microprocessors, is designed to be programmed in standard 6502 assembly language. However, the 65C02 contains a number of new features. Along with the 56 instructions used in conventional 6502 assembly language, the 65C02 is equipped with several additional instructions. It also recognizes a number of addressing modes that were not available in earlier 6502-series microprocessors.

If you know BASIC—even a little BASIC—you can learn to program in assembly language. Once you know assembly language, you'll be able to

- Write programs that will run 10 to 1000 times faster than programs written in BASIC.
- Use up to 16 colors simultaneously in any Apple IIc or Apple IIe graphics mode—including double high-resolution graphics.

- Custom design your own screen displays, mixing text and graphics in any way you like.
- Create your own customized character sets.

Knowing assembly language can also enable you to create music and sound effects for Apple IIc/Apple IIe programs, write programs that will boot from a disk and run automatically when you turn your computer on, and intermix BASIC and assembly language in the same program, combining the simplicity of BASIC with the speed and versatility of assembly language.

In other words, once you learn how to program in assembly language, you will be able to start writing programs using the same techniques that professional programmers use. Many of those techniques are impossible without a knowledge of assembly language.

Finally, as you learn assembly language, you will be learning what makes computers tick. That will make you a better programmer in any language.

While teaching you assembly language, *Apple Roots* will provide you with a comprehensive collection of assembly-language routines that can be typed and assembled and then used in user-written assembly-language programs. It also contains a library of interactive tutorial programs that are especially designed to take the drudgery out of learning assembly language.

Chapter 1 is an easy-to-understand introduction to assembly language. The main feature of Chapter 2 is a clear explanation of binary numbers. In addition, Chapter 2 contains a series of four type-and-run BASIC programs that can convert numbers from one base to another.

In Chapter 3 you will learn about the 6502B/65C02 chip used in the Apple IIc and the Apple IIe. In Chapter 4, you'll start actually writing assembly-language programs. The rest of the book presents a number of advanced programming lessons and type-and-run assembly-language programs.

The first thing you need in order to use this book is an Apple IIc or Apple IIe computer equipped with a TV set or a computer monitor (preferably a color model) and at least one disk drive. A line printer is highly recommended but not essential. A game controller, a mouse, or both will also come in handy. So will a second disk drive.

The assembly-language programs in this book were written using three assemblers: the Apple ProDOS assembler, the Merlin Pro, and the ORCA/M. If you don't own one of those packages, it would be a good idea to buy one before starting this book. All of the programs in the

book were also written under ProDOS. If your Apple IIe was purchased before ProDOS was introduced, you will need to buy a ProDOS package from your Apple dealer and learn to use it.

One prerequisite for using the assembly-language lessons in this book is a basic understanding of ProDOS, which you can gain by reading a ProDOS manual. You should also have at least a fundamental knowledge of Applesoft BASIC or some other high-level programming language.

There are at least two other books that you should have before you start studying assembly language. The first of these books is the user's manual that came with your computer. The second is a reference manual for your computer. (Apple publishes separate reference manuals for the Apple IIc and the Apple IIe.) Other useful books include *Programming the 6502* by Rodnay Zaks, *Assembly Lines: The Book* by Roger Wagner, and *6502 Assembly Language Programming* by Lance A. Leventhal. These books, and others that may come in handy while you're studying assembly language, are listed in the Bibliography.

1

Breaking the Assembly Language Barrier

If you want to learn *assembly language*, you've opened the right book. With this volume and an Apple IIc or Apple IIe computer, you can start programming right now in *machine language*. Then we'll see how machine language relates to assembly language. Turn on your computer and type HI.TEST.BAS, the BASIC program listed in Program 1-1. Then run the program, and you'll immediately see how it got its name.

Program 1-1
HI.TEST.BAS
The HI.TEST Program (BASIC Version)

```
10  REM   *** HI.TEST.BAS ***
20  DATA  169,200,32,237,253,169,201,32,237,253,96
30  FOR L = 32768 TO 32778: READ X: POKE L,X: NEXT L
40  CALL 32768
```

Machine Language and Assembly Language

As you can see, Program 1-1 is written in BASIC. However, when you type the program and execute it, your computer will run a machine-language program.

Please note that this is machine language, not assembly language. As you'll see later in this chapter, machine language and assembly language are very closely related, but they are not exactly the same. Machine language is made up of numbers—nothing but numbers. Since "number-crunching" is what computers do best, machine language is ideal for a computer. In fact, machine language is the only language that a computer actually understands. No matter what language a program is originally written in, it must be converted into machine language before a computer can process it.

The main reason that assembly language is different from machine language is that it was designed for humans, not for machines. From the standpoint of both structure and vocabulary, assembly language is very similar to machine language. In fact, assembly language is not actually a programming language at all, but merely a notation system designed to make it easier to write programs in machine language.

Despite its structural similarity to machine language, however, a program written in assembly language looks quite different from a program written in machine language. Whereas machine language consists solely of numbers, assembly language uses three-letter abbreviations called *mnemonics*. It's therefore easier to write programs in assembly language than in machine language.

In one respect, though, assembly language is just like any other programming language: before an assembly-language program can be executed by a computer, it must be converted into machine language. For this reason, programs written in assembly language are often called *source-code* programs. And machine-language programs generated from source-code programs are often referred to as *object-code* programs.

Source-code programs are usually written with the aid of a special kind of software package called an *assembler-editor*, or simply an *assembler*. An assembler-editor package usually includes at least two kinds of utility programs: an assembly-language *editor*, which enables the user to write programs in assembly language, and an *assembler*, which can convert (or *assemble*) assembly-language programs into machine language.

Assembly language and machine language will be discussed in more detail later in this chapter.

How the HI.TEST.BAS Program Works

Now we're ready to take a closer look at the HI.TEST.BAS program shown in Program 1-1. The HI.TEST.BAS program begins with a title line. The next line in the program, line 20, is a line of data that equates to a series of machine-language instructions. Line 30 contains a loop that pokes the machine-language data in line 20 into a block of RAM (which we will define shortly) that extends from memory address 32768 to memory address 32778, or $8000 to $800A in hexadecimal notation. (A memory address—sometimes referred to as a *memory location* or *memory register*—is nothing but a number that can be used to pinpoint the location of any piece of data, or *byte*, stored in a computer's memory. There are 65,535 memory addresses in an off-the-rack Apple IIe, and there are 131,070 memory addresses in an Apple IIc or an Apple IIe equipped with an expanded 80-column card. More information on memory addresses will be provided later in this book, primarily in Chapter 11, which will focus specifically on the memory structure of the Apple IIc and the Apple IIe.) Finally, in line 40, there's a CALL instruction that executes the machine-language program that has just been loaded into memory.

To understand what your computer does when it receives the CALL instruction in line 40, it will help to have a basic understanding of the architecture of microcomputers and how your Apple processes a machine-language program.

Inside a Microcomputer

Every microcomputer can be divided into three parts:

- A *central processing unit (CPU).* A central processing unit, as its name implies, is the central component in a computer system, the component in which all computing functions take place. All of the functions of a central processing unit are contained in a *microprocessor unit* (or *MPU*). Your Apple computer's MPU—as well as its CPU—is a *large-scaled integrated circuit (LSI)* (a 6502B chip if you own an Apple IIe, and a 65C02 chip if you own an Apple IIc).

- A *memory.* Memory can be further divided into *RAM (random-access memory)* and *ROM (read-only memory).* These two types of memory are discussed in the following section.

- Some *input/output (I/O)* devices. Your computer's main input device is its *keyboard.* Its main output device is its *video monitor.* Other devices that your Apple can be connected to—or, in computer jargon, can be *interfaced* with—include telephone modems, graphics tablets, printers, and disk drives.

Figure 1-1 is a block diagram that illustrates the architecture of the Apple IIc and the Apple IIe. In this chapter we will not concern ourselves with the I/O. However, keyboard and screen I/O will be covered later, beginning with Chapter 8.

Your Apple's Memory

Figure 1-1 shows the two kinds of memory a computer has: random-access memory (RAM) and read-only memory (ROM). The important difference between them is that RAM can be modified, while ROM cannot. ROM is permanently etched into a bank of memory chips inside your Apple, so it's always there, whether the power to your computer is off or on. Every time you turn off your Apple, everything that you've

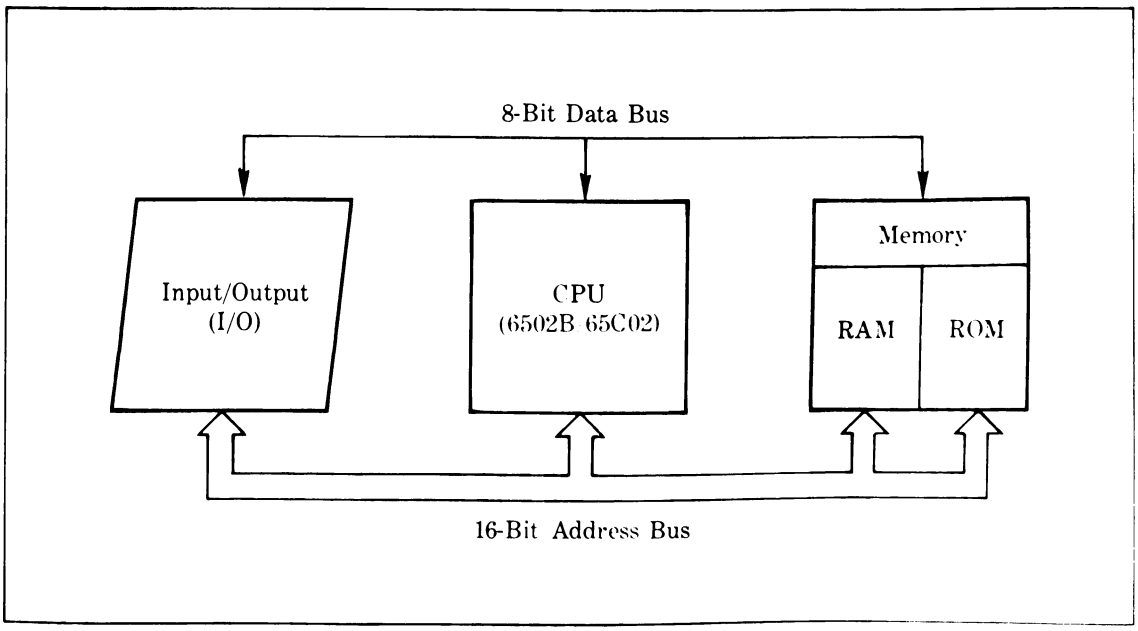

Figure 1-1. Block diagram of a microcomputer

stored in RAM immediately disappears. But everything in ROM remains and will spring back into action when you turn your computer on again.

The largest block of ROM in your Apple extends from memory address 53248 ($D000 in hexadecimal notation) to memory address 65535 ($FFFF in hexadecimal notation). A number of important programs are permanently situated in this block of ROM, including your computer's BASIC interpreter and its built-in machine-language monitor.

Machine-Language Programs in RAM In introductory books about computers, a bank of RAM is often compared to a bank of mailboxes built into a wall in a postal station. Each memory address in a RAM bank, like each mailbox in a tier of post office boxes, has an identifying number. And a computer program (like an ideal employee in a post office) can get to any of the memory addresses in a bank of RAM with equal ease. In other words, information stored in RAM can be retrieved at random. That's why RAM is called *random-access memory*.

What happens when your computer processes a machine-language program? Every machine-language program is made up of a series of numbers. When a machine-language program is loaded into a computer's memory, the numbers that make up the program are stored in a series of addresses in RAM. The starting address of the memory block in which the program is stored (known as the program's *origin* address) is usually stored in a special, predetermined memory location. Thus, when it is time to run the program, its starting address can be easily located.

Once a machine-language program has been loaded into RAM and its origin address has been stored in an accessible memory location, the program can be executed in several ways. For example, a machine-language program stored in an Apple II computer can be executed using a CALL instruction, a USR(X) instruction, a ProDOS dash (–) command, or a ProDOS BRUN command. These and other methods for running machine-language programs will be explained in Chapter 5.

Processing Executable Code When your computer goes to a memory location that has been identified as the starting address of a program, it should find the beginning of a block of executable code—that is, the beginning of a machine-language program. If it finds a program, it will carry out the first instruction in that program and then move on to the next consecutive address in its memory.

Your computer will keep repeating this process until it either

reaches the end of a program or encounters an instruction telling it to jump to another address.

Your Apple's CPU

In a microcomputer, a central processing unit (CPU) usually consists of a single microprocessor chip. Apple IIc and Apple IIe computers use either the 6502B chip or the 65C02.

The 6502B chip was designed for the Apple IIe and was originally built into all Apple IIe's. The 65C02 was designed for the Apple IIc and is the only microprocessor that has ever been used in the IIc. The 65C02 is now being built into all new Apple IIe's and is available as an optional, user-installable upgrade to older IIe's.

Both the 6502B chip and the 65C02 chip are improved and updated versions of an earlier chip, the 6502, developed by MOS Technology, Inc. The 6502 and chips based on it are used not only in Apple computers, but also in personal computers manufactured by Atari, Commodore, and several other companies.

The 6502B chip used in the Apple IIe is really just a faster-running version of the original 6502. But the 65C02 that's built into the Apple IIc and newer Apple IIe's has some extra capabilities that the old 6502 didn't have. In addition to being faster than the 6502, it uses less power, and it recognizes a number of instructions that the 6502 didn't understand. The 65C02 also has some additional addressing modes, a feature that will be explained in a later chapter.

For most purposes, however, the similarities among the 6502B, the 65C02, and the other chips in the 6502 family are more important than their differences. Although the model numbers of 6502-series chips may sometimes get confusing, all of the chips in the 6502 family are designed to be programmed using the same assembly-language dialect generically known as 6502 assembly language. Once you learn how to write programs in 6502/65C02 assembly language, you'll be able to program many different kinds of personal computers in addition to your Apple, including many manufactured by Atari and Commodore.

Even more important, the *principles* used in Apple assembly-language programming are universal; they're the same principles that all assembly-language programmers use, no matter what kinds of computers they're writing programs for. Once you learn 6502/65C02 assembly language, therefore, you can easily learn to program other kinds of chips, such as the Z80 chip used in Radio Shack and CP/M-based computers, and even the powerful newer chips used in 16-bit and 32-bit microcomputers such as the Apple Macintosh and the IBM PC.

Compilers, Interpreters, and Assemblers

Now that you have a basic understanding of what your Apple is made of and how it works, we're ready to take a closer look at the relationship among the three categories of computer languages: machine language, assembly language, and high-level languages.

High-level languages did not get their name because they're particularly esoteric or profound. They're called high-level languages merely because they're several levels removed from machine language, your computer's native tongue.

There are hundreds, perhaps thousands, of high-level languages, but most of them have at least one feature in common: they all bear at least a passing resemblance to English. BASIC, for example, is made up almost completely of instructions—such as PRINT, LIST, LOAD, SAVE, GOTO, and RETURN—that are derived from English words. Most other high-level languages also have instruction sets based largely upon plain-language words and phrases.

But computers can't understand a word of English; the only language they can understand is machine language, which is composed only of numbers. For this reason, a program written in any other language has to be translated into machine language before a computer can understand it. As mentioned previously, people who write programs in assembly language usually use special software products called assemblers to convert assembly language programs to machine language. Similarly, people who program in high-level languages use special kinds of software packages called *interpreters* and *compilers* to help them translate the programs they have written into machine language.

Interpreters and Compilers

The most important difference between interpreters and compilers is that an interpreter is designed to convert a program into machine language every time the program is run, while a compiler is designed to convert a program into machine language only once. When you write a program using an interpreter, you can store the program on a disk in its original form; your interpreter will automatically convert it into machine code every time you run it. But when a program is written using a compiler, it has to be converted into machine language and then stored on a disk as a machine-language program. Then it can be run like any other machine-language program, without any further need for a compiler.

Interpreters are easier to use than compilers because they're designed to be "transparent" to the user; that is, they are so unobtrusive that you're not even aware they're there. The BASIC utility that's built into your Apple IIc or Apple IIe is an interpreter, and using it will show you how transparent a BASIC interpreter can be. When you write a program in Applesoft BASIC, your computer's built-in interpreter translates every line of code that you write, as you write it, into a special kind of language called a *tokenized* language. Then, each time the program is run, it is translated into machine language.

This is a very roundabout way to run a program, and it's one factor that makes BASIC a rather slow-running language. But the process does work quite smoothly; if you've ever run an Applesoft BASIC program, you probably never even noticed the process of BASIC-to-machine-language translation.

One advantage of interpreters over compilers is that they can check each line of a program for obvious errors as soon as the line is written. If they don't check each line, they usually do spot errors as soon as the program is run. The errors can then be fixed on the spot. Compilers are less interactive. Most compilers can't check a program for errors until the program has been compiled. After an error is found and fixed, the program must be compiled again.

Compilers do have one significant advantage over interpreters: they produce faster-running programs. When a program is written using an interpreter, it has to be processed through the interpreter every time it's run. But a compiler has to do its job just once and never has to be used when a program is actually running.

Assemblers and Assembly Language

Assembly language, as we have seen, is neither an interpreted language nor a compiled language. Converting an assembly-language program into machine language requires an assembler-editor (also referred to as an assembler).

Because of the close relationship between machine language and assembly language, an assembler does not have nearly as difficult a task as an interpreter or a compiler. Each time an interpreter or compiler converts an instruction into machine language, a block—sometimes a very large block—of machine code has to be generated. But an assembler has to translate only one instruction at a time. The instructions used in assembly language perform much simpler functions than the instructions used in most high-level languages, so source-code programs written in assembly language tend to be much longer than similar programs

written in high-level languages. However, the instruction set used in assembly language is extremely versatile, since the mnemonics can be combined in an almost endless variety of ways.

Assembly language is also extremely memory-efficient, because assembly-language programs are created by human programmers, not churned out robotically by electromechanical code-generating machines. When a program is written in a high-level language, the result is usually a series of machine-language routines that are rather mindlessly strung together, one after another, like clothes hanging on a line. The interpreter or compiler that translates such a program into machine code typically repeats the same sequence of code again and again, usually wasting both memory and processing time.

In contrast, a good assembly-language programmer will usually write an important block of code just once during the course of a program. From that point on, it will be used as a subroutine, which conserves memory and shortens processing time.

Machine Language: A Language of Numbers

To understand what happens when an assembly-language program is assembled into a machine-language program, it helps to identify some of the differences and similarities between machine language and assembly language.

Machine language, in its purest form, is composed of *binary numbers*—numbers written as strings of 1's and 0's. Program 1-2 shows what the HI.TEST program presented at the beginning of this chapter looks like when it's written as a pure machine-language program in binary numbers.

Program 1-2
HI.TEST.BIN
The HI.TEST Program (Binary Code Version)

```
10101001
11001000
00100000
11101101
11111101
10101001
11001001
00100000
11101101
11111101
01100000
```

As you can see, it wouldn't be easy to write a program in binary numbers. Fortunately, you'll seldom have to. Although your computer sees every machine-language program as a string of binary numbers, nobody actually writes programs in binary notation. Instead, listings of machine-language programs are usually written in a notation system called the *hexadecimal system.*

In spite of appearances, hexadecimal numbers are closely related to binary numbers. Hex numbers are written not as strings of 1's and 0's, but as ordinary arabic numerals, with the letters A through F thrown in to express some extra values. Hex numbers are written as combinations of letters and numbers because, unlike ordinary decimal numbers, they aren't based on the value 10. Instead, they're based on the value 16, and the letters A through F are used to represent the values 10 through 15. You'll learn more about the hexadecimal system—and why it's used in assembly-language programs—beginning with Chapter 2. In the meantime, look at Program 1-3 to see what the HI.TEST program looks like written in hexadecimal numbers.

Program 1-3
HI.TEST.HEX
The HI.TEST Program (Hexadecimal Version)

```
A9 C8
20 ED FD
A9 C9
20 ED FD
60
```

The hex numbers in Program 1-3 have exactly the same values as the binary numbers that you saw in the binary version of the program. You may not yet understand what the instructions in the program mean, but you can see that the hexadecimal version of the program *looks* a little more comprehensible than the binary version. (It's easier to type into the computer, too!)

Assembly Language: A Language of Mnemonics

You now know that assembly language is very closely related to machine language. But the relationship between assembly language and machine language is not always obvious at first glance. Assembly language, unlike machine language, is written using three-letter instructions called *mnemonics.* To the casual observer, then, assembly language doesn't look a thing like machine language.

For every three-letter instruction used in assembly language, there

is a numeric instruction that means exactly the same thing in machine language. In other words, there is generally a one-to-one correlation between the mnemonics used in an assembly-language program and the numeric instructions used in the machine-language version of the same program.

This relationship between machine language and assembly language makes it easy to convert a machine-language program into assembly language or to convert an assembly-language program into machine language. Simply change each instruction to the equivalent instruction of the other language. Program 1-4 shows the close relationship between machine language and assembly language.

Program 1-4
HI.TEST.ASM
The HI.TEST Program (Assembled Version)

```
COL. 1      COL. 2          COL. 3
LINE NO.    OBJECT CODE     SOURCE CODE

1           A9 C8           LDA   #$C8
2           20 ED FD        JSR   $FDED
3           A9 C9           LDA   #$C9
4           20 ED FD        JSR   $FDED
5           60              RTS
```

Object Code and Source Code

If you look carefully at Columns 2 and 3 of Program 1-4, you'll see close similarities. For reasons that will become clear later, the letters and numbers in Column 2 are arranged slightly differently from those in Column 3, but certain patterns are the same in both columns. In Column 2, for example, the machine-language instruction A9 is used twice: once in line 1 and again in line 3. In Column 3, the assembly-language mnemonic LDA is also used twice—on the same lines and in the same positions as the machine-language instruction A9. Apparently, the object-code instruction A9 equates to the source-code instruction LDA. As it turns out, that's true.

Refer once more to the object-code listing in Column 2; you'll see that the machine-code instruction 20 is also used twice. In both cases, it is the machine-code equivalent of the source-code instruction JSR.

Now you've had a first-hand look at how assembly language compares with machine language. Later in this chapter, we'll discuss the similarities and differences between machine language and assembly language in greater detail. First, though, let's examine Program 1-5, a listing of the assembly-language version of the HI.TEST program.

Program 1-5
HI.TEST.S
The HI.TEST Program (Source-Code Version)

```
LDA    #200
JSR    $FDED
LDA    #201
JSR    $FDED
RTS
```

What the HI.TEST.S Program Does

As you can see, HI.TEST.S is a very short and simple assembly-language program. It contains only three mnemonics—LDA, JSR, and RTS—and three numbers—the hexadecimal number FDED and the decimal numbers 200 and 201. The number 200 is a screen-display code that equates to the letter H. The number 201 is a display-code number for the next letter in the alphabet, the letter I. And the hexadecimal number FDED (65005 in decimal notation) is the starting address of a handy machine-language subroutine (built into your Apple) that will print a character on the screen.

In the HI.TEST.S program, the numbers 200 and 201 are preceded by the symbol "#", and the hex number FDED is preceded by the symbol "$". In 6502/6502B/65C02 assembly language generally, when the symbol "#" precedes a number, it means that the number is to be interpreted as a literal number, not as a memory address. In the HI.TEST.S program, if the numbers 200 and 201 were not preceded by the symbol "#", they would be interpreted as addresses in your computer's memory. Since they do have a "#" prefix, however, they are interpreted as actual numbers.

The other special symbol in the HI.TEST.S program—the dollar sign in front of the number FDED—is an assembly-language prefix for hexadecimal numbers. If you're familiar with hexadecimal notation, you can probably tell by looking at the number FDED that it's a hexadecimal number. But sometimes decimal numbers and hex numbers look exactly alike. Therefore, in the HI.TEST.S program, the "$" symbol is used to show that the number $FDED is to be treated as a hexadecimal number.

In Apple IIc/IIe assembly language, it is possible to use both the symbol "#" and the symbol "$" in front of the same number (as long as the "#" comes first). Please note, however, that the symbol "#" is not used in front of the number $FDED in the HI.TEST.S program because in

this program, $FDED should be interpreted as a memory address, not as a literal number. In your Apple, as mentioned previously, $FDED is the memory address of a built-in subroutine that prints a character on the screen. That is the subroutine called in lines 2 and 4 of the HI.TEST.S program.

Before we can understand how the HI.TEST.S program works when assembled into machine language, we need to take a closer look at your computer's main microprocessor: its 6502B or 65C02 chip.

The 6502B/65C02 chip is the heart—or, more accurately, the brain—of your computer. The 6502B/65C02 is a very complex chip, but it has only seven main components: an *arithmetic logical unit*, or *ALU*, and six *internal registers*. The functions and features of all of these components will be covered in later chapters. To help you understand how the HI.TEST.S program works, though, here's a sneak preview of a very special 6502B/65C02 register, called the accumulator.

The accumulator is the busiest register in the 6502B/65C02 chip. Before any mathematical or logical operation can be performed on a number in 6502B/65C02 assembly language, the number has to be loaded into the accumulator. The assembly-language instruction that is usually used to load a number into the accumulator is LDA.

Let's look at line 1 of the HI.TEST.S program.

```
LDA #200
```

In this line, the statement "LDA #200" means "Load the accumulator with the literal number 200." In the world of computer programming, a number can be used to represent many different things. In the HI.TEST.S program, the number 200 represents the letter "H". Here's why.

A special system called ASCII is often used to encode letters, numbers, and special characters in computer programs. In the ASCII system, each letter, number, and special symbol on the typewriter keyboard has a number that can represent it in programs. Over the years, the ASCII system has become more or less standardized in the computer industry. However, since Apple computers make use of inverse video, flashing video, and other special effects, Apple uses a modified ASCII system. In the Apple system, the number 200 is an ASCII code for the letter "H", displayed as a capital letter in normal video. What line 1 of the HI.TEST.S program really means, then, is "Load the accumulator with the modified ASCII code for an uppercase 'H' ".

```
JSR $FDED
```

In 6502B/65C02 assembly language, the mnemonic JSR means "Jump to subroutine." This instruction is used in much the same way as the GOSUB instruction is used in BASIC. When the mnemonic JSR is used in an assembly-language program, it causes the program to jump to a subroutine that is expected to start at the memory address that follows the JSR instruction.

In assembly language, the mnemonic JSR is usually used along with another mnemonic, RTS, which means "Return from subroutine."

The RTS instruction also corresponds to a BASIC instruction, RETURN. When a JSR instruction is encountered in an assembly-language program, the address of the very next instruction in the program is first placed in an easily accessible location in a special block of memory called a *stack*. Then the program jumps to whatever address follows the JSR instruction. This address is usually the starting address of a subroutine.

When a subroutine is called with a JSR instruction, the subroutine usually ends with an RTS instruction. When that RTS instruction is reached, any address that has been placed on the stack by a JSR instruction is retrieved. The program then returns to that address and processing of the main body of the program resumes.

Line 2 of the HI.TEST.S program should now be clear. The statement "JSR $FDED" means "Jump to the subroutine that begins at memory address $FDED." This subroutine takes whatever screen code is stored in the accumulator and automatically displays the corresponding character on the screen. Then it returns control to the program in progress—in this case, the HI.TEST program.

A number of handy I/O routines similar to this one are built into the Apple IIc and the Apple IIe. We'll be using quite a few of them in this book.

```
LDA #201
```

In Apple IIc/IIe assembly language, the number 201 is an Apple ASCII code for a normal capital "I". In the HI.TEST.S program, then, the statement "LDA #201" means "Load the accumulator with the Apple ASCII code for an uppercase 'I'."

```
JSR $FDED
```

This statement is identical to the statement in line 2. It means "Jump to the subroutine that starts at memory address $FDED." This time, however, since the accumulator has been loaded with the value 201, the subroutine that starts at $FDED will cause an "I" to be displayed on the screen.

```
RTS
```

When an RTS instruction is used to terminate a subroutine, it usually causes a program to jump back to where it was before the subroutine was called. In this case, however, RTS is used to terminate a whole program, not just a subroutine. When RTS is used to terminate a complete program, it usually returns control of the computer to whatever program or system was in control before the terminated program began. If you were to call the HI.TEST program from BASIC, then, the RTS instruction in line 5 would transfer control to your computer's BASIC interpreter.

Two Additional Programs

Quite a bit of ground has been covered in this introductory chapter. We've taken a look at the overall architecture of microcomputers in general, and the Apple IIc and IIe in particular. We've compared assembly language with various high-level languages, and we've discussed the ways in which assembly language and high-level languages are translated into machine language. We've compared decimal, hexadecimal, and binary numbers, and we've seen how hexadecimal numbers are used in assembly-language programs. We've peeked inside your Apple's central microprocessor, we've seen how source code is assembled into object code, and we've observed how your computer steps through its memory as it processes a machine-language program. We've even made a line-by-line analysis of a short assembly-language program and seen how a machine-language program can be called from BASIC without having to be processed through an assembler.

Now let's take a look at another BASIC program that makes use of a little assembly language. Program 1-6 is called FLASH.BAS. Type it and run it, and you'll see an interesting display on your computer screen.

Program 1-6
FLASH.BAS
A BASIC Program for Flashing a Message on the Screen

```
10  REM     *** FLASH.BAS ***
20  TEXT : HOME
30  PRINT : POKE 49164,1: POKE 49166,1: POKE 50,127
40  PRINT : PRINT "FLASH! APPLE OWNER BREAKS MACHINE CODE!"
50  GOTO 50
```

How FLASH.BAS Works

If you've done much programming in BASIC, you've probably seen—and may have even written—BASIC programs that produce flashing text displays like the one in FLASH.BAS. Nevertheless, this little BASIC program is unusual because it doesn't use BASIC instructions to generate its flashing 40-column screen display. Instead, it does its job with a series of well-placed POKE commands.

In the FLASH.BAS program, POKE commands are used to insert numbers directly into three memory addresses that your computer uses to generate screen displays.

In line 30, the statement POKE 49164,1 puts your Apple into its 40-column mode. In the same line, the statement POKE 49166,1 makes sure that your computer's main character set is turned on and that an *alternate character set* that it also has access to is turned off.

The last statement in line 30—POKE 50,127—is the statement that turns on your computer's flashing mode. Then your Apple is ready to print the flashing message that appears in line 40. Finally, in line 50, the program winds up in an endless loop that prevents anything else from being displayed on the screen.

A Program for Displaying Mouse Icons

Before we move on to Chapter 2, here's one more program, presented especially for owners of Apple IIc's and late model Apple IIe's. It isn't an assembly-language program or a machine-language program, and it doesn't even include any machine-language instructions. But it will probably interest you if you're an Apple IIc owner, and an assembly-language version of the program will be presented later on in this volume.

Here's how the program works. The Apple IIc/IIe has two character sets—a standard character set and an alternate character set. But the alternate character sets built into Apple computers vary from model to model. If your computer is an Apple IIc, or an Apple IIe with built-in mouse ROMs, its built-in character set includes 32 mouse *icons* (special graphics characters designed for use with the Apple IIc mouse). In the Apple IIc and the current-model Apple IIe, mouse icons are what you get when your computer is in uppercase and in its flashing mode. But if you own an Apple IIe and haven't had special mousetext ROM installed, then you'll get just what you'd ordinarily expect in an uppercase flashing print mode: uppercase flashing characters.

If your computer is an Apple IIc or a fairly new Apple IIe, you can display all of your mouse icons on the screen with this BASIC program.

Type the MOUSETEXT.BAS program, run it, and enjoy!

Program 1-7
MOUSETEXT.BAS
A BASIC Program That Displays Mouse Icons

```
10  REM     *** MOUSETEXT.BAS ***
20  PRINT  CHR$ (4);"PR#3": REM  TURN ON ENHANCED VIDEO
    FIRMWARE
30  PRINT  CHR$ (27); CHR$ (17): REM  ESC/CONTROL-Q (SET
    40-COL MODE)
40  PRINT  CHR$ (15): REM  CONTROL-O (SET REVERSE MODE)
50  PRINT  CHR$ (27): REM   ESCAPE KEY (TURN ON MOUSETEXT)
60  FOR L = 64 TO 95: PRINT  CHR$ (L);: NEXT L: REM  PRINT
    MOUSE ICONS
70  PRINT  CHR$ (24): REM  CONTROL-X (TURN OFF MOUSETEXT)
80  PRINT  CHR$ (14): REM  CONTROL-N (DISPLAY NORMAL
    CHARACTERS)
```

2

Number Systems

Most people are accustomed to using only decimal numbers, which are based on the digits 1 through 10. But at some time, you may have also encountered the roman numeral system, which uses letters to represent numbers. There are many other numeric systems that are different from the decimal system, such as the Chinese system, the Hebrew system, and the Sanskrit system.

In the world of computer programming, three numeric systems are commonly used. They are as follows:

- The decimal system, which is based on the value 10 and is written using the digits 0 through 9.
- The binary system, which is based on the value 2 and is written using only two digits: 0 and 1.
- The hexadecimal system, which is based on the value 16 and is written using the digits 0 through 9 plus the letters A through F.

Number-Base Prefixes

When a binary number appears in an Apple IIc or IIe assembly-language program, the prefix "%" is used by most assemblers to distinguish it from a decimal or hexadecimal number. When a hexadecimal number appears in an Apple assembly-language program, the prefix "$" is used by most assemblers to indicate that it is a hexadecimal number.

No special prefix is used in front of a decimal number; if a number with no prefix appears in a program, it is presumed to be a decimal number.

The following illustration shows how prefixes are used to distinguish among binary, hexadecimal, and decimal numbers in assembly-language programs.

%1101	The binary number 1101 (decimal 13)
$1101	The hexadecimal number 1101 (decimal 4353)
1101	The decimal number 1101

Using Binary Numbers

A computer can understand only one language: a language that is made up solely of numbers and is called machine language. Machine language, at its most basic level, consists of binary numbers (1's and 0's). Before data can be loaded into a computer, it must somehow be converted into strings of 1's and 0's.

In the binary notation system, the 1's and 0's that make up binary numbers are known as *bits*. A series of four bits is called a *nibble* (or nybble), a series of eight bits is called a *byte*, and a series of 16 bits is usually called a *word* (although there are 8-bit words too).

When the bits and bytes that make up a machine-language program are processed by a computer, they are converted into strings of on-and-off electrical pulses. Inside a computer, these on-and-off pulses cause the current flowing through various electrical lines to fluctuate between low and high levels. When the electrical current falls below a certain predetermined level, the switch is considered off, and its state is represented as a 0 in the binary notation system. When the level of the current rises above a certain level, the switch is considered on, and its state is represented as a 1.

Now we're going to examine a series of 8-bit binary numbers. Look at the numbers in this list closely, and you'll see that every binary number that ends in 0 is twice as large as the previous number.

```
00000001 =   1
00000010 =   2
00000100 =   4
00001000 =   8
00010000 =  16
00100000 =  32
01000000 =  64
10000000 = 128
```

Now here are two more numbers that are noteworthy, but for completely different reasons:

```
%11111111 = 255
%11111111 11111111 = 65,535
```

The number %11111111, or 255, is noteworthy because it's the largest 8-bit number. And the number %11111111 11111111, or 65,535, is the largest 16-bit number. (The space in the middle of the number %11111111 11111111 was put there so the number would be easier to read. Spaces are often inserted in the middle of 8-bit numbers for the same reason. Sometimes, for example, you might see the binary number 11111111 written 1111 1111.)

The Hexadecimal Number System

Since computers "think" in binary numbers, the binary system is obviously an excellent notational system for representing computer data. But binary numbers have one serious shortcoming: they're extremely difficult to read. Thus the binary system is not the numeric system that is most often used in assembly-language programming. The numeric system that you'll encounter most often in assembly-language programming is the *hexadecimal system.*

Just as binary numbers are based on the value 2, hexadecimal numbers are based on the value 16.

Hexadecimal numbers are often used in assembly-language programming because they can help bridge the gap between the binary

and decimal systems. Since binary numbers have a base of 2 and hex numbers have a base of 16, a series of four binary bits can always be translated into one hexadecimal digit. So a series of eight bits (a byte) can always be represented by a pair of hexadecimal digits, and a series of 16 bits (a word) can always be represented by a four-digit hexadecimal number.

In Table 2-1, the decimal, hexadecimal, and binary numbers from 1 to 16 are compared. Examine the chart closely, and you'll see that odd-looking letter and number combinations like "FC1C", "5DA4", and even "ABCD" are perfectly good numbers in the hexadecimal system.

As you can see from Table 2-1, the decimal number 16 is written "10" in hex and "00010000" in binary and is thus a round number in both the binary and hexadecimal systems. The hexadecimal digit F, which comes just before hex 10, is written 00001111 in binary.

As you become more familiar with the binary and hexadecimal systems, you will begin to notice many other similarities between them. For example, the decimal number 255 (the largest 8-bit number) is 11111111 in binary and FF in hex. The decimal number 65,535 (the highest memory address in a 64K computer) is written FFFF in hex and 11111111 11111111 in binary.

Table 2-1. Comparing Decimal, Hexadecimal, and Binary Numbers

Decimal	Hexadecimal	Binary
1	1	00000001
2	2	00000010
3	3	00000011
4	4	00000100
5	5	00000101
6	6	00000110
7	7	00000111
8	8	00001000
9	9	00001001
10	A	00001010
11	B	00001011
12	C	00001100
13	D	00001101
14	E	00001110
15	F	00001111
16	10	00010000

Converting Numbers From One System to Another

Since hexadecimal numbers, decimal numbers, and binary numbers are all used in assembly-language programming, it would obviously be handy to have some kind of tool that could be used to convert numbers back and forth among these three numeric systems. Fortunately, a number of such tools are available. Here are a few.

Software-Based Converters

The machine-language monitor built into your Apple includes a decimal-to-hexadecimal converter and a hexadecimal-to-decimal converter. So do the Merlin Pro assembler and the Bugbyter debugging utility, which comes with the Apple ProDOS assembler. For more details on these utilities, see the Apple IIc and IIe technical reference manual and the manuals that come with the Merlin and Apple ProDOS assemblers.

Programmers' Calculators

Texas Instruments makes an extremely useful calculator called the *Programmer*, which can perform decimal-to-hexadecimal conversions in a flash and can also add, subtract, multiply, and divide both decimal and hexadecimal numbers. Many assembly-language program designers use the TI Programmer or a similar calculator and wouldn't dream of trying to get along without it.

Charts and Tables

Many books on assembly language contain charts that you can consult when you convert numbers from one notation system to another. You'll find a few such charts in this chapter, and you'll also find something much better: a series of BASIC programs that will automatically perform decimal-to-hexadecimal, decimal-to-binary, and binary-to-hexadecimal conversions.

Let's start with a program that will convert binary numbers to decimal numbers.

Converting Binary Numbers To Decimal Numbers

It isn't difficult to convert a binary number to a decimal number. In a binary number, as we've seen, the bit farthest to the right represents 2

Table 2-2. Values of the Bits in an 8-Bit Binary Number

Bit 0 =	2 to the	0th	power = 1
Bit 1 =	2 to the	1st	power = 2
Bit 2 =	2 to the	2nd	power = 4
Bit 3 =	2 to the	3rd	power = 8
Bit 4 =	2 to the	4th	power = 16
Bit 5 =	2 to the	5th	power = 32
Bit 6 =	2 to the	6th	power = 64
Bit 7 =	2 to the	7th	power = 128

to the power 0. The next bit to the left represents 2 to the power 1, the next represents 2 to the power 2, and so on.

The digits in an 8-bit binary number are therefore numbered 0 to 7, starting from the far-right digit. The far-right bit—often referred to as bit 0—represents 2 to the 0th power, or the number 1. And the far-left bit—often called bit 7—equals 2 to the 7th power, or 128.

Table 2-2 is a list of simple equations that illustrate what each bit in an 8-bit binary number means.

Table 2-3 provides an easy method of converting any 8-bit binary number into its decimal equivalent. Instead of writing the number from

Table 2-3. Converting a Binary Number Into a Decimal Number

1 ×	1 =	1
0 ×	2 =	0
0 ×	4 =	0
1 ×	8 =	8
0 ×	16 =	0
1 ×	32 =	32
0 ×	64 =	0
0 ×	128 =	0
Total	=	41

left to right, write it instead in a vertical column, with bit 0 at the top of the column and bit 7 at the bottom. Then multiply each bit in the binary number by the decimal number that it represents. Add the results of all of these multiplications, and the total will be the decimal value of the binary number.

Suppose, for example, you wanted to convert the binary number 00101001 into a decimal number. Table 2-3 shows how the conversion could be done.

If the calculation in Table 2-3 is correct, the binary number 00101001 should equal the decimal number 41. Look up either 00101001 or 41 on any binary-to-decimal or decimal-to-binary conversion chart, and you'll see that the calculation was accurate. This conversion technique will work with any other binary number.

Converting Decimal Numbers To Binary Numbers

Now we'll go in the opposite direction and convert a decimal number to a binary number.

First, divide the number by 2. Write down both the quotient and the remainder. Since we're dividing by 2, the remainder will be either a 1 or a 0. We will therefore write down the quotient followed by either a 1 or a 0.

Next, we'll take the quotient, divide it by 2, and write down the result of that calculation. If there's a remainder (a 1 or a 0), we'll also write that below the first remainder.

When we are left with no more numbers to divide, we'll write all of the remainders that we have, reading from the bottom to the top. Then we'll have a binary number—a number made up of 1's and 0's. That number will be the binary equivalent of the original decimal number.

This conversion technique is illustrated in Table 2-4.

To complete the decimal-to-binary conversion illustrated in Table 2-4, simply copy the binary digits in the right-hand column, writing them horizontally from right to left with the top digit on the right. You'll then see that the binary equivalent of the decimal (not hexadecimal) number 117 is 01110101. If you have a decimal-to-binary conversion chart handy, you can use it to confirm the accuracy of this calculation.

Table 2-4. Converting a Decimal Number Into a Binary Number

```
117/2 = 58 with a remainder of 1
 58/2 = 29 with a remainder of 0
 29/2 = 14 with a remainder of 1
 14/2 =  7 with a remainder of 0
  7/2 =  3 with a remainder of 1
  3/2 =  1 with a remainder of 1
  1/2 =  0 with a remainder of 1
  0/2 =  0 with a remainder of 0
```

Converting Binary Numbers To Hexadecimal Numbers

Here's an even easier number-base conversion. To convert hexadecimal numbers to their binary equivalents and vice versa, merely use the chart in Table 2-5.

To convert a multiple-digit hex number into a binary number, all you need do is string the letters and digits in the hex number together and convert each one separately, as shown in Table 2-5. For example, the binary equivalent of the hexadecimal number C0 is 1100 0000. The binary equivalent of the hex number 8F2 is 1000 1111 0010. The binary equivalent of the hex number 7A1B is 0111 1010 0001 1011. And so on.

To convert binary numbers to hexadecimal numbers, use the chart in reverse. The binary number 1101 0110 1110 0101, for example, is equivalent to the hexadecimal number D6E5.

Converting Decimal Numbers To Hexadecimal Numbers

It's almost as simple to convert decimal numbers to hexadecimal numbers as it is to translate binary numbers to decimal numbers.

First, take a decimal integer that you want to convert and divide it by 16. Write down the remainder, like this:

64540/16 = 4033 with a remainder of 12

Table 2-5. Hexadecimal-to-Binary Conversion Chart

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

Divide the integer part of the quotient by 16 and write down the result of that calculation.

4033/16 = 252 with a remainder of 1

Keep repeating this process until you have a quotient of 0. Here's the entire set of calculations that are needed to convert the decimal number 64540 into a hexadecimal number:

64540/16 = 4033 with a remainder of 12
4033/16 = 252 with a remainder of 1
252/16 = 15 with a remainder of 12
15/16 = 0 with a remainder of 15

When you've finished this series of calculations, you must convert any remainder that's greater than 9 into its hexadecimal equivalent. In this problem, three remainders are greater than 9: the value 12 in the first line, the value 12 in the third line, and the value 15 in the fourth line. The decimal number 12 equates to the letter C in hexadecimal

notation, and the decimal number 15 equates to the letter F. So the remainders in the problem, converted into hex, are

C
1
C
F

Read the four numbers, starting from the bottom, and you have the hexadecimal number FC1C, which is the number we're looking for, the hexadecimal equivalent of the decimal number 64540.

Doing It the Easy Way

In this chapter, we've compared three different number bases: the decimal system, the hexadecimal system, and the binary system. Now you know how to convert numbers from any of these three bases to any other. Some of the conversion techniques are quite simple; others are fairly complicated, and unless you have a photographic memory, you may not remember any of them. But fortunately, you won't have to. Just type and save the program presented in Program 2-1, and you can let your computer do it for you.

Program 2-1, titled "By the Numbers," is a menu-driven BASIC program that can convert numbers from any of the three bases discussed in this chapter to any other. The program assumes that you have an 80-column display.

In the next chapter, we'll take a look inside your Apple's main microprocessor and see what makes it tick. Then we'll be ready to start actually writing some programs in assembly language.

Program 2-1
BY THE NUMBERS
(A BASIC Number-Conversion Program)

```
10  REM  ************************************************
20  REM  ***************** BY THE NUMBERS ***************
30  REM    ********* A NUMBER CONVERSION PROGRAM *********
40  REM    ************************************************
50  PRINT  CHR$ (4);"PR#3"
60  DIM HEX$(8),BIT$(8),H$(16),B$(16),TEMP$(2),BIT(8)
70  DATA  0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
80  DATA  0000,0001,0010,0011,0100,0101,0110,0111
90  DATA   1000,1001,1010,1011,1100,1101,1110,1111
100  FOR L = 1 TO 16: READ H$(L): NEXT L
110  FOR L = 1 TO 16: READ B$(L): NEXT L
```

```
120  TEXT : HOME
130  PRINT : PRINT "          BY THE NUMBERS: A NUMBER-BASE
    CONVERSION PROGRAM"
140  PRINT "        COPYRIGHT (C) 1985, MARK ANDREWS": PRINT
150  PRINT : PRINT "THIS PROGRAM WILL CONVERT:"
160  PRINT : PRINT "   (A) DECIMAL NUMBERS TO HEXADECIMAL
    NUMBERS"
170  PRINT "   (B) HEXADECIMAL NUMBERS TO DECIMAL NUMBERS"
180  PRINT : PRINT " (C) BINARY NUMBERS TO DECIMAL NUMBERS"
190  PRINT "   (D) DECIMAL NUMBERS TO BINARY NUMBERS"
200  PRINT : PRINT "   (E) HEXADECIMAL NUMBERS TO BINARY
    NUMBERS"
210  PRINT "   (F) BINARY NUMBERS TO HEXADECIMAL NUMBERS"
220  PRINT : INPUT "WHAT KIND OF CONVERSION DO YOU WANT?
    (TYPE A-F): ";A$
230  IF A$ = "" THEN 220
240  IF  LEN (A$) <  > 1 THEN 220
250  IF A$ < "A" OR A$ > "F" THEN 220
260 A =  ASC (A$) - 64: REM  TRANSLATE A$ INTO AN INTEGER
    FROM 1 (A) TO 6 (F)
270  ON A GOTO 290,490,660,870,1040,2120
280  REM  ********* DECIMAL-HEXADECIMAL CONVERSION *******
290  TEXT : HOME : PRINT "DECIMAL-TO-HEXADECIMAL CONVERSION
    (RANGE: 0 TO 99999999)"
300  PRINT : PRINT "TYPE A POSITIVE DECIMAL INTEGER (OR 'M'
    FOR MENU):"
310  PRINT : INPUT "DEC: ";A$
320  FOR L = 1 TO 8:HEX$(L) = "": NEXT L
330  IF A$ = "M" THEN 120
340  FOR L = 1 TO 8:T$ =  RIGHT$ (A$,L)
350  IF  ASC (T$) < 48 OR  ASC (T$) > 57 THEN 310
360  NEXT L
370  IF  LEN (A$) < 1 OR  LEN (A$) > 8 THEN 310
380 N =  VAL (A$)
390 I = 8
400 TMP = N:N =  INT (N / 16)
410 TMP = TMP - N * 16
420  IF TMP < 10 THEN HEX$(I) =  RIGHT$ ( STR$ (TMP),1):
    GOTO 440
430 HEX$(I) =  CHR$ (TMP - 10 +  ASC ("A"))
440  IF N <  > 0 THEN I = I - 1: GOTO 400
450  PRINT "HEX: ";
460  FOR L = 1 TO 8: PRINT HEX$(L);: NEXT L: PRINT
470  GOTO 310
480  REM  ************ HEXADECIMAL-TO-DECIMAL CONVERSION
    *********************
490  TEXT : HOME : PRINT : PRINT "HEXADECIMAL-TO-DECIMAL
    CONVERSION (RANGE 0 TO FFFFFFFF)"
500  PRINT : PRINT "TYPE HEXADECIMAL NUMBER (OR 'M' FOR
    MENU):": PRINT
510  PRINT : INPUT "HEX: ";A$
520  IF A$ = "M" THEN 120
530  IF  LEN (A$) > 8 THEN 510
540 N = 0
550  FOR L = 1 TO  LEN (A$)
560 HEX$(L) =  MID$ (A$,L,1)
570  IF HEX$(L) < "0" OR HEX$(L) > "F" THEN 510
```

```
580  IF HEX$(L) <  = "9" THEN N = N * 16 +  VAL (HEX$(L)):
    GOTO 620
590  IF HEX$(L) < "A" THEN 510
600  IF HEX$(L) > "F" THEN 510
610 N = N * 16 +  ASC (HEX$(L)) -  ASC ("A") + 10
620  NEXT L
630  PRINT "DEC: ";N: PRINT
640  GOTO 510
650  REM  ************** BINARY-TO-DECIMAL CONVERSION ****
    **********
660  TEXT : HOME : REM  CLEAR SCREEN
670  PRINT : PRINT "BINARY-TO-DECIMAL CONVERSION PROGRAM"
680  PRINT : PRINT "INSTRUCTIONS: ENTER AN 8-BIT BINARY
    NUMBER (OR 'M' FOR MENU)"
690  PRINT : INPUT "BIN: ";A$
700  IF A$ = "M" THEN 120
710  IF  LEN (A$) <> 8 THEN 690
720  FOR L = 8 TO 1 STEP  - 1
730 B$(L) =  MID$ (A$,L,1)
740  IF B$(L) <  > "0" AND B$(L) <  > "1" THEN 680
750  NEXT L
760  FOR L = 1 TO 8
770 BIT(L) =  VAL (B$(L))
780  NEXT L
790 ANS = 0
800 M = 256
810  FOR L = 1 TO 8
820 M = M / 2:ANS = ANS + BIT(L) * M
830  NEXT L
840  PRINT "DECIMAL:";ANS
850  GOTO 680
860  REM  ********** DECIMAL-TO-BINARY CONVERSION
    **********************
870  TEXT : HOME : PRINT "DECIMAL-TO-BINARY CONVERSION
    PROGRAM (RANGE 0-255)"
880  PRINT : PRINT "ENTER A POSITIVE INTEGER (OR 'M' FOR
    MENU)"
890  PRINT : INPUT "DEC: ";A$
900  IF A$ = "M" THEN 120
910  IF  VAL (A$) < 0 OR  VAL (A$) > 255 THEN 890
920 NR =  VAL (A$)
930  FOR L = 8 TO 1 STEP  - 1
940 Q = NR / 2
950 R = Q -  INT (Q)
960  IF R = 0 THEN BT$(L) = "0": GOTO 980
970 BT$(L) = "1"
980 NR =  INT (Q)
990  NEXT L
1000  PRINT "BINARY: ";
1010  FOR L = 1 TO 8: PRINT BT$(L);: NEXT L: PRINT
1020  GOTO 880
1030  REM  ******** HEXADECIMAL-TO-BINARY CONVERSION
    ******************
1040  TEXT : HOME : PRINT "HEXADECIMAL-TO-BINARY CONVERSION
    PROGRAM (RANGE: 0 TO FF)"
1050  PRINT : PRINT "TYPE HEXADECIMAL NUMBER (OR 'M' FOR
    MENU)"
```

```
1060  PRINT : INPUT "HEX: ";A$
1070  IF A$ = "M" THEN 120
1080  IF  LEN (A$) > 2 OR  LEN (A$) < 1 THEN 1060
1090 HEX$(1) = "":HEX$(2) = ""
2000  FOR L = 1 TO  LEN (A$)
2010 HEX$(L) =  MID$ (A$,L,1)
2020  IF HEX$(L) < "0" OR HEX$(L) > "F" THEN 1060
2030  IF HEX$(L) > "9" AND HEXS$(L) < "A" THEN 1060
2040  NEXT L
2050  IF HEX$(2) = "" THEN HEX$(2) = HEX$(1):HEX$(1) = "0"
2060  FOR L = 1 TO 16: IF HEX$(1) = H$(L) THEN BIT$(1) =
    B$(L)
2070  NEXT L
2080  FOR L = 1 TO 16: IF HEX$(2) = H$(L) THEN BIT$(2) =
    B$(L)
2090  NEXT L
2095  PRINT "BIN: ";
2100  PRINT BIT$(1);BIT$(2): GOTO 1060
2110  REM  ************ BINARY TO HEXADECIMAL CONVERSION
    **************
2120  TEXT : HOME : PRINT "BINARY-TO-HEXADECIMAL
    CONVERSION"
2130  PRINT : PRINT "TYPE AN 8-BIT BINARY NUMBER (OR 'M'
    FOR MENU):"
2140  PRINT : INPUT "BIN: ";A$
2145  IF A$ = "M" THEN 120
2150  IF  LEN (A$) <  > 8 THEN 2140
2160  FOR L = 8 TO 1 STEP  - 1
2170 BIT$(L) =  MID$ (A$,L,1)
2180  IF BIT$(L) <  > "0" AND BIT$(L) <  > "1" THEN 2140
2190  NEXT L
2200 BIT$ = BIT$(1) + BIT$(2) + BIT$(3) + BIT$(4) + BIT$(5)
     + BIT$(6) + BIT$(7) + BIT$(8)
2210 T1$ =  LEFT$ (BIT$,4):T2$ =  RIGHT$ (BIT$,4)
2220  FOR L = 1 TO 16: IF T1$ = B$(L) THEN HEX$(1) = H$(L)
2240  NEXT L
2250  FOR L = 1 TO 16: IF T2$ = B$(L) THEN HEX$(2) = H$(L)
2260  NEXT L
2270  PRINT "HEX: ";HEX$(1);HEX$(2)
2280  GOTO 2140
```

3

In the Chips

As discussed in Chapter 1, the brain of your computer is its central processing unit, or CPU.

A central processing unit, as its name implies, is the central component in a computer system—the component in which all computing functions take place. In a microcomputer, such as the Apple IIc and IIe, all of the functions of a central processing unit are contained in a large-scale integrated circuit (LSI), sometimes referred to as a microprocessor unit (MPU).

Originally, the MPUs that were used in the Apple IIc and the Apple IIe differed slightly. The Apple IIe was originally built around a microprocessor called the 6502B. But the Apple IIc, and newer Apple IIe's, are built around a slightly more advanced chip called the 65C02. Both of these chips are members of the popular 6502 family of microprocessors, so all Apple IIc's and Apple IIe's are almost 100 percent compatible. The 65C02 chip can run programs written for the 6502B, but programs written for the 65C02 will not necessarily run on the older 6502B. (All of the programs in this book will run on both chips, however.)

In addition to the 6502B and the 65C02, two new 16-bit chips are now available for Apple II-series computers. One of these chips, the 65802, can process data at 16-bit speeds but can address only 64K of memory space—the same amount of memory space that can be handled by an unimproved (64K) Apple IIe. The 65802 is completely pin-compatible with a 6502, 6502B, or 65C02 chip, so it can be plugged directly into an Apple IIc or an Apple IIe. The 65802 can run standard 6502 software, as well as new software that is especially written to take advantage of its high-speed 16-bit data-handling capabilities.

The other new chip, the 65816, has a 16-bit data bus and a 24-bit address bus. It can handle up to 16 megabytes of address space. It manages its memory in a more complex fashion than other 6502-series chips do, so it is not pin-compatible with the 6502B and 65C02 chips built into the Apple IIe and the Apple IIc. It will, however, run software designed for the 6502 family of 8-bit chips, as well as specially designed 65802 and 65816 software.

The architecture of the 6502B and 65C02 chips will be discussed later in this chapter, and the two new "superchips" will be covered in greater detail later in this book. But before you can understand the operation of any microcomputer chip—even the plain, no-frills 6502 chip that was used in the first Apple II—you'll have to know something about the operation of computer chips in general. You should also have a general idea of what goes on inside a chip when it processes data and how a chip accesses data that's stored in the memory of an Apple II.

We begin by discussing how your Apple's CPU locates data that is stored in your computer's memory.

Your Apple's Memory

You know that RAM is your computer's short-term memory, while ROM is its long-term memory. In computer jargon, RAM is said to be *volatile*, while ROM is said to be *non-volatile*. That means that that RAM can be changed (or lost), while ROM cannot be.

RAM is the free memory space in your computer. When you turn your computer on, its RAM is as blank as a sheet of white paper. You can store anything in it that you wish, including text, data, programs, or even pictures that can be displayed on a screen (all of which, of course, must be represented by numbers).

When you turn your computer off, however, everything that you've stored in RAM disappears. That's what keeps computer-disk manufac-

turers in business. When you load a program that is on a disk into your Apple's memory, the program always gets loaded into the part of memory that is RAM. When the power to your computer goes off, the part of its memory that gets erased is also RAM.

We have compared the RAM banks in a microcomputer with a bank of mailboxes built along a wall inside a post office. Inside a computer, each of these "mailboxes" is called a memory register. And each memory register, like each box in a bank of post office boxes, has a unique memory address. In an Apple IIc or Apple IIe, each memory address can hold only one 8-bit number—that is, a number ranging from 0 to 255. That number can represent one of only four things:

- The stored number itself
- A code representing a text character
- A machine-language instruction
- A data element (such as a part of a graphics picture).

Since a computer contains many memory registers, and since the value stored in each memory address or location can have various meanings, a computer has to be told two things before it can run a program: where the program is situated in its memory and whether the value in each number in the program should be interpreted as a number, a text character, a machine-language instruction, or a data element.

Running a Machine-Language Program

Before a computer can run a machine-language program, it has to know the *starting address* of the program—the location at which the program has been stored in the computer's memory. Once the computer knows where a program starts, it can go directly to the first instruction in the program and carry that instruction out. The computer will then move on to the next address in its memory where, if the program has been properly written, it will find another instruction.

When you write an assembly-language program with any of the three assemblers that were used in the writing of this book, you will usually indicate the memory location where a program begins by typing a line that looks something like this:

```
ORG $8000
```

In an assembly-language program, a line like the one above is called an *origin directive (ORG)*. The Apple ProDOS assembler requires the

use of an ORG directive, but you can get by without one if you own a Merlin assembler or an ORCA/M, since both of those assemblers will assign a default address to a program if you don't provide one.

When your assembler encounters an ORG line at the beginning of a program, it will assemble the source code that follows into a machine-language program that begins at the address given in the directive; in this case, at hexadecimal memory address $8000 (or 32768 in decimal notation). Once the program has been assembled at the address given in the ORG directive—or at a default address—it can be saved on a disk and later run using a ProDOS command as simple as

```
BRUN PATHNAME
```

where the word PATHNAME represents the actual pathname of the program.

Using Data in an Assembly-Language Program

When an ORG line is used in an assembly-language program, it must point to the first memory address in the program that contains executable code (machine code that has been generated by valid assembly-language instructions). An ORG line should never point to a program segment that is made up of non-executable machine code, such as a table of data. If it does, the computer will try to interpret the data as executable code and will attempt to run it, and the results will be unpredictable.

This does not mean that you can't use data tables in an assembly-language program. You can, of course; but when a block of non-executable data is included in an assembly-language program, it is usually stored in a separate block of memory. Then the data can be accessed as needed from the main body of the program without its getting mixed up with executable code during the program's execution.

8-Bit and 16-Bit Numbers

The Apple IIc and the Apple IIe both belong to a class of computers called *8-bit computers*. That's because Apple II-series computers were all originally designed around the 6502 microprocessor, which is an 8-bit chip. Eight-bit chips can process binary numbers up to 8 places long, but no longer. As we saw in the previous chapter, a 6502/6502B/65C02 chip cannot perform a calculation on a binary number larger than 255 without breaking it down into smaller numbers. In fact, the 6502/6502B/65C02 chip can't even perform a calculation with a result that's greater than 255!

Obviously, this 8-bit limitation makes the manipulation of large numbers very inconvenient. In effect, a 6502/6502B/65C02 chip is like a calculator that can't handle a number larger than 255.

To work with numbers larger than 255, an 8-bit computer has to perform a rather complex series of operations. If a number is greater than 255, an 8-bit computer has to break it down into 8-bit chunks and perform each required calculation on each 8-bit number. Then the computer has to patch all of these 8-bit numbers together again.

If the result of a calculation is more than 8 bits long, things get even more complicated. That's because the memory registers in the Apple IIc and the Apple IIe are also incapable of handling numbers that are larger than 255. Each cell in your Apple's random-access memory (RAM), as well as in its read-only memory (ROM), is an 8-bit memory register. So, in order to store a number larger than 255 in your computer's memory, you'll always have to break it up into two or more 8-bit numbers and then store each of those numbers in a separate memory location. If you ever want to refer to the original number again, you have to patch the 8-bit pieces back together.

The Memory Architecture Of the 6502B/65C02 Microprocessor

We are now at a point that can be somewhat confusing. Although the Apple IIc and the Apple IIe cannot process numbers that are more than eight bits long, they can handle *addresses* that are up to 16 bits long. Figure 3-1, a simplified block diagram of your computer's CPU and components that are connected to it, may help you understand what that means.

When you examine Figure 3-1, you'll see that a 6502B/65C02 chip has

- Six internal registers (labeled PC, SP or S, P, X, Y, and A)
- An arithmetic/logical unit (labeled ALU)
- An 8-bit *data bus* and a 16-bit *address bus.*

Figure 3-1 also includes a block representing your computer's input/output (I/O) devices, as well as a block representing your Apple's memory (RAM and ROM). I/O devices will be covered in Chapter 11. In this section we'll focus on the portions of Figure 3-1 critical to memory: the data and address buses and the *program counter.*

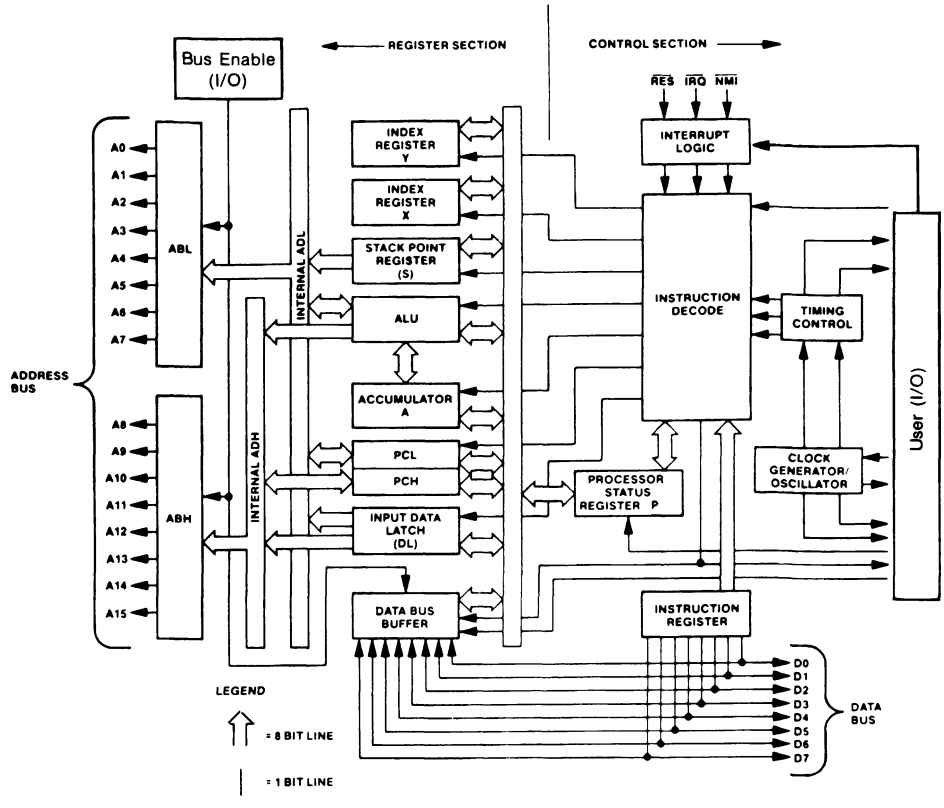

Figure 3-1. Block diagram of the 6502B/65C02 microprocessor

Data and Address Buses

At the bottom of Figure 3-1, there's a series of lines labeled "data bus." Along the left side of the drawing there's a row of arrows labeled "address bus." In computer terminology, a bus is a line over which information is transmitted inside a computer. The long bars at the top and bottom of Figure 3-1 represent lines that are used for the transmission of data and addresses inside your Apple IIe or Apple IIc.

Here's a very important point about the two buses illustrated in Figure 3-1: the data bus at the bottom of the diagram is an *8-bit* bus, and the address bus at the left is a *16-bit* bus.

An 8-bit bus is a line over which information can be transmitted eight bits at a time. A 16-bit bus is a line over which information can be

moved 16 bits at a time. Therefore, since a 6502B/65C02 chip has an 8-bit data bus, it cannot manipulate chunks of data that are more than eight bits long. But, since it has a 16-bit address bus, it can keep track of memory addresses that are up to 16 bits long. Let's see why.

The 6502B/65C02 Program Counter

In the center of Figure 3-1, there is a pair of boxes labeled *PCL* (for *"Program Counter-Low"*) and *PCH* (for *"Program Counter-High"*). These two boxes represent an internal register called a *program counter* (PC). In 6502-family chips, the program counter is a register that is used to keep track of the addresses of memory registers being used by a program. When your computer is running a machine-language program, the program counter in the CPU always holds the address of the next byte of data to be processed. Each time a byte is processed, the program counter is automatically incremented. It then holds the address of the next byte to be processed.

The PCL and PCH boxes in Figure 3-1 were given separate labels because the program counter in a 6502-family chip is actually made up of two 8-bit registers. These registers—the PCH and the PCL—are always used together as one 16-bit register which, as you have seen, is called the program counter.

Now you can understand how your computer keeps track of memory addresses up to 16 bits long: it simply uses two 8-bit registers together as a program counter. Thus it can handle memory addresses that extend from $0000 to $FFFF (or, in decimal notation, from 0 to 65535). As you may recall from Chapter 2, the binary equivalent of $FFFF, or 65535, is 1111 1111 1111 1111—the largest 16-bit number.

Now you know what people mean when they talk about 8-bit data buses and 16-bit address buses. And you also know why most 8-bit computers on the market can access 64K of memory.

Bank Switching

If your computer is an Apple IIe with 64K of memory, that's about all you need to know right now about the way your computer's CPU accesses data in its memory.

If you own an Apple IIc, or an Apple IIe with 128K of memory, there's one more point to cover: how the engineers who designed your computer managed to fit 128K of memory into 64K of RAM.

They did it with a design technique called *bank-switching.*

Bank-switching is not a very difficult concept. It involves switching different blocks of memory into the same address space. If you own an Apple IIc or an expanded Apple IIe, your computer has two 64K blocks of memory, one called *main memory* and the other called *auxiliary memory.* To switch back and forth between these two blocks of memory, all you do is change the contents of a certain series of memory locations (specifically, the locations that extend from memory address $C003 to memory address $C016). These locations are sometimes called *soft switches* because they are used just like hardware switches to turn things off and on.

If it were not for bank-switching, your Apple IIc or expanded Apple IIe would have a memory capacity of only 64K. With the help of bank-switching, an Apple IIc or an expanded Apple IIe can hold 128K of data, even though it isn't a true 128K computer. It's actually more like two 64K computers hooked together and wired to the same keyboard. Once you know how to use the right soft switches to bank-switch between such a computer's main and auxiliary memories, you can move back and forth between these two banks of memory quite freely. But you can never have access to both banks of memory at the same time. (Because switching occurs very quickly, it is not usually noticeable to the user. A programmer, however, must take it into account.)

Bank-switching techniques can also be used to switch certain segments of your computer's main memory from ROM to RAM and back to ROM. All of these bank-switching operations will be explained in more detail in Chapter 11, which is devoted to the memory organization of the Apple IIc and the Apple IIe. At this point, it is sufficient to understand that bank-switching techniques expand the memory capacity of the Apple IIc from 64K to 128K, and can boost the capacity of the Apple IIe even more.

Your Apple's CPU

Refer again to Figure 3-1. The 6502B and 65C02 microprocessors that came with your Apple IIe or IIc contain, as mentioned earlier, seven main parts: six addressable registers and an arithmetic logical unit or ALU. (The new 65802 and 65816 chips will be discussed in connection with addressing at the conclusion of this chapter and in Chapter 7.)

The Arithmetic Logical Unit

Just as your computer's command center is its central microprocessor, a CPU's command center is its ALU. Every time a 6502-based chip performs a calculation or a logical operation, the ALU is the component inside the chip where the work is actually done.

The ALU can actually perform only two kinds of calculations: addition and subtraction. Division and multiplication problems can also be solved by the ALU, but only in the form of sequences of addition and subtraction operations.

The ALU can compare values, too, but only by performing subtraction operations. To compare two values, the ALU simply subtracts one of them from the other. It can then determine whether one of the values is larger than the other or whether both values are the same.

The Accumulator

In Figure 3-1 the ALU is pictured just above the accumulator. When two numbers are to be added, subtracted, or compared, one of the numbers is first stored in the 65C02/6502B's accumulator. Next, the accumulator deposits that number in the ALU through one of the ALU's inputs. The other number is then placed in the ALU through the ALU's other input. Finally, the ALU carries out the requested calculation and the result appears at the output of the ALU. As soon as the answer appears, it is placed in the accumulator, where it replaces the value that was originally stored there.

Program 3-1 is a short assembly-language program that shows how this process works. This program is called ADDNRS.

Program 3-1
THE ADDNRS PROGRAM
An Example of an ALU Operation

```
LDA #2
ADC #3
STA $0300
```

The first instruction in the ADDNRS program, LDA, means "Load the accumulator." When the instruction LDA is encountered in a program, the accumulator is loaded with a certain value: specifically, the value of the operand that follows the instruction. In the ADDNRS program the effect of the instruction LDA is to load the accumulator with the literal number 2. (The "#" sign in front of the numeral 2 means that

the 2 in the instruction is to be interpreted as a literal number. If there were no "#", the 2 would be interpreted as the address of a memory location.)

The second instruction in the ADDNRS program, ADC, means "Add with carry." This command results in the addition of two numbers *plus* the carry bit (if the carry bit is set). In order to avoid an improper result when adding two 8-bit numbers, you should use the CLC (CLEAR CARRY command) prior to the ADC command. With the carry bit cleared, all the ADC instruction does is add 2 and 3.

As soon as the line "ADC #3" appears in the ADDNRS program, the 2 that has been loaded into the accumulator is deposited into one of the inputs of the 6502 chip's ALU. And the number 3, along with the instruction ADC, is placed in the ALU's other input. The ALU then carries out the ADC instruction that it has received: it adds 2 and 3 and places their sum back in the accumulator.

Now we're ready for the third and last instruction in the ADDNRS program. The instruction in line 3, STA, means "Store the contents of the accumulator" (in the memory address that follows). Since the accumulator now holds the value 5 (the sum of 2 and 3), the number 5 is about to be stored somewhere.

As you can see, the memory address that follows the instruction STA is $0300—the hexadecimal equivalent of the decimal number 768. So it appears that the number 5 is now going to be stored in memory location $0300.

Now take a look at the hexadecimal number $0300 in line 3. Since there is no "#" sign in front of the number $0300, the Apple ProDOS assembler will not interpret it as a literal number. Instead, $0300 will be interpreted as a memory address since it carries no other identifying labels. (If you did want your assembler to interpret $0300 as a literal number, you would have to write it as #$0300. When a "#" symbol and a dollar sign both appear before a number, that number is interpreted as a literal hexadecimal number.)

If the third line of the program were "STA #$0300", however, that would be a syntax error since STA ("Store the contents of the accumulator in ...") is an instruction that must be followed by a value that can be interpreted as a memory address.

The X Register

The *X register* (abbreviated "X") is an 8-bit register with a very special feature. It can be incremented and decremented automatically with a

pair of convenient one-byte instructions: INX, which stands for "Increment the X register," and DEX, which means "Decrement the X register." Since the X register can be incremented and decremented so easily, it is often used as an index register, or counter, during loops and read/data-type instructions in programs. When its special incrementing and decrementing features are not being used, the X register can be used for the temporary storage of data.

The Y Register

The *Y register* (abbreviated "Y") is also an 8-bit register and can also be incremented and decremented with a pair of special one-byte instructions. The mnemonics that are used to increment and decrement the Y register are INY and DEY. When its special incrementing and decrementing features are not being used, the Y register can also be used for the temporary storage of data.

The Program Counter

The program counter (PC) was described during our discussion of your computer's memory. You will recall that the PC is a pair of 8-bit registers designed to be used together as one 16-bit register. When the 65C02/6502B is running a machine-language program, the program counter always contains the 16-bit address of the next memory register to be accessed by the program. When that instruction has been carried out, the address of the next instruction is loaded into the program counter.

We have already mentioned that the two 8-bit registers that make up the program counter are often referred to as the Program Counter-Low (PCL) register and the Program Counter-High (PCH) register.

The Stack Pointer

The *stack pointer* (abbreviated as either "S" or "SP") is an 8-bit register that always "points to," or contains, the address of the top element in a block of RAM called the *hardware stack.* The hardware stack (also referred to as the *stack*) is a special block of memory in which data is often stored temporarily during the execution of a program. When subroutines are used in assembly-language programs, the 65C02/6502B chip uses the stack as a temporary storage location for return addresses. Beginning with Chapter 7, you will learn to use the stack for other purposes in assembly-language programs.

The Processor Status Register

The *processor status register* (also called the *status register* but abbreviated as "P" so it won't be confused with the stack pointer register) is an 8-bit register that keeps track of the results of operations that have been performed by the 65C02/6502B. Let's take a look at this very important register.

The P register is built differently from the other registers in the 65C02/6502B, and it is used differently, too. Unlike the others, it isn't designed for storing or processing ordinary 8-bit numbers. Instead, the P register's bits are used as flags designed to keep track of several kinds of important information.

Four of the status register's eight bits are called *status flags.* These four flags, with their abbreviations, are

- The *carry flag* (*C*)
- The *overflow flag* (*V*)
- The *negative flag* (*N*)
- The *zero flag* (*Z*).

These flags are used to keep track of the results of operations being carried out by the other registers inside the 65C02/6502B processor.

Since the P register is an 8-bit register, it has four additional flags. Three of these flags are called *condition flags* and are used to determine whether certain conditions exist in a program:

- The *interrupt disable flag* (*I*)
- The *break flag* (*B*)
- The *decimal mode flag* (*D*).

The processor status register's eighth bit is not used.

Layout of the Processor Status Register

The processor status register can be visualized as a rectangular box containing eight square compartments. Each compartment in the box is actually one of the P register's eight bits. In the processor status register, each of these bits is used as a flag.

If a given bit has the binary value 1, then it is said to be *set.* If it has the binary value 0, then it is said to be *clear.*

The bits in the 65C02/6502B status register—like the bits in all 8-bit registers—are customarily numbered from 0 to 7. By convention, the far-right bit in an 8-bit register is generally referred to as bit 0,

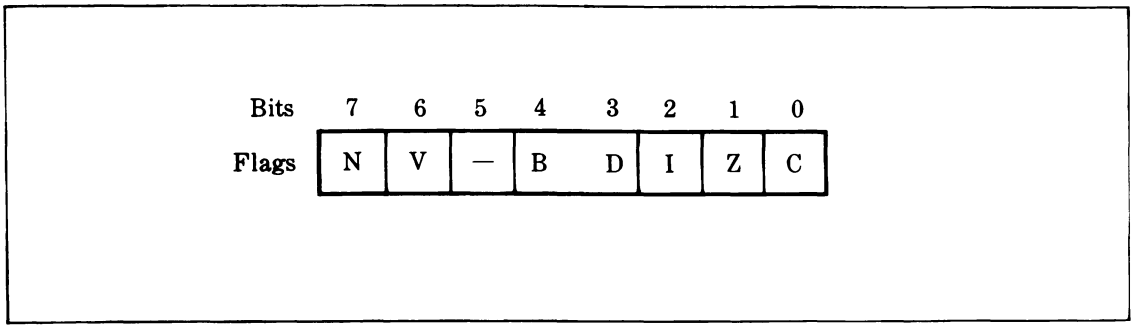

Figure 3-2. The 65C02/6502B processor status register

and the far-left bit is generally referred to as bit 7. The positions of these bits in the 6502B/65C02 status register are illustrated in Figure 3-2 above.

Here's a complete listing and explanation of the flags in the 65C02/6502B processor status register.

Bit 0: The Carry Flag (C) The carry flag (C) is one of the busiest bits in the 65C02/6502B processor status register. It tells whether a number must be carried from one byte to another during an arithmetic operation.

In an 8-bit chip like the 65C02/6502B, the carry flag has special importance. Without a carry flag, the 65C02/6502B would not be able to perform operations on numbers larger than 255. When the 65C02/6502B chip has to perform an addition operation on a number greater than 255, or if the result of a calculation might turn out to be greater than 255, that number must be broken down into 8-bit segments for processing and then patched back together. The P register's carry flag plays an important role in this mathematical cutting and pasting.

If two 16-bit numbers are to be added, each number must first be broken down into two 8-bit bytes. The low-order bytes of the two numbers must then be added together. If this operation results in a sum greater than 255, the carry flag will automatically be set. Then, if the high-order bytes of the two numbers are added, the carry bit will be added automatically to their sum.

The carry flag is set and cleared not only in addition operations, but also in many other kinds of operations performed by the 65C02/6502B chip. Detailed instructions covering the use of the carry flag will be presented in Chapters 9 and 10.

The assembly-language instruction to clear the carry flag is CLC, which stands for "Clear carry." The mnemonic that sets the carry bit is SEC, which stands for "Set carry."

Bit 1: The Zero Flag (Z) The zero flag is set when the result of an arithmetic operation, a logical operation, or a comparison operation is 0. When a memory location or an index register has been decremented to 0, that will also set the zero flag.

When you write routines that make use of the zero flag, remember this 6502 convention which may seem odd to you: when the result of an operation is zero, the zero flag is *set* (to 1), and when the result of an operation is *not* zero, the zero flag is *cleared* (to 0). Don't let this convention trip you up.

There are no assembly-language instructions to clear or set the zero flag because it's strictly a read-only bit.

Bit 2: The Interrupt Disable Flag (I) In assembly-language terminology, an *interrupt* instruction brings all of a computer's operations to a halt so that some very important or time-critical operation can take place. Some interrupts are called *maskable interrupts* because special instructions can prevent them from taking place. Other interrupts are called *nonmaskable* because there is no way that a programmer can prevent them from occurring. Nonmaskable interrupts are not used in Apple IIc/IIe programming.

You can disable a maskable interrupt by clearing the interrupt disable bit of the processor status register. When this flag is set, maskable interrupts cannot take place. When it is clear, they can.

The mnemonic that clears the interrupt flag is CLI. The mnemonic that sets it is SEI.

Bit 3: The Decimal Mode Flag (D) Normally, the 65C02/6502B processor operates in what is called *binary mode*, using standard 8-bit binary numbers. But your computer's CPU can also operate in what is known as a *binary-coded decimal*, or *BCD* mode. To make your Apple operate in BCD mode, you have to set the decimal mode flag of the 65C02/6502B status register.

When the 65C02/6502B chip is put in BCD mode, it uses only the 10 standard decimal digits—the numbers 0 through 9. The hexadecimal digits A through F are not used in BCD operations. Furthermore, every digit in a BCD number is treated as an individual byte. For example, it would require three bytes (one byte per digit) to express the decimal

number 255 as a BCD number. In your computer's memory, the BCD number 255 would be stored this way:

BCD number:	2	5	5
Binary equivalent:	00000010	00000101	00000101

That is quite different, of course, from the way the decimal number 255 would be expressed in conventional binary (non-BCD) notation. In binary arithmetic, the kind you'll probably use most often in your assembly-language programs, the decimal number 255 would be expressed as a hexadecimal number.

Decimal number: 255

Hexadecimal equivalent: FF

Binary equivalent: 11111111

As you can see, at the rate of one byte per digit, it takes much more memory to store BCD numbers than it takes to store conventional binary numbers. (It is possible to "pack" BCD digits into half that amount of space with special procedures and additional processing time, as you'll see in a later chapter.) Another disadvantage of BCD arithmetic is that it's slower than binary arithmetic. But BCD math is based on the number 10 rather than the number 8, so it is much more accurate than hexadecimal math in terms of real-world, base-10 arithmetic problems.

Another advantage of BCD numbers is that they're easier to convert into decimal numbers than standard binary numbers are. BCD numbers are therefore sometimes used in programs calling for instant number display on a video monitor.

BCD numbers will be discussed further in a later chapter. For now, it's sufficient to remember that when the status register's decimal mode flag is set, the 65C02/6502B chip will perform all of its arithmetic using BCD numbers. If you aren't using BCD arithmetic in your assembly-language programs, you must make sure that the decimal flag is clear before your computer starts performing arithmetical operations.

The assembly-language instruction that clears the decimal flag is CLD. The instruction that sets it is SED.

Bit 4: The Break Flag (B) The break flag is often used by programmers during the debugging of assembly-language programs. It is set by the assembly-language instruction BRK, an instruction ordinarily used only during the debugging of programs.

When a programmer is writing an assembly-language program, BRK instructions are often inserted at critical points in the program. The break flag is set following a software BREAK instruction. When encountered during the processing of the program, certain error-flagging operations return control of the computer to the programmer. Thus, the BRK instruction halts program execution and causes the contents of the 65C02/6502B chip's A, X, Y, P, and S registers to be displayed on the screen. The contents of these registers can then be examined to see what, if anything, went wrong during the processing of the program.

Once a program has been debugged, any BRK instructions that were placed in it for debugging purposes are usually removed. The program will then run normally.

Bit 5: Unused The engineers who designed the 6502 chip left this bit unused. As you will see later in this chapter, though, bit 5 of the P register is used in the 6502's two new 16-bit cousins, the 65802 and the 65816.

Bit 6: The Overflow Flag (V) The overflow flag is used to detect an overflow in a binary number from bit 6 to bit 7. It is used primarily in operations dealing with signed (plus or minus) numbers. In an ordinary, unsigned binary number, the highest or most significant bit is bit 7. In a signed number, however, bit 7 is used to designate the number's sign. If bit 7 of a signed number is clear, the number is positive. If bit 7 is set, the number is negative.

Since bit 7 of a signed number is not used as part of the number itself, the most significant bit of a signed number is bit 6. It is rather difficult, then, to perform carrying and borrowing operations in arithmetic problems that deal with signed numbers, so the overflow flag is often used to keep track of carrying and borrowing operations in signed binary arithmetic. In an addition problem, the overflow flag is set when bit 7 of both addends is the same value and bit 7 of the sum is the opposite value. In a subtraction operation, the overflow flag is set when bit 7 of the subtrahend and minuend are opposite and bit 7 of the result has the same value as bit 7 of the subtrahend. In other words, the overflow flag is set when there is an overflow from bit 6 to bit 7 but there is no external carry or, conversely, when there is no overflow from bit 6 to bit 7 but there is an external carry. A more technical way to say this is that the overflow flag is set by performing an exclusive-OR operation on the carry-in and carry-out of bit 7. This procedure may seem quite complicated, but without the help of the overflow flag it would be impossible to write programs to handle operations involving signed numbers.

The assembly-language mnemonic that clears the overflow flag is CLV. The V flag can be cleared, but there is no specific instruction to set it.

Bit 7: The Negative Flag The negative flag (N) of the processor status register is set when the result of an operation is negative and cleared when the result of an operation is 0 or positive. The negative flag is often used in comparison operations and in loop countdowns designed to extend all the way down to 0.

The N bit is a read-only bit, so no instruction to set it is provided.

65802/65816 Architecture

As illustrated in Figure 3-1, the architecture of the 6502, 65C02, and 6502B chips can be represented by the same block diagram. There are significant differences, however, between the architecture of the 65C02/6502B chip and that of the two new 16-bit chips in the 6502 family, the 65802 and the 65816.

The E (Emulation) Flag

In both the 65802 and the 65816, there is an extra status flag called the *emulation* (or *E*) flag.This flag is not part of the 65802/65816 processor status register but is an independent toggle switch built into the CPU itself. However, the 65802/65816 instruction XCE can be used to change the status of the E flag. By using an XCE instruction during an assembly-language program, you can temporarily exchange the positions of the free-floating E flag and the C (carry) flag of the processor status register. Since the carry flag ordinarily resides in bit 0 of the P register, the effect of the instruction XCE is to store the content of the E register in bit 0 of the P register, while placing the carry flag temporarily in the E register. Once this exchange has taken place, an assembly-language instruction such as SEC (set carry) or CLC (clear carry) can be used to change the status of the E flag. Another XCE instruction can then be issued to restore the E flag and C flag back to their normal positions. The state of the E flag will then be changed, and the C flag can again be used normally.

When the E flag of the P register is cleared to 0, its default setting, the 65802/65816 is a 16-bit chip. When the E flag is set to 1, however, the 65802/65816 chip is placed in what is called a 6502 emulation mode.

When the 65802/65816 chip is in its default 16-bit mode, it can be programmed in a language that is a superset of standard 6502 assembly

language. When the chip is in its 6502 emulation mode, it can be used exactly like a standard 6502 chip (or like a 65C02 or a 6502B). In its emulation mode, the 65802/65816 can recognize and process the same machine-language instructions as its 8-bit cousins can, so it can run software written for the 6502, the 6502B, and the 65C02.

Additional P Register Flags

There are two additional differences between the processor status register of a 6502B/65C02 chip and the P register of the 65802 and the 65816. Bit 4 of the P register—the break (B) flag on a 6502B/65C02 chip—is known as the *X flag*, or *index register select flag*, on the 65802/65816. Bit 5 of the P register, which is not used by the 6502B/65C02, is called the *M flag*, or *memory select flag*, on the 65802/65816. Figure 3-3 shows how the bits of the 65802/65816 P register are coded when the chip is in its 16-bit mode.

The M Flag When both the E flag and the M flag of the 65802/65816 are clear, the chip's accumulator is 16 bits long and is called the *C register*. When the accumulator is in its 16-bit C-register mode, its lower eight bits are known as the *A register* and its higher eight bits are known as the *B register*. But the names of assembly-language instructions to the accumulator are not affected by these changes; the mnemonic LDA, for example, remains LDA, and STA remains STA.

There are some changes, of course, in the way that LDA, STA, and other accumulator instructions work when the 65802/65816 is in its 16-bit mode. When the chip is in this mode, instructions such as LDA and STA can be used with two-byte operands. When a two-byte operand is used in this fashion, its low byte affects the accumulator's A register, and its high byte affects the B register. For example, the statement

```
LDA #$8000
```

would load the literal value 00 into the A register and the literal value 80 into the B register.

However, the statement

```
LDA $8000
```

would load the A register with the value of memory register $1000 and the B register with the value of memory register $1001.

The X Flag When both the E flag and the X flag of the 65802/65816 are clear, the chip's X and Y registers are 16 bits long. When the X and

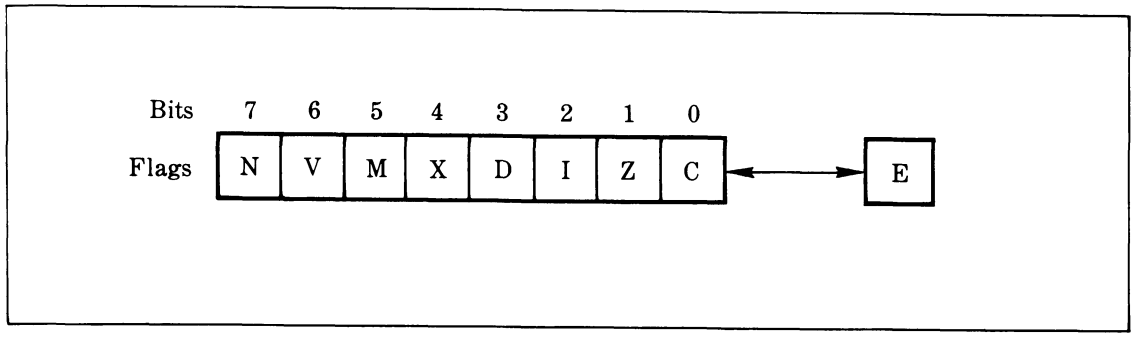

Figure 3-3. The 65802/65816 processor status register (when chip is in 16-bit mode)

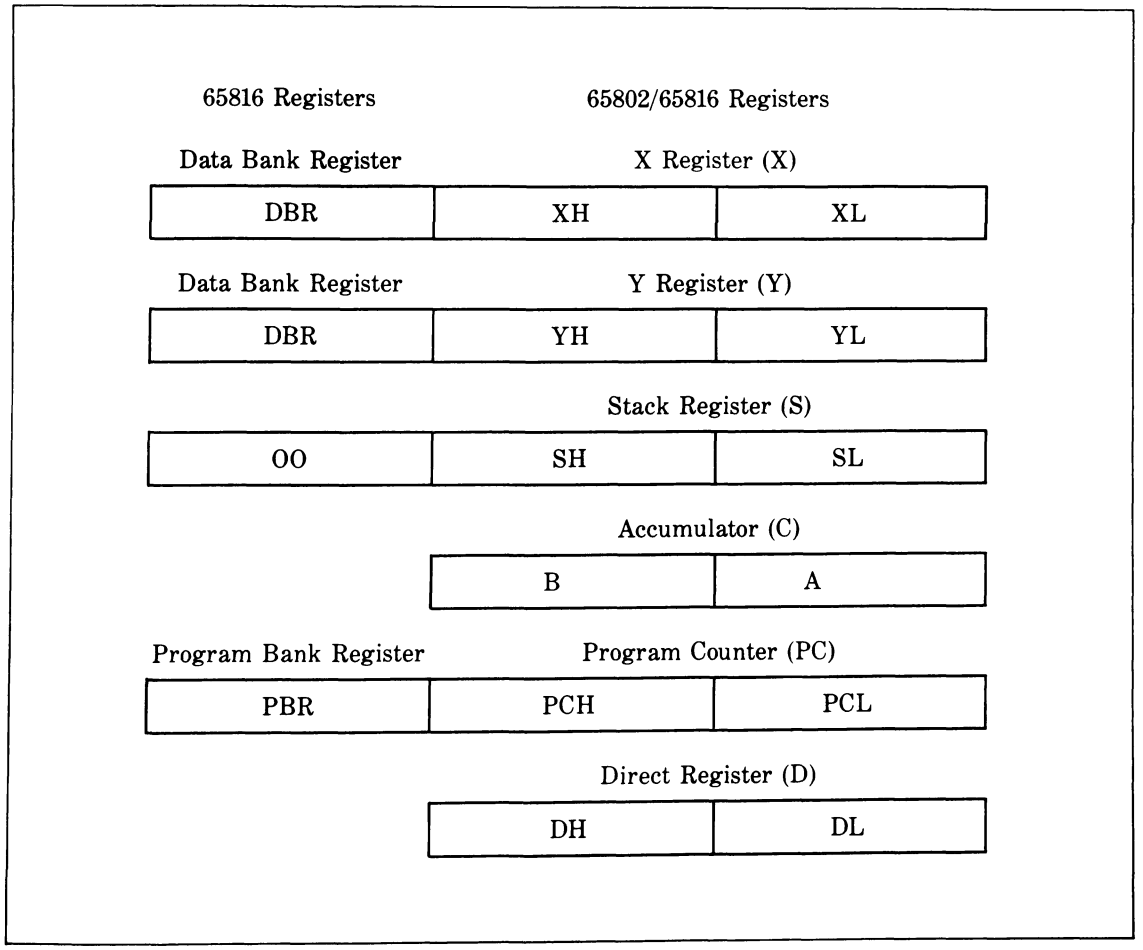

Figure 3-4. Internal registers of the 65802 and 65816 microprocessors

Y registers are in their 16-bit mode, their lower eight bits are known as XL and YL, and their higher eight bits are known as XH and YH.

The S Register

When the E register of the 65802/65816 is clear, the chip's stack register (S register) is 16 bits long. This expansion makes it possible to place the 65802/65816 chip's hardware stack anywhere in memory.

The D Register

The 65802 chip has one more internal register than its 8-bit relatives. This extra register is called the *direct page register*, or *D register*. When the 65816's E register is clear, the D register can be used to enhance the chip's ability to handle a very useful and speedy type of addressing called zero-page addressing in 6502 assembly-language programming. The operation of the D register will be covered in detail in Chapter 7.

The 65816 Chip's 24-Bit Registers

In addition to the D register, the 65816 chip has two other special registers: a *program bank register*, or *PBR*, and a *data bank register*, or *DBR*.

The program bank register is an 8-bit register that can extend the length of the 65816's program counter to 24 bits. When the PBR is used, therefore, the 65816 can address up to 16 megabytes of memory.

With the help of the data bank registers, the 65816's X and Y registers can also be expanded into 24-bit registers. And that capability provides the 65816 with a simple method for addressing up to 24 megabytes of data. The addressing capabilities of the 65816 will be explained in greater detail in Chapter 7. Meanwhile, Figure 3-4 is a block diagram that shows the internal registers of the 65802 and 65816.

4

Writing and Assembling An Assembly-Language Program

In this chapter, you finally get a chance to write a real assembly-language program. You can't create and assemble an assembly-language program, of course, unless you have a software package called a machine-language assembler. You may recall from the Introduction to this book that three different assemblers were used to write the programs in *Apple Roots*. To get the maximum possible value out of this book, you should probably own at least one of them.

- The Apple *ProDOS Assembler Tools* package, manufactured by Apple.

- The *Merlin Pro* assembler, manufactured by Roger Wagner Publishing, Inc., of Santee, California.
- The *ORCA/M* assembler, manufactured by The Byte Works, Inc., of Albuquerque, New Mexico.

These three assemblers were compared fairly comprehensively in the Introduction. Now you will start using the assembler you have chosen.

This chapter is divided into three parts. I will explain how to type and assemble a program first using the Apple ProDOS assembler, then using the Merlin Pro assembler, and finally using the ORCA/M assembler.

No matter which assembler you're using, you should read the discussion of the Apple ProDOS assembler carefully, since it contains the only line-by-line explanation of how the program works.

As you may have guessed, the term "assembler" can mean two different things, depending upon the way in which it is used. In its more accurate sense, an assembler is just one part of an assembler/editor software package: the part that does the actual work of converting assembly language into machine language. But sometimes the word "assembler" is used to refer to a complete software development kit such as the Apple ProDOS assembler, the ORCA/M assembler, or the Merlin Pro. This book uses the word in both ways. The context in which the word is used should make its meaning clear.

The Apple ProDOS Assembler

The Apple *ProDOS Assembler Tools* kit is made up of four different programs:

- An editor, which can be used to create and edit assembly-language programs and to store them on disks (and also to create and edit ProDOS EXEC files and BASIC programs).
- An assembler, used to assemble source-code files into executable machine-language programs.
- An assembly-language debugging utility called the *Bugbyter*, which allows you to track down and correct errors in your assembly-language programs.
- A relocating loader used to load and execute assembly-language programs during the execution of BASIC programs.

All of the programs in the Apple ProDOS assembler package are stored on a single disk. This disk is not copy-protected, so you can (and should) make at least one backup copy of it before you start using your ProDOS assembler package to write any programs. Once you have copied the master assembler disk, you can use your copy as a master. Then you can put the original master disk away for safekeeping.

Once you've made a copy of your Assembler Tools disk, boot your duplicate disk just as you would any other ProDOS disk. You should now see a display that looks like this:

```
PRODOS BASIC 1.0
COPYRIGHT APPLE, 1983
]
```

After the "]" prompt, there will be a flashing cursor. There you can now type the line

```
]-EDASM.SYSTEM
```

After a little disk-spinning, you will see this video display:

```
PRODOS EDITOR-ASSEMBLER //

ENTER THE DATE AND
PRESS RETURN

DD-MMM-YY
```

There will be a flashing cursor over the first D in the last line of the display. Type the date over the letters DD (for date), MMM (for month), and YY (for year). You will then see

```
1
PRODOS EDITOR-ASSEMBLER //
BY JOHN ARKLEY
(C) COPYRIGHT 1982

APPLE COMPUTER INC.
@


          DD-MMM-YY

          :
```

(DD-MMM-YY will vary according to what you type in.)

When you see that display, you'll know that your assembler is in its command level mode, the mode in which you'll be writing and editing assembly-language programs.

The program that you will write in this chapter is the ADDNRS program presented in Chapter 3. As you may recall, ADDNRS is a very short and simple program. It simply adds the numbers 2 and 3 and stores their sum in a certain memory register (specifically, memory register $0300).

The version of the ADDNRS program that will be written using the Apple ProDOS assembler is called ADDNRS.SRC.

Entering the ADDNRS.SRC Program

Near the bottom of your monitor screen, just below the date, you will see a colon followed by a flashing cursor. The ":" prompt just behind the cursor is the prompt you'll always see when your assembler is in command (or editing) mode—that is, when you're using the editor module of your Apple *ProDOS Assembler Tools* package.

When you see the colon prompt, type A for "Append." You'll then see the number 1 appear on your screen. That 1 is a line number—the number of the first line you'll type when you start writing a source-code program.

Line Numbers in Apple ProDOS Assembler Programs The number 1 appeared automatically on your screen because the Apple ProDOS source-code editor automatically generates *line numbers* in source-code programs, beginning with the number 1 and progressing in increments of 1. It is important to note, however, that the line numbers generated by the Apple ProDOS source-code editor are very different from the line numbers commonly used in BASIC programs. Unlike BASIC, 6502 assembly language does not require the use of line numbers. In fact, some 6502-family assemblers—such as the ORCA/M assembler—do not use line numbers at all.

Line numbers are optional in assembly-language programs because a routine is never referred to by its number in 6502 assembly language; segments of source-code are usually accessed with the help of descriptive labels. A descriptive label can be assigned to the first line of any routine in an assembly-language program and can then be used to access that routine whenever desired.

Since line numbers are not essential to the operation of assembly-language programs, the Apple ProDOS source-code editor generates what are sometimes referred to as *relative line numbers*. A relative line number is not an integral part of an assembly-language program; it is provided only as a convenience so that programmers and program users can find their way around more easily in assembly-language programs.

There are advantages and disadvantages to using relative numbers. You never have to worry about numbering or renumbering the lines in a program. Relative line numbers are generated automatically, starting at the number 1 and progressing consecutively in increments of one. When the Apple ProDOS source-code editor numbers the lines of a program, it never skips a number. It will automatically renumber the lines in a program, too; when a line is inserted or deleted, every line below it is immediately and automatically renumbered.

It's important to treat relative line numbers with great care when you're writing and editing an assembly-language program because they can change without notice. When you're making multiple deletions of lines, for example, the lines that you delete first will change the numbers of later lines, so it's smart to delete the lines with higher numbers first. Otherwise, you might later delete the wrong lines.

Take a close look at the number 1 on your screen, and you'll see that it is followed by one space and a flashing cursor. Now type one more space, and then, without touching your RETURN key, type

```
ORG $1000
```

Line 1 of your program will read

```
1  ORG $1000
```

Now press the RETURN key. Your ProDOS editor will move down to the next line on your screen and print a 2, thus signaling that it's ready for you to type line 2 of the ADDNRS.SRC program.

After the relative line number 2—but this time without an extra space—type a semicolon. Line 2 of your program will be displayed as

```
2 ;
```

Press the RETURN key again. Your editor will advance to line 3. Then type these two lines:

```
3 ; ADDNRS.SRC
4 ;
```

Press the carriage return again after line 4 and you will see

```
1  ORG $1000
2 ;
3 ; ADDNRS.SRC
4 ;
5
```

In a moment, you learn what those lines mean. First, though, let's examine a few of the important editing features of the Apple ProDOS assembler/editor system.

Single-Character Editing Functions If you make a mistake while you're typing a line of source code using the ProDOS source-code editor, there are several ways you can correct it. If you are in "append" mode, you can change any letter in a line by typing another letter directly over it. You can delete a character that lies directly under your cursor by typing **CONTROL-D**. You can delete the character to the left of your cursor by using the DELETE key. You can insert a character into a line at the location of your cursor by typing **CONTROL-I**. If you change your mind about correcting a line and want to restore the original version, you type **CONTROL-R**. (One word of caution, however: When the ProDOS assembler is in its "edit" mode, some of its responses to editing commands are different from the responses to the same commands in "add" mode. In "edit" mode, for example, the DELETE key erases the character under the cursor, not the character to its left.)

Line-By-Line Editing Functions The Apple ProDOS source-code editor is a line-oriented editor; you can move back and forth within a line using your left and right ARROW keys, but you can't move from line to line using your up and down ARROW keys. When you want to exit a line, you have to use the carriage return. And each time you type a carriage return at the end of a line, the line that precedes the carriage return line is automatically included in the program that you're typing.

When you finish typing a program—or when you want to stop typing lines to go back and do some editing—you can make your assembler stop generating new line numbers by simply pressing the RETURN key twice. Then your editor will stop creating new line numbers and the ":" prompt will be displayed on the screen.

```
14-JUN-85
:
```

When you see that display, you can type **A** (for "append"), just as you did when you began this programming session. Your assembler will then resume numbering lines exactly where it left off, and you can continue writing your program.

Two other commands that can be used after the ":" prompt are the L (for LIST) and P (for PRINT) commands. If you type an L command, your entire program will be listed, complete with line numbers, on your

computer screen. The P command will also display your program on your screen, but without line numbers.

If you want to look at a specific line in a program, you can type the line number that you want after either the L command or the P command. If you want to display a block of lines, you can specify the first and last line number that you want to see by using this kind of format:

```
L begin#-end#
```

or

```
P begin#-end#
```

The DELETE Command Another command that can follow the ":" prompt is D (for DELETE). To use the DELETE command, all you have to do is type the letter **D** followed by the number of the line (or lines) you want to delete. Suppose, for example, that you want to delete lines 2 and 3 in the above listing. You can do that by entering **D2-3** after the ":" prompt.

Still another command that can be used after the ":" prompt is I (for INSERT). To use the INSERT command, type the letter **I** after a colon prompt, followed by the number of the line where you want your new line inserted. Let's insert another semicolon at line 2 in the program you are creating. Press your RETURN key to get a colon prompt, and then type

```
I2
```

You'll see your assembler-editor respond with the number 2.

Now type a semicolon followed by two carriage returns. The Apple ProDOS source-code editor will display its ":" prompt again, and you can then type **L** for list. Your assembler will then list your program; you'll see that another line containing a semicolon has been inserted into your program at line 2. Notice that your assembler also has automatically renumbered each line after the line that you've inserted, a process that was explained earlier.

Now that you've seen your assembler's relative line-numbering system in action, you can delete the extra semicolon that you added to your program. Press the carriage return to get a colon prompt and type **D2**. Then you can type **L** for LIST, to generate a listing showing you your program again.

```
1  ORG $1000
2 ;
3 ; ADDNRS.SRC
4 ;
```

Besides the A, I, and D commands that you've just used, there's also an R command that you can use to restore a line. In addition, the Apple ProDOS source-code editor also has commands that you can use to copy lines, find and replace strings, and perform many other useful functions. Full details on how to use these editing functions and many others are provided in the instruction manual that came with your Apple ProDOS assembler.

Other Features of the ProDOS Editor The Apple ProDOS editor has many other features that won't be covered in this chapter. For example, there's a SWAP command that you can use to tuck a source-code program away in a hidden block of memory in order to write or edit another program. You can then SWAP your two programs back and forth at will, until you're either ready to SAVE both of them or to delete one or both of them using another assembler command, KILL2.

Completing the ADDNRS.SRC Program

Now it's time to finish typing the ADDNRS.SRC program. Program 4-1 is a complete source-code listing of this program, and now is a good time to enter the rest of the listing into your computer. Before you start typing, though, here's a word of caution. The Apple ProDOS assembler/editor, like most assembler/editors, is very finicky about spacing, so be sure to type the ADDNRS.SRC program exactly as it appears in Program 4-1. In lines 2 through 5, there should be no extra spaces between the relative line numbers generated by your assembler and the instructions that follow them. (The ProDOS editor will automatically insert one space after a line number.) There should be one extra space after line 1, however, and there should also be one extra space after line numbers 6 through 10. If you follow all of these rules of spacing (which will be explained more fully later in this chapter), your own source-code listing of the ADDNRS.SRC program should look just like Program 4-1.

Program 4-1
THE ADDNRS.SRC PROGRAM
(Apple ProDOS Assembler Version)

```
1  ORG $1000
2 ;
3 ; ADDNRS.SRC
4 ;
5 ADDNRS CLD
```

```
 6  CLC
 7  LDA #2
 8  ADC #3
 9  STA $0300
10  RTS
```

Listing Your Program

When you've finished typing line 10 of your ADDNRS.SRC program, just press your RETURN key twice. Then type either **L** or **LIST**, and the complete program will be listed on your computer screen.

You may have noticed that the ADDNRS.SRC program that you've just typed and listed is the same program that was first introduced in Chapter 3. Later on, when you run the program, you'll learn exactly how it functions. You know that it adds the numbers 2 and 3 and stores the result in memory address $0300—or in decimal notation, memory register 768. You can see by looking at the program that the numbers are added in lines 7 and 8, and their sum is stored in memory register $0300 in line 9.

Spacing

Program 4-2 is an "explosion diagram" of the ADDNRS.SRC program. It does not follow the rules of spacing that were explained a few paragraphs back, but it may give you a clearer picture of how the information contained in a source-code listing can be split up into *fields*, or columns.

Program 4-2
THE ADDNRS.SRC PROGRAM, COLUMN BY COLUMN

```
LINE    LABELS AND    OP      OPERAND
NO.     REMARKS       CODE                    COMMENTS
----    ------------  ----    ----------      -------

 1                    ORG     $1000
 2      ;
 3      ; ADDNRS.SRC
 4      ;
 5      ADDNRS        CLD
 6                    CLC
 7                    LDA     #2
 8                    ADC     #3
 9                    STA     $0300
10                    RTS
```

Fields in Source-Code Listings

As you can see by looking at Program 4-2, assembly-language programs that are created with the Apple ProDOS assembler-editor can be

divided into several columns, or fields. The fields illustrated in Program 4-2 are the *label* field, the *op-code* field, the *operand* field, and the *comments* field. In addition, one column is used for line numbers. However, since line numbers are optional in assembly-language programs, the column used for line numbers in a source-code listing is not considered a separate field.

The Label Field Labels, when they are used, always occupy the first field in an assembly-language source-code listing. Most types of statements in an assembly-language program can be identified with labels, and labels are actually required with a few types of assembly-language commands.

Even though they are optional, labels are very important in 65C02/6502B assembly language, since they are used instead of line numbers to address routines and subroutines in assembly language. In the ADDNRS.SRC program, the abbreviation ADDNRS in line 5 is a label, so it appears in the first field of Program 4-2.

In an assembly-language program, remarks can also start in the label field. When a remark begins in the label field, however, it must be preceded by some kind of identifying mark, usually an asterisk or a semicolon (depending on the assembler). On the Apple ProDOS assembler, a remark can be preceded by either an asterisk or a semicolon.

When a remark starts in the label field, it may extend across other fields. A remark, like every other line in an assembly-language program, always ends with a carriage return.

When a label is assigned to an assembly-language routine or an assembly-language program, the program can then be saved on a disk and used later as a subroutine or a secondary routine in a larger program. Since the ADDNRS.SRC program has been assigned a label—the label ADDNRS, which appears in line 5—the program could be included in a larger program and then accessed using either the instruction JSR ADDNRS (the assembly-language equivalent to the GOSUB instruction in BASIC) or the instruction JMP ADDNRS (which works like BASIC's GOTO instruction). Other jumping and branching instructions can be used with labeled routines and subroutines; they will all be covered in later chapters.

If ADDNRS.SRC were used as a subroutine in a longer program, the RTS (return from subroutine) instruction in line 10 would end the subroutine and return control to the main program. If ADDNRS.SRC were used to label a complete program, the RTS instruction would end the program. (The JSR and JMP instructions are discussed further in later chapters.)

A label can be as short as one character or as long as the length of a statement permits. Most programmers use labels three to six characters long. However, some labels (such as A, X, Y) are restricted, and long labels are generally undesirable because they slow down assembly.

The Op-Code Field An *operation-code* (or *op-code*) mnemonic is another name for an assembly-language instruction. There are 57 op-code mnemonics in the 6502B microprocessor used in the Apple IIe, and there are ten additional mnemonics in the expanded instruction set that can be used with the 65C02 chip used in the Apple IIc.

Op-code mnemonics, such as CLC, CLD, LDA, ADC, STA, and RTS, are typed in the op-code field of assembly-language source-code listings. When you write a program using the Apple ProDOS Assembler Tools package, each op-code mnemonic you use must start at least two spaces after a line number or one space after a label. An op-code mnemonic placed in the wrong field will not be flagged as an error when you type your program, but will be flagged as an error when your program is assembled.

The op-code field in a source-code listing is also used for *directives* and *pseudo-ops*—words and symbols that are entered into a program like mnemonics but that are not actually included in the 6502 instruction set. The difference between an op code and a pseudo-op is that an op code tells a computer's microprocessor to do something, while a pseudo-op tells an assembler to do something while it is assembling a program. Thus, although op codes and mnemonics may resemble each other, they are actually quite different. Since op codes are part of the 6502 instruction set, they never vary from assembler to assembler. But the pseudo-ops used by one assembler are often different from those used by another assembler, although there are many nearly standard pseudo-ops.

The ORG directive in line 1 of the ADDNRS.SRC program is one example of a pseudo-op. Almost all assemblers use an origin directive of one kind or another, but some use an equal sign (=) instead of the abbreviation ORG. In a 6502/65C02 assembly-language program, the ORG directive (or its equivalent) is used to tell the assembler where the assembly-language program will be stored in memory after it is converted into machine code.

The Apple ProDOS assembler recognizes a number of other pseudo-ops; a number of them will be covered in this chapter, and all of them are discussed in detail in the *ProDOS Assembler Tools* instruction manual.

The Operand Field The operand field in an Apple IIc/IIe source-code listing starts one space after an op-code mnemonic. When an operand is

used, its purpose is to expand an op-code mnemonic into a complete instruction. Some mnemonics—such as CLC, CLD, and RTS—do not require operands. Others—such as LDA, STA, and ADC—do require operands. You will learn more about operands in Chapter 6.

The Comments Field Although comments can start in the label field of an assembly-language program, they can also start to the right of the operand field, in a field of their own. Comments are used in assembly-language programs in much the same way that remarks are used in BASIC programs; they don't affect a program in any way, but are used to explain programming procedures and to provide eye-saving white space in program listings.

The ADDNRS.SRC Program Line by Line

Now that we've looked at the ADDNRS.SRC program field by field, let's examine it line by line. When the program has been typed as shown in Program 4-1 and then listed using the L command, it will appear on the screen as illustrated in Program 4-3.

Program 4-3
THE ADDNRS.SRC PROGRAM
(Apple ProDOS Assembler Version)

```
 1                      ORG     $1000
 2      ;
 3      ; ADDNRS.SRC
 4      ;
 5      ADDNRS          CLD
 6                      CLC
 7                      LDA     #2
 8                      ADC     #3
 9                      STA     $0300
10                      RTS
```

Here is a line-by-line analysis of Program 4-3.

Line 1: The ORG Directive Line 1 of Program 4-3 is the *origin line* of the ADDNRS.SRC program. When an ORG line is used in an assembly-language program, it tells where the program will be stored in RAM after it has been converted into object code. Not every assembler requires an ORG directive (the directive is optional, for example, in programs written for the Merlin Pro and ORCA/M assemblers). But the Apple ProDOS assembler does require every program to have an origin directive. If there is no origin line in a program written for the Apple ProDOS assembler, the assembler will not generate any source code.

It's not always easy to decide where in memory a program should start, particularly if you're new to assembly-language programming. Your computer has many blocks of memory that you can't use for assembly-language programs because they're reserved for other uses, such as holding your computer's operating system, its disk operating system, and its BASIC interpreter. Even the assemblers that were used to write the programs in this book take up blocks of memory, and if you accidentally overwrite the program that runs your assembler, your assembler won't be able to assemble your program.

Deciding where to store a program in a computer's memory is a tricky job, with many variables to be taken into account. This topic will be covered in more detail in Chapter 11, which is devoted solely to memory management. For now, it's sufficient to remember that if you type the programs in this book exactly as they are written, they won't venture into any reserved areas of your computer's RAM.

Lines 2 Through 4: Remarks The semicolons that precede the text in lines 2 through 4 show that these lines are made up solely of remarks. Line 3 gives the name of the program. Lines 2 and 4 only serve to surround the title line with space to make it easier to read.

It's good programming practice in assembly language—as well as in most other programming languages—to use remarks and comments freely. You will find many explanatory comments and remarks in the programs in this book.

Line 5: A Label and an Op Code As you can see, the label field in this line has been used to assign the label ADDNRS to the entire ADDNRS.SRC program. The program would work perfectly without any label at all but, as previously noted, a program becomes more useful when it does have a label, since it can then be accessed by that label and used as part of a larger program. Therefore, it's a good idea to give labels to important routines and subroutines. A label not only makes a routine easier to reference, it also serves as a reminder of what the routine does (or, until your program is debugged, what it's supposed to do).

As previously mentioned, the 65C02/6502B chip can perform arithmetical operations on two kinds of numbers: ordinary binary numbers and binary-coded decimal (BCD) numbers. Much more information on binary and BCD numbers will be provided in Chapter 10. It's sufficient now to recall that binary arithmetic is the kind most often used in 65C02/6502B programs and that the mnemonic CLD clears the decimal flag of the 65C02/6502B microprocessor so that calculations can be

carried out using binary numbers. (The command to set the decimal flag is SED.) It's not necessary to clear the decimal flag before every arithmetical operation in a program, but you should clear it before the first addition or subtraction operation in a program. That way you'll never worry about whether it may have been set during a previous program.

Line 6: An Op Code Before you carry out an addition operation, clear the carry flag (CLC). The carry bit is affected by many kinds of operations and it's best to be safe. It takes only one half millionth second and one byte of RAM to clear the carry flag. Compared to the time and energy that debugging can cost, that's a bargain.

Line 7: Op Code and Operand (LDA #2) Before an addition operation takes place, the accumulator has to be loaded with one of the numbers that is to be added. In the ADDNRS program, LDA #2 is the statement that loads the accumulator. In this statement, the "#" sign in front of the number 2 means that the 2 is a literal number, not an address. If the instruction were LDA 2, the accumulator would be loaded with the contents of memory address 0002 rather than with the number 2.

Line 8: Op Code and Operand (ADC #3) In this line, the statement ADC #3 is used to add the number in the accumulator to the literal number 3. The mnemonic ADC means "add with carry." When this instruction is used in a program, it adds the value specified in the operand to the value of the accumulator, plus the value of the carry bit of the processor status register. In this case, adding the carry bit has no effect, since the carry bit was cleared prior to our addition operation. There is no 65C02/6502B assembly language instruction that means "add without carry." If you did want to add a number without a carry, though, you could do it by clearing the status register's carry flag and then performing an "add with carry" operation.

Line 9: Op Code and Operand (STA $0300) This line stores the contents of the accumulator—in this case, the sum of 2 and 3—in memory address $0300. Note that the symbol "#" is not used before the operand ($0300) in this instruction, since in this case the operand is a memory address, not a literal number.

Line 10: An Op Code (RTS) If the mnemonic RTS is used at the end of a subroutine, it works like the RETURN instruction in BASIC; it ends the subroutine and returns to the main body of a program, beginning at the line following the line in which the RTS instruction appears. But if RTS is used at the end of the main body of a program, as it is

here, the instruction has a different function. Instead of passing control of the program to a different line, it terminates the whole program and returns control of the computer to the next higher calling program that was in control before the program began—usually a disk operating system (DOS), a keyboard-screen editor, or a machine-language monitor.

Printing Your Program

When you have finished typing your source-code listing, you can print it on a printer in two steps. First you must open a channel to your printer by typing the command **PR#** followed by the device number of your printer. If your computer is an Apple IIc, the device number of your printer will always be 1 (unless the printer is hooked up to the modem port, in which case it will be PR#2). So, if you own an Apple IIc, type **PR#1**. If you have an Apple IIe equipped with a printer, the number that follows the instruction PR# will always be the number of the expansion in which your printer card is installed. (Most Apple IIe's have their printer interface installed in slot 1.) You type the slot number of your printer card.

After you've opened a channel to your printer, you'll have to activate your printer by typing the command PTRON, which stands for "printer on." You can then type either **LIST** or **PRINT** (or the abbreviation **L** or **P**) to get a hard-copy listing (or printout) of your program

Saving Your Program

Each time you write an assembly-language program using the Apple *ProDOS Assembler Tools* package, the source code that you enter winds up in an *edit buffer* that extends from memory address $0800 to memory address $9BFF. That's a 37,887-byte block of RAM, large enough for quite a large source-code program. Unfortunately, however, that's almost exactly the same block of memory that the assembler module in the ProDOS assembler-editor package has to use when it's assembling a program. Since two programs can't occupy the same memory space in a computer at the same time, any source code that's in the Apple ProDOS edit buffer has to be cleared from the edit buffer before the ProDOS assembler can be loaded into RAM and the program assembled.

Since clearing a program from memory wipes it out forever, be sure you save a source-code listing on a disk as soon as it's written. Fortunately, the Apple ProDOS assembler-editor makes it difficult to wipe out a source-code listing accidentally. The assembler-editor won't let you load its assembler module into memory (and thus wipe out whatever

may be stored in the edit buffer) until you type the word **NEW**. It's up to you to save any source code that you've been working on before you type **NEW** and erase the contents of the edit buffer.

Since it's so important to save a source-code listing before assembly, let's save the ADDNRS.SRC program on a disk. Then we can assemble the program.

Using ProDOS Commands

It isn't difficult to save a program that has been created with the Apple *ProDOS Assembler Tools* package, since the editor module included in the package supports most of the commonly used ProDOS commands (such as PREFIX, CREATE, DELETE, LOCK, UNLOCK, CAT, LOAD, and SAVE). To invoke any supported ProDOS command while using the *ProDOS Assembler Tools* package, you type the desired command while your assembler-editor is in its command (edit) mode. If you understand the principles of the ProDOS environment, it's easy to save a source-code program using any legal pathname. To save the ADDNRS.SRC program that you've just written, for example, set up whatever ProDOS prefix you want to use, and then type

```
SAVE ADDNRS.SRC
```

When your disk drive has stopped whirring and clicking and the light on it has gone out, check to see if the ADDNRS.SRC program has been saved successfully.

Type the ProDOS command

```
CAT
```

Then, if you see the ADDNRS.SRC program listed on your disk's directory, you can clear the edit buffer by typing

```
NEW
```

That operation will clear your edit buffer. Then you're ready to assemble the ADDNRS.SRC program.

Assembling a Program With the Apple ProDOS Assembler

Once you've cleared your text buffer by typing **NEW**, assembling a source-code listing that has been saved on a disk is easy. Type a line like this:

```
ASM ADDNRS.SRC
```

Your Apple ProDOS assembler will then do two things. It will read the source-code file named ADDNRS.SRC from the disk on which it is stored, and it will also generate an object-code file from the source-code file that it has read and store the object code on a disk. Unless requested to do otherwise, the assembler will store its object-code file on the same disk that the source-code file is stored on and will assign its object-code listing a default pathname that is exactly the same as the source-code listing of the same program, plus the suffix .0.

If you have a source-code program named ADDNRS.SRC saved on a disk, then you can easily assemble that program into executable object code with the Apple ProDOS assembler. Boot up your assembler-editor, make sure its text buffer is clear, and type the line **ASM ADDNRS.SRC**. Your assembler will then generate an executable object-code file from your source-code file and will call its new object-code file ADDNRS.SRC.0.

If you wish, try that procedure now. Next we'll discuss some of the finer points of using the Apple ProDOS assembler. Finally, you'll learn how to execute the object code that your assembler has generated from your ADDNRS.SRC program.

Using Optional Parameters

If you wish, you can instruct your assembler to save its object code on a different disk (or, more accurately, on a different ProDOS directory or subdirectory). You can also instruct your assembler to give its object-code file a name of your choosing rather than a default name.

Suppose, for example, that you had your ADDNRS.SRC program saved on a volume called SRCVOL and wanted to save the program in its object-code version on a different volume called OBJVOL. You can carry out that entire procedure by invoking a couple of optional parameters:

```
ASM /SRCVOL/ADDNRS.SRC,/OBJVOL/ADDNRS.OBJ
```

The source-code program stored on the volume SRCVOL under the pathname ADDNRS.SRC will then be assembled into an object-code file on the volume OBJVOL, under the pathname ADDNRS.OBJ. In addition, a listing of the assembled program will be displayed on your computer screen. Program 4-4 shows how the assembly listing will look on your screen.

Program 4-4
ADDNRS.SRC
An Assembled Listing

```
SOURCE    FILE #01 =>SRCVOL/ADDNRS.SRC
```

```
----- NEXT OBJECT FILE NAME IS OBJVOL/ADDNRS.OBJ
1000:          1000     1            ORG    $1000
1000:                   2 ;
1000:                   3 ; ADDNRS.SRC
1000:                   4 ;
1000:D8                 5 ADDNRS     CLD
1001:18                 6            CLC
1002:A9 02              7            LDA    #2
1004:69 03              8            ADC    #3
1006:8D 00 03           9            STA    $0300
1009:60                10            RTS
#?1000 ADDNRS
** SUCCESSFUL ASSEMBLY := NO ERRORS
** ASSEMBLER CREATED ON 14-DEC-83 15:21
** TOTAL LINES ASSEMBLED     10
** FREE SPACE PAGE COUNT    84
#
```

If you examine Program 4-4 carefully, you'll see that lines 1 through 4 of the ADDNRS program don't generate any object code. The reason is that they contain no op codes or operands. Line 1 contains a pseudo-op—ORG—which performs no function except to tell the ProDOS assembler where to set its program counter when it starts assembling object code. And lines 2 through 4 contain nothing but remarks, which show up on the ADDNRS program's source-code and assembly listings but generate no object code.

Directing a Listing To a Disk Drive or Printer

When the ProDOS assembler-editor assembles an Apple IIc or Apple IIe program, an assembled listing like the one shown in Program 4-4 is ordinarily displayed on the computer's screen. However, the ProDOS assembler has a PR# command that can be used to direct an assembled listing to another output device, such as a printer or even a disk drive. When an assembled listing is written to disk, it is saved in the form of an ordinary ASCII text file, not as a binary file of executable object code.

To store an assembled listing on a disk instead of displaying it on a screen, all you have to do is type the command PR# followed by the slot number of your disk drive, a comma, and an appropriate pathname. If your computer is an Apple IIc, the PR# command will always be followed by the number 6. If you own an Apple IIe, the number that follows the PR# command will be the number of the expansion slot in which your disk-drive card is installed (usually slot 6).

This procedure is not nearly as complicated as it may seem at first glance. Let's assume that your disk-drive card is installed in slot 6. Let's

also suppose that you have a source-code program named /SRCVOL/ADDNRS.SRC and that you want to assemble it into an object-code program named OBJVOL/ADDNRS.OBJ. Finally, let's suppose that you also want to save an assembled listing of the same program under the pathname /SRCVOL/ADDNRS.ASM. You could do all of that by typing these two lines:

```
PR# 6,/SRCVOL/ADDNRS.ASM
ASM /SRCVOL/ADDNRS.SRC,/OBJVOL/ADDNRS.OBJ
```

Just type those two lines into your assembler and, if you have the necessary volumes in the right disk drives, your assembler should:

- Assemble the source-code program called /SRCVOL/ADDNRS.SRC.
- Save the assembled listing under the pathname /SRCVOL/ADDNRS.ASM.
- Save the object code generated by the assembler under the pathname /OBJVOL/ADDNRS.OBJ.

If you haven't already done so, assemble and save an ADDNRS.OBJ program. First make any ProDOS prefix adjustments that you might have to make, and then type the line

```
ASM ADDNRS.SRC,ADDNRS.OBJ
```

Suppressing the Generation Of Object Code

If you want to display an assembler listing on your screen or print it on paper without generating an object-code file, it's easy to do. To suppress the generation of an object-code file, type a comma and the symbol "@" following the line containing your ASM command.

```
ASM ADDNRS.SRC,@
```

Your Apple ProDOS assembler will then print an assembly listing of your ADDNRS.SRC program but will not generate an object-code file.

The Merlin Pro Assembler

The rest of this chapter will be devoted to discussions of the Merlin Pro and ORCA/M assemblers. So if you're working with an Apple ProDOS assembler, you may want to skip the rest of this chapter and move right

on to Chapter 5. If you own an ORCA/M assembler, you may want to go directly to the final section of this chapter, where you will learn how to write and assemble a program using the ORCA/M system.

There are many similarities between the Apple ProDOS assembler and the Merlin Pro. Both assemblers use relative line numbers and many of the same editing functions. The pseudo-op codes used by the two assemblers are almost identical.

There are also many differences between the Merlin and Apple assemblers. The Merlin Pro, unlike Apple's ProDOS assembler, is menu-driven. And Merlin is fully compatible with the instruction set of the 16-bit 65802 and 65816 chips, as well as the instruction sets of the Apple IIe's 6502B chip and the 65C02 chip used in the Apple IIc. Merlin also has a number of other special features—for example, an excellent linking utility and a number of additional editing commands.

The Merlin Pro also comes with an extensive library of *macros*—prewritten assembly-language routines that can be inserted automatically into user-written programs and can save a lot of programming (and typing) time. Another bonus that you get with Merlin is a very fine *disassembler* called the Sourceror.

A disassembler is a utility for converting machine-language programs back into source code. If you want to find out how a machine-language program runs, you can use a disassembler to do a little "reverse engineering." The Sourceror disassembler that comes with Merlin is one of the best disassemblers available.

In addition to the Sourceror, the Merlin Pro package includes an even more specialized utility called the *Floating Point Sourceror*, or *Sourceror.FP*. With the Sourceror.FP you can get a complete, 150-page, fully labeled disassembled listing of the resident Applesoft BASIC package that's built into your Apple IIe or IIc.

Merlin's Modules

When you buy a Merlin Pro package, you get two disks. Both disks can be copied, so as soon as you take them out of their sleeves you should make backup copies. One of the disks is formatted using DOS 3.3 (the disk operating system that Apple used for its II-series computers until ProDOS was developed). The other disk is a ProDOS disk, and that's the one we'll discuss in this section. The programs on the ProDOS disk are divided into five modules:

- An *executive module*
- An *editor module*
- An *assembler module*

- A *linker module*
- A *command interpreter module.*

When Merlin is in its executive mode, a master menu is displayed on the screen, and any of the assembler's other modes can be selected from the menu. If you want to create or edit an assembly-language program, you can select Merlin's editor module. To assemble a program, you can use the assembler module. Use the linker module to link long programs together. You can also invoke most of Apple's ProDOS file functions from Merlin's command interpreter module.

Merlin's Menu

Boot Merlin from a disk; the first screen display will look something like this:

```
1
MERLIN-PRO 1.34
By Glen Bredon
@
(OF
C: Catalog
L: Load source
S: Save source
A: Append file
D: Disk command
E: Enter ED/ASM
O: Save object code
@: Set date
Q: Quit

%_
```

All of the options on this menu are explained in detail in the instruction book that comes with the Merlin Pro assembler package. For now, you should note that Merlin can do quite a few things in executive mode, including loading and saving source code and object code and listing the contents of a disk (using the menu's C command). You can read and write text files using Merlin's R and W menu commands. You can even format disks, scratch files from disks, and perform numerous other disk-management functions using the executive menu's X command.

At the bottom of your screen you will see a "%" sign followed by a flashing cursor. The "%" prompt always indicates that Merlin is in executive mode. When the assembler is in editor mode, the prompt changes to a colon. When it's in monitor mode, the prompt is a dollar sign.

When you've located the "%" prompt, type the letter

```
E
```

for "enter editor/assembler mode." Then press a carriage return to activate Merlin's editor module. To let you know that it's in edit mode, Merlin will print a ":" prompt on your screen. Then type

```
A
```

for "add." You will see the number 1 appear on your screen. That 1 is intended to be used as the first line number in a source-code program. Merlin automatically generates line numbers, beginning with the number 1 and progressing in increments of 1. The number 1 on your screen means that Merlin is ready to accept the first typed line of a source-code program.

If your assembler is working properly, the 1 on your screen is followed by a space and a flashing cursor. Beginning at the cursor location, type an asterisk—without any additional spaces in front of it. Line 1 of your program will look like this:

```
1 *
```

Type a return and Merlin will advance automatically to line 2. Following the numeral 2, again without any extra spacing, type

```
*ADDNRS.S
```

and press the RETURN key. When Merlin advances to line 3, type another asterisk.

You should now see

```
1 *
2 * ADDNRS.S
3 *
4
```

There is one obvious difference between the above lines and the first three lines of the ADDNRS program that was created using the Apple ProDOS assembler. In the Apple ProDOS version of the program, comments in the label field can begin with either an asterisk or a semicolon. In the Merlin version, label-field comments always start with asterisks. If you try to set off a Merlin comment with a semicolon, your editor will accept it but your comment will automatically be tabbed all the way to the comment field (the fourth field on your screen).

When you've finished typing line 4 of the ADDNRS.S program, press your carriage return and you'll see Merlin's ":" prompt again. Then you can type A (for ADD) again and continue writing your program. If you prefer, you can type some other command.

The Merlin assembler, like the Apple ProDOS assembler, will list a program on your screen if you type an **L** command after the ":" editing prompt. The Merlin will also delete a line if you type a **D**. To list or delete a series of lines using Merlin, you can type the first and last line numbers in that series after your L or D command. Note that when you're using Merlin, you have to separate the two numbers that follow a command with a comma, not with a dash. Here's a Merlin command that will delete Lines 2 and 3:

```
D2,3
```

Try the command. You can then restore the lines you've deleted by using the A command.

Another command that can be used after the ":" prompt is I (for INSERT). Type the letter I after a colon prompt, followed by the number of the line where you want your new line inserted. For example, to insert another asterisk at Line 2 in the ADDNRS.S program, you could type

```
I2
```

Merlin will respond with the number 2.

Now type an asterisk, followed by two carriage returns. Merlin will display its ":" prompt again, and you can type **L** for list. Merlin will list your program, and you'll see that another line containing an asterisk has been inserted into your program at line 2.

The Merlin Pro assembler, like the Apple ProDOS assembler, uses relative line numbering, so it automatically renumbers each line after you have inserted or deleted one or more lines. Remember to insert or delete higher-numbered lines before you make changes in lower-numbered lines to avoid deleting the wrong lines.

You can now delete that extra asterisk that you've just added to your program. Just press the carriage return to get a colon prompt and type **D2**. Then you can type **L** for LIST; you will get a listing showing that your program has been restored to this condition:

```
1 *
2 * ADDNRS.S
3 *
4
```

In addition to the A, I, and D commands, there's also an R command that you can use to replace a line when you're using the Merlin assembler. And Merlin, like the Apple ProDOS assembler, also has commands

that you can use to copy lines, to find and replace strings, and to perform many other useful functions. You can find details on how to use all of these functions by reading the Merlin Pro instruction manual.

Now let's finish typing the ADDNRS.S program. Program 4-5 shows the program in its entirety.

Program 4-5
THE ADDNRS.S PROGRAM
(As Typed on the Merlin Pro Assembler)

```
 1 *
 2 * ADDDNRS.S
 3 *
 4             ORG    $8000
 5 ADDNRS      CLD
 6             CLC
 7             LDA     #2
 8             ADC     #2
 9             STA     $02A7
10             RTS
11
```

Before you've finished typing the ADDNRS.S program, you'll probably notice that Merlin tabulates columns automatically, dividing a program into easy-to-read fields. Merlin automatically generates a space after each line number, so you don't have to type one. If you do type a space, you wind up in the third column (the one in which all of the three-letter abbreviations appear).

You may also notice that the ORG directive is followed by the number $8000 in the Merlin listing rather than by the number $1000, which appeared after ORG in the Apple ProDOS assembler listing. The reason is that the memory configurations of the two assemblers are quite different. More details on this subject will be provided in Chapter 11.

Listing Your Program

When you've reached Line 12 in your ADDNRS.S program, just press the RETURN key. Then you can type either **L** or **LIST** to see the complete program listed on your computer screen.

A Closer Look At the ADDNRS Program

Program 4-6 is an explosion diagram of the ADDNRS program, as written using a Merlin Pro assembler. In this example (just as in Program 4-2 earlier this chapter), the program is divided into four fields, or columns. Each field has a heading that describes the kind of information it contains.

As you examine this listing, please note that the Merlin assembler does not ordinarily produce listings that use this kind of spacing. These tabulations were used to give you a clear picture of the organization of an assembly-language program.

Program 4-6
THE ADDNRS.S PROGRAM
(Merlin Pro Version)

```
LINE            OP
NO.  LABEL      CODE    OPERAND      REMARKS

 1 *
 2 * ADDDNRS.S
 3 *
 4              ORG     $8000
 5 ADDNRS       CLD
 6              CLC
 7              LDA     #2
 8              ADC     #2
 9              STA     $02A7
10              RTS
11
```

Printing Your Program

When you've finished typing your source-code listing using the Merlin editor, you can print it by typing the command

```
PR#1
```

(This command will work for an Apple IIc, or an Apple IIe with a printer card installed in expansion slot 1. If your printer card is installed in another slot, use the appropriate slot number.) If you want to print listings with page headings, you can also use the command PRTR. For detailed instructions on using these commands, please refer to your Merlin Pro instruction manual.

Assembling and Saving Your Program

To assemble the ADDNRS.S program using the Merlin assembler, simply type the command **ASM** following the ":" prompt. Merlin will then ask you if you want to update your source-code file (with the current date, for example). If you don't want to update your file, you can type **N** (for "no") and Merlin will assemble your source-code program very rapidly.

Before we save the ADDNRS.S program on a disk, let's take time to compare the object code that your assembler has generated with the source code from which the object code was derived. In Program 4-7, in

the column labeled SOURCE CODE, you'll see your source-code listing. In the next column you can see the machine-language version of the program. To the right of that you'll find the meaning of each assembly-language/machine-language instruction.

Program 4-7
THE ADDNRS.S PROGRAM
(Source Code/Machine Code Comparison)

```
SOURCE CODE   MACHINE CODE   MEANING

CLD           D8             Clear status register's decimal-
                                mode flag
CLC           18             Clear status register's carry
                                flag
LDA #2        A9 02          Load accumulator with the number 2
ADC #2        69 03          Add 2, with carry
STA $0300     8D 00 03       Store result in Memory Address
                                $0300
RTS           60             Return from subroutine
```

Now we'll save both your source-code listing and your object-code listing on a disk. First, type **Q** (for QUIT) after the ":" prompt to put your assembler back into executive (menu) mode. Merlin's main menu will reappear. You can then save your source code by selecting menu choice S and your object code by choosing menu choice O.

When you type an **S** or an **O** to save a source-code or object-code listing, Merlin will ask you to name your program. When you name your program, you don't have to add any suffix to indicate whether it's a source-code listing or an object code program because Merlin will do that automatically. If you're saving a source-code listing, the assembler will automatically add an S suffix to your pathname. If you're saving a machine-code listing, Merlin will not add a suffix to the pathname of your object-code.

The ORCA/M Assembler

The ORCA/M assembler-editor package is manufactured by The Byte Works, Inc., of Albuquerque, New Mexico. If you own an ORCA/M or would like to learn about it, continue reading this chapter. If you're using an Apple or Merlin assembler and are anxious to run the ADDNRS program, you should move right on to Chapter 5.

The ORCA/M ("MACRO" spelled backwards) is completely different from the Merlin Pro, the Apple ProDOS assembler, and almost every other microcomputer assembler. ORCA/M, although designed for small

computers, is not a small assembler. The goal of its designers was to create a microcomputer assembler that would work like the assemblers used with big mainframes. They achieved that goal with admirable success. ORCA/M is one of the finest microcomputer assemblers on the market. And for a package that packs such power, it's remarkably easy to operate.

ORCA/M comes on two floppy diskettes, both formatted for ProDOS. Because both sides of each disk are used, ORCA/M is actually a four-disk program. The ORCA program is on side one of disk one. On the flip side of disk one there's a set of ORCA and ProDOS utilities. On side one of disk two, there's a gigantic library of macros—prewritten assembly-language routines that can be automatically inserted into user-written programs and thus save typing and programming time. On side two of disk two, there's a large library of useful subroutines, some of which are designed to work with the macros on side one.

The ORCA/M assembler is beautifully designed. Once you learn how to use the system, it can do much of your programming work for you. With ORCA/M, you can write long programs in the form of short, easy-to-manage modules. When all of your modules are written, you can link your entire program together with an elegant linking system. And ORCA/M fully supports not only the 6502B and 65C02 chips, but also their descendents, the new 16-bit 65802 and 65816.

The weakest link in the ORCA/M system is its instruction manual. Even if you know a lot about assembly language, you might find the manual difficult to follow. The following discussion is intended to help you understand enough to get started programming with ORCA/M.

Booting the ORCA/M Assembler

Both disks in the ORCA/M package can be copied. Make copies and put your original disks away for safekeeping. (I strongly recommend that you equip your Apple with at least two disk drives before trying to use the ORCA/M program. It's a very sophisticated program and does a lot of switching back and forth between disk and memory while it's running. A pair of disk drives—or even a hard disk drive—can save you a lot of time when you're using the ORCA/M program.)

Place your duplicate ORCA/M program disk in the disk drive that's built into your Apple IIc, or in drive 1 if you own an Apple IIe. Then initialize your duplicate program disk as instructed in the *System Initialization Manual* that came with your ORCA/M assembler.

When you've initialized your program disk, put your duplicate program disk in drive 1 and place an empty but formatted ProDOS disk in

drive 2. Then boot your duplicate program disk as you would any other ProDOS diskette. You will then see a title screen.

```
1
ORCA/HOST 4.0
Copyright (C) January 1985
By The Byte Works, Incorporated
@

     #_
```

The "#" prompt that follows the title indicates that the ORCA/M assembler is in a mode called the *command processor* mode. When ORCA/M is in its command processor mode, it will accept a list of commands called *monitor* commands. This is a confusing name, since the ORCA/M monitor commands bear no relationship to your Apple's built-in machine-language monitor.

Descriptions of the monitor commands that can be issued to the ORCA/M command processor can be found on pages 59 through 72 of the ORCA/M instruction manual. In this section, we will focus on only three of those commands: PREFIX, EDIT, and NEW.

PREFIX is a very convenient command since it can be used to change the ProDOS volume from which ORCA/M accesses files. If you have a two-drive system, you can instruct ORCA/M to get data from drive 2 by typing

```
PREFIX .D2
```

as soon as the title screen comes on. From then on, the assembler will load and save source-code and object-code files using drive 2.

The NEW and EDIT Commands

When you've booted your ORCA/M disk and its title screen is displayed, type the command **NEW** following the "#" prompt on your screen. The NEW command means that you want to get into the editor mode but don't have a file to load. If you want ORCA/M to load a file into memory and then go into edit mode to edit the file, type the word **EDIT** after the "#" prompt, followed by the pathname of the requested file.

```
#EDIT PATHNAME
```

When you type the monitor command NEW (or an EDIT command followed by a pathname), your disk drive will start spinning and you will soon see a new display: a blank screen with a flashing cursor in the

upper-right corner and, across the bottom of the screen, a status line printed in inverse video. Examine the status line, and you should see a line of caret (^) symbols at various intervals, plus a line of text. The text will tell what line and column your cursor is on, the amount of memory that you've used so far, and the pathname (if any) of the source-code file that's displayed on your screen.

Full-Screen Editor and Line Numbers

When you start typing a program using ORCA/M, you don't have to worry about line numbers; there aren't any. In addition, the ORCA/M editor is a screen editor, not a line editor, so you can use your cursor arrow keys to move your cursor all over the screen, just as you can with a word processor.

ORCA/M has something else in common with a word processor; it can copy blocks of text from one part of a program to another, it can delete them, and it can move them around. To copy, move, and delete blocks of text, you have to use ESCAPE key sequences that are described in detail in the ORCA/M instruction manual.

The ORCA/M editor also accepts many CONTROL key (^) commands. For example, you can move your cursor to the top of the screen with a ^T, and to the bottom of the screen with ^B. You can go to the first line of a file with a ^F and to the last line with a ^L. You can insert a line anywhere in a program by moving your cursor to the point where you want the line to be and then pressing an ESCAPE-B key sequence. You can delete a line with an ESCAPE-Y.

Other control and escape functions that can be used with the ORCA/M editor are described in the instruction manual and on the quick-reference card that comes with the ORCA/M package, so we won't go into any more detail about them here. You should be ready now for some hands-on experience with the ORCA/M package. Type the listing in Program 4-8. It's called ADDNRS.SC, and it's the ORCA/M version of the ADDNRS program.

Program 4-8
THE ADDNRS.SC PROGRAM
(ORCA/M Version)

```
          KEEP   ADDNRS

ADDNRS    START

          CLD
          CLC
          LDA    #2
```

```
ADC    #3
STA    $0300
RTS

END
```

Notice that the ORCA/M version of the ADDNRS.SC program has a few lines that are lacking in the versions produced on the Apple assembler and the Merlin assembler. The first line of the program, which reads "KEEP ADDNRS", is one such line. There's also an extra line that reads "ADDNRS START" and another that reads "END".

Notice also that the ADDNRS.SC program lacks an ORG line. The reason is that programs written on the ORCA/M assembler don't require origin lines. If you provide ORCA/M with an ORG line, it can use it; but if you don't, the assembler will automatically assign your program a starting address of $2000, and that's usually a good place to start an ORCA/M program.

Let's get back to the extra lines in the ADDNRS.SC program.

Keep, Start, and End Directives

As I've mentioned, the first unusual line in the ADDNRS.SC program is the one that reads "KEEP ADDNRS". KEEP is a pseudo-op code that's often used in ORCA/M programs. If a KEEP directive precedes a program, ORCA/M will save the object code that is produced when the program is assembled. The word used as the operand of the KEEP directive will be the pathname under which the object code is saved. If the KEEP directive is not used, the program's object code will not be saved.

START and END are two other directives often seen in ORCA/M programs. They are used to delineate the starting and ending points of named code segments that can later be linked together to form longer programs. Since the START directive is used to identify a code segment by name, it requires a label—in this case, ADDNRS, the name of the program. The END directive requires no label.

Comments in ORCA/M Programs

The way in which ORCA/M handles comments deserves special mention. In an ORCA/M program, a comment line can be preceded by a number symbol (#), a semicolon, or an exclamation point, but blank lines in ORCA/M programs are also treated as comments. A blank line has only one purpose, of course—to make a program easier to read by breaking up segments and providing space. But blank spaces can be quite effective in a program listing, as you can see in Program 4-8.

Assembling the ADDNRS.SC Program

When you've typed the ADDNRS.SC program, hit a CONTROL-Q (the CONTROL key and the Q key simultaneously), and ORCA/M should respond with this screen display:

```
======================================

        <R> Return to Editor
        <S> Save to the Same Name
        <N> Save to a New Name
        <L> Load Another File
        <E> Exit Without Updating

======================================

        Enter Selection: _
```

As you can see from the entries on this menu, its primary purpose is to enable you to load and save programs. Since you've written a program that doesn't yet have a name, the best way to save it would be to select menu choice R ("Save to a New Name"). Press your computer's N key. The bottom line on your screen will change from "Enter Selection:" to "File Name:". You can then type in **ADDNRS.SC**, which is the pathname under which your ADDNRS source-code file will be saved.

When you've saved the ADDNRS.SRC program, you can put ORCA/M back into monitor mode by typing **E** (for "Exit Without Updating"). The ORCA/M "#" prompt will then appear on your screen. Type the word

```
ASSEMBLE
```

following the "#" prompt. The ADDNRS.SC program will then be assembled into machine language, and its object code will be saved to disk automatically, under the pathname ADDNRS (the word that was used in the program's source-code listing as the operand of the KEEP directive).

Printing an Assembly Listing

When you assemble an ORCA/M program, you can send its assembly listing to a printer rather than to the screen with a statement such as

```
ASSEMBLE >. PRINTER
```

In the next chapter, you'll learn how to run an assembled program.

5

Running an Assembly-Language Program

Once you've written and assembled an assembly-language program, what you have is a machine-language program. There are several methods that you can use to execute machine-language programs.

You can run a machine-language program using the ProDOS/BASIC command BRUN, which exists solely for the purpose of loading and running machine-language programs. There's also another DOS command, BLOAD, which will load a machine language into RAM but not execute it. BLOAD is designed to load your machine code so that you can execute it later, whenever you like and in whatever way you like. The BRUN and BLOAD commands are quite versatile; you can use them as direct commands or from within a BASIC program. You can also invoke the BLOAD command from many software packages,

including the Apple ProDOS assembler, and from the Merlin Pro assembler and the *Bugbyter* debugging utility that comes in the Apple ProDOS Assembler Tools package.

Once you've loaded a machine-language program into RAM, you can run it using your computer's built-in machine-language monitor. The Apple IIc/IIe monitor has a special command—the G command—for running machine-language programs.

You can also execute a machine-language program using the *Bugbyter* debugging utility. The *Bugbyter* is similar to the Apple IIc/IIe monitor, but it has many additional functions.

You can execute a machine-language program from an Applesoft BASIC program using the CALL command, the USR(X) function, or the G command. (The Apple ProDOS Assembler Tools package also contains a utility called a relocating loader that can help you run assembly-language programs from BASIC programs. The Merlin Pro assembler and the ORCA/M assembler can also produce relocatable machine code that can be called from BASIC programs.)

You can even make a machine-language program load and execute automatically as soon as a disk on which it has been stored is booted. This procedure can be performed easily by using the ProDOS/BASIC STARTUP command.

Loading a Machine-Language Program

Before you can run a machine-language program, you have to load it into your computer's memory. There are several different ways to load the program. From either a ProDOS or a BASIC environment, for example, you can both load and run a machine-language program by typing

```
BRUN ADDNRS.OBJ
```

If this line were entered as a direct BASIC/ProDOS command, the computer would check all available disk volumes for a machine-language program with the pathname ADDNRS.OBJ on the default drive. If the computer found such a program, it would load and execute the program. If the computer failed to find such a program, you would receive an error message.

The same thing would happen if your computer encountered the following line while running a BASIC program:

```
10 PRINT CHR$(4);"BRUN ADDNRS.OBJ"
```

(The syntax used in this line is the standard syntax for issuing ProDOS commands from BASIC programs. More details on this subject can be found in Apple's *BASIC Programming With ProDOS* manual and other instruction books on ProDOS and Applesoft programming.)

Load Now, Run Later

If you want to load a machine-language program into your computer's memory but don't want to run the program immediately, you can use the BASIC/ProDOS command BLOAD. To use BLOAD as an immediate command, just type

```
BLOAD ADDNRS.OBJ
```

To use the BLOAD command from a BASIC program, use the following format:

```
10 PRINT CHR$(4);"BLOAD ADDNRS.OBJ"
```

The machine-language program will then be loaded into your computer's memory but will not be executed.

The BLOAD command can be invoked not only from ProDOS or a BASIC program, but also from the Apple ProDOS assembly-language editor and the Apple *Bugbyter* debugging utility. More information on these two methods of using the BLOAD command is provided later in this chapter.

Executing a Machine-Language Program

Once you've loaded a machine-language program into RAM, you can execute the program from the Apple IIc/IIe monitor, from the Apple *Bugbyter* debugging utility, or from a ProDOS or BASIC environment. Let's start with the Apple IIc/IIe monitor.

The Apple IIc/IIe Machine-Language Monitor

One of the easiest ways to execute a machine-language program is with your Apple's built-in monitor. The Apple monitor is an extremely useful programming tool that has been built into Apple computers since they

first went on the market. With this monitor you can peek into your computer's memory and can list—and even change—the contents of its memory locations. The Apple monitor can translate the contents of memory locations into assembly language and can display assembly-language listings on your computer screen. It can also move blocks of memory from one location to another and can compare blocks of memory to see if they match. And—most important for our purposes at the moment—the Apple monitor can execute machine-language programs.

The Apple monitor can be invoked easily from Applesoft BASIC. You can also call the monitor from both the Apple ProDOS assembly-language editor and the Merlin assembler-editor. You can transfer control to your computer's monitor with either package by typing the command **MON** and pressing RETURN.

You can invoke the monitor from BASIC by typing **CALL -151** or **CALL 65385** and then pressing RETURN.

Using the Apple IIc/IIe Monitor

Now let's load a machine-language program into RAM and execute it using your computer's built-in monitor.

Turn on your computer, get its BASIC interpreter up and running, and put the disk containing your ADDNRS.OBJ program into the disk drive. Make any ProDOS prefix adjustments that are necessary and then type **BLOAD ADDNRS.OBJ** and press RETURN. Next, type the command **CALL -151** and press RETURN again. You should then see the asterisk (*) prompt that lets you know your monitor is up and running.

When you see the "*" prompt, type **4096** and press RETURN. That number is the decimal equivalent of the hexadecimal number $1000, which should be the starting address of your ADDNRS.OBJ program.

When your disk drive stops spinning, you can find out whether your ADDNRS.OBJ program has been successfully loaded or not by typing

```
*1000L
```

(The asterisk will be there already.)

The L in the monitor command 1000L stands for LIST. When you type the command followed by a carriage return, your monitor will disassemble up to 20 lines of machine-language instructions beginning at memory address $1000 and list them on your computer screen. "Disassemble" means to translate a machine-language instruction back into assembly language. When your monitor disassembles the program that is now stored in the block of memory starting at $1000, you will see a screen display that looks like the following:

A MONITOR DISASSEMBLY OF THE ADDNRS PROGRAM

```
COLUMN 1   COLUMN 2      COLUMN 3

1000-      D8            CLD
1001-      18            CLC
1002       A9 02         LDA #$02
1004-      69 03         STA #$03
1006-      8D 00 03      STA $0300
1009-      60            RTS
```

Below the displayed addresses, you'll see 14 lines displaying the contents of a series of addresses that are not part of the ADDNRS program. Your monitor's L command always creates a 20-line screen display, but only six of those lines are needed to list the ADDNRS program. The contents of the extra addresses can vary, depending upon what kinds of programs you were running on the computer before you invoked your monitor. The extra addresses may contain nothing but a string of 0's, each followed by the mnemonic BRK (an assembly-language instruction that equates to a 0 in machine language). Alternatively, the extra addresses may hold bits and pieces of a previously loaded program. Whether the addresses used by the ADDNRS.OBJ program have been previously occupied by another program or not, though, they're all user-addressable RAM, which means that they can be freely overwritten. As long as the ADDNRS program ends with an RTS instruction (which it does), the contents of the addresses that follow it don't matter.

The numbers in Column 1 of the illustration are the hexadecimal memory addresses in which the ADDNRS.OBJ program is stored. The numbers in Column 2 are hexadecimal machine-language instructions that are stored beginning at the addresses listed in Column 1. As you can see, an assembly-language listing of the complete ADDNRS.OBJ program is provided in Column 3.

Running a Program Using the Apple IIc/IIe Monitor

Once you've called up a disassembled listing of a program and have examined it, you know that the program has been loaded successfully into your computer's memory. But you won't know whether the program actually works until you've executed it.

It's very easy to execute a program using the Apple monitor. Before we do it, though, let's take a look at the contents of another memory register—the one at memory address $0300. Memory register $0300, you may recall, is the one that is used to store the sum of 2 and 3 in the

ADDNRS program. Let's look into that register before we run the ADDNRS program.

Type

```
*0300
```

Your monitor display of the contents of memory register $0300 should look something like this:

```
0300- 00
*
```

You will know from that response that memory register $0300 contains a 0. If $0300 doesn't contain a 0, you can place a 0 in it by typing the number **0300** followed by a colon and a 0.

```
*0300:0
```

You can use the colon command to place any 8-bit value in any memory register in RAM. You may want to experiment a little now, using the colon command to change the contents of memory address $0300 to different values and then checking to see whether it worked. When you've finished experimenting, be sure to clear register $0300 to 0.

Now we can execute the ADDNRS program using the Apple IIc/IIe's monitor. Following the asterisk prompt, type the number 1000 (the starting address of the program) and the letter G.

```
*1000G
```

Then press a carriage return. The ADDNRS program should then run.

If everything is working correctly, you won't see much happening when the ADDNRS program runs; all you're likely to notice is another asterisk prompt on your screen. If all is as it should be, though, something will definitely have happened. To find out what, just type **0300** after your monitor's asterisk prompt. You should then see

```
0300- 05
```

That line will tell you that your computer's monitor has successfully executed the ADDNRS.OBJ program.

This book is not really about the Apple IIc/IIe monitor, so we only touch on the features of the monitor in this chapter. The Apple IIc/IIe monitor can also change, move, and compare the contents of small or large blocks of memory, display and change the contents of the 65C02/

6502B chip's registers, and even perform 8-bit hexadecimal arithmetic. Details of these and other features of the Apple IIc/IIe monitor can be found in your computer's Reference Manual.

The ADDNRS.SR2 Program

Let's take a look at a couple of machine-language programs that not only demonstrate how programs can be loaded and saved, but also show how a program, once loaded into memory, can provide an output to the screen. Look at Program 5-1, a new and improved version of the ADDNRS.SRC program. It was written using the Apple ProDOS assembler, but it will also run on the Merlin Pro and (with the minor modifications explained in Chapters 4 and 5) can easily be made to work on the ORCA/M. Since it's based on the ADDNRS program, I've named it ADDNRS.SR2. However, if you own a Merlin Pro assembler, which automatically assigns the suffix ".S" to every source-code listing it saves, you can call the program ADDNRS2.S.

Program 5-1
ADDNRS.SR2

```
 1  ORG $1000
 2 *
 3 * ADDNRS.SR2
 4 *
 5 HOME EQU $FC58
 6 PRHEX EQU $FDE3
 7 *
 8 ADDNRS JSR HOME
 9  CLD
10  CLC
11  LDA #2
12  ADC #3
13  STA $0300
14  JSR PRHEX
15  RTS
```

The ADDNRS.SR2 program does everything that its predecessor did, plus a little more. First, it clears the screen, which the ADDNRS.SRC program didn't do. Then, like ADDNRS.SRC, it adds the numbers 2 and 3 and stores their sum in memory register $0300. However, the new program also has an output function that prints the sum of 2 and 3 on the screen.

The easiest way to type the ADDNRS.SR2 program is to load your original ADDNRS.SRC program into the computer and then use the editing features of your assembler to expand it into its new form. You can then save the edited version of the program under the new pathname ADDNRS.SR2.

A Symbol Table

When you start expanding the ADDNRS.SRC program into ADDNRS.SR2, the first two additions you'll encounter are the ones in lines 5 and 6. Together, these two lines make up what is known in the world of assembly-language programming as a symbol table. Symbol tables are used in assembly-language programs to define constants and variables; most symbol tables also include line labels. Most assembly-language programs have symbol tables, and you'll find a symbol table in every program that you'll be working on from now on in this book.

In the ADDNRS.SR2 program, the symbol table in lines 5 and 6 defines only two values, and they are both constants. In line 5, a constant called HOME is defined as the hexadecimal value $FC58. In line 6, a constant called PRHEX is defined as the hex value $FDE3. The two constants are defined with the help of a pseudo-op code that is written EQU, which stands for "equals." (In a symbol table it isn't necessary to use the symbol "#" to indicate that a value is a literal number because every value listed there is assumed to be a literal number.) But what do the values and labels in this short symbol table mean?

You will recall a short assembly-language program called HI.TEST that was presented in Chapter 1. HI.TEST made use of a convenient machine-language subroutine that is built into the Apple IIc and the Apple IIe. That subroutine, often called COUT in assembly-language programs, is built into ROM in both the Apple IIc and the Apple IIe. It starts at memory address $FDED, and it can be incorporated into any assembly-language program with a simple JSR command.

When the COUT subroutine is called, it expects the Apple ASCII code for a typed character to be stored in the 65C02 6502B accumulator. If the value of a valid character is in the accumulator, that character will be printed on the screen.

COUT is only one of a number of useful machine-language subroutines that are built into the Apple IIc and Apple IIe computers. All of these subroutines are listed and described in the Apple IIc and Apple IIe reference manuals. If you continue to study assembly language after you finish this book, you should get to know all of these subroutines very well. Many of them perform extremely useful functions, and all of them

are designed to be invoked with one simple assembly-language instruction: JSR. Since all of these routines are prewritten, they can save tremendous amounts of labor, energy, and time.

Before we continue, I suggest that you assemble the ADDNRS.SR2 program under the filename ADDNRS.OB2 and store it on a disk in both its source-code and object-code versions. Then we can work with the program during the rest of this chapter.

When you execute your expanded version of the ADDNRS program, you'll see very clearly how the built-in routines labeled HOME and PRHEX are used. The HOME routine, which appears in Line 8, does the same thing that the instruction sequence HOME does in Applesoft BASIC. It clears the screen and places the cursor in the upper-left corner of your computer's text display.

Lines 9 through 13 of the ADDNRS.SR2 perform exactly the same functions as lines 5 through 9 of the original ADDNRS.SRC program. When these lines are executed, the ADDNRS.SR2 program adds the numbers 2 and 3 and stores their sum in memory register $0300.

A Subroutine That Is Displayed On the Screen

Line 14 is another new addition to the original ADDNRS program. In this line, the built-in subroutine PRHEX is used to display the sum of 2 and 3 on the screen. If you consult the list of built-in subroutines in your computer's reference manual, you'll discover that PRHEX actually displays one hexadecimal digit on the screen. But, since the digit 5 (the sum of 2 and 3) happens to be written the same way in the hex and decimal systems, it doesn't matter in this case that the output of the PRHEX routine is actually a hexadecimal number.

In line 15, the ADDNRS.SR2 program ends the same way its predecessor did, with the traditional RTS instruction.

The BRUN Command

Once you've typed and assembled a program, the easiest way to run it is with the ProDOS/BASIC BRUN command. To run the ADDNRS.OB2 program using the BRUN command, store the program on a startup disk (one with both ProDOS and a BASIC.SYSTEM file recorded on it). Then boot the disk and type the command

```
BRUN ADDNRS.OB2
```

As soon as your startup disk boots, you will see the number 5 displayed in the upper-left corner of your screen.

Creating a Startup Program

If your ADDNRS.SR2 program runs well with the BRUN command, you can easily fix it so that it will boot and run automatically whenever you turn your computer on. With the disk that contains the ADDNRS.SR2 program still in your disk drive, just type

```
RENAME ADDNRS.SR2,STARTUP
```

Then turn your computer off without changing disks, and after waiting 30 seconds, turn it on again. Your ADDNRS.SR2 program, now renamed STARTUP, should now load itself and run automatically. If you reboot the disk with the ADDNRS.SR2 right now, and if everything works the way it should, your disk drive will whir and click the way it always does when it's loading a program, and then you'll see the number 5 displayed at the top of your computer screen.

Devising a Better Startup Program

You probably won't find many uses for a startup program that loads itself and then prints the sum of 2 and 3 in the upper-left corner of a computer screen. But the same principles that we used to turn the ADDNRS program into a startup program can be employed to create more useful kinds of startup routines. Program 5-2, for example, is an assembly-language program that will boot a disk and print a short greeting on your video monitor.

Program 5-2
AN ASSEMBLY-LANGUAGE HELLO PROGRAM

```
 1 *
 2 * HELLO.SRC
 3 *
 4  ORG $1000
 5 *
 6 COUT EQU $FDED ;SCREEN-PRINTING ROUTINE
 7 HOME EQU $FC58 ;ROUTINE TO CLEAR SCREEN
 8 *
 9 HELLO JSR HOME ;CLEAR SCREEN
10  LDA #$C8 ;LOAD THE LETTER 'H'
11  JSR COUT ;PRINT IT
12  LDA #$C9 ;LOAD THE LETTER 'I'
13  JSR COUT ;PRINT IT
14  LDA #$A1 ;LOAD EXCLAMATION POINT
15  JSR COUT ;PRINT IT
16  RTS  ;END OF PROGRAM
```

Notice that Program 5-2 is a slightly improved and expanded version of the program called HI.TEST.SRC that was presented in Chapter

1. With what you now know about assembly language, you shouldn't have much trouble understanding it. It uses machine-language routines that are built into your Apple to clear the screen and type the message "HI!" It ends, as most good assembly-language programs do, with an RTS instruction.

Type the program, assemble it (as HI.TEST.OBJ) and execute it; it will clear your screen and place the cursor in home position. Then it will display the word

```
HI!
```

on your computer screen.

That message isn't very long, but we're not yet ready to display longer messages on a screen, since we haven't discussed how text strings are handled in assembly language. Short as it is, though, the greeting "HI!" does make some sense, which is more than can be said for the cryptic 5 produced by the ADDNRS program. So perhaps we should give the ADDNRS.OBJ program its original name back and turn HELLO.OBJ into a startup program.

To restore ADDNRS.OB2's original name, type

```
RENAME STARTUP,ADDNRS.OB2
```

Then you can type

```
RENAME HELLO.OBJ,STARTUP
```

Once you've done that, you'll see a nice warm "HI!" in place of a mysterious 5 each time you boot your startup disk. Later on, when we discuss text strings, you'll learn how to display longer startup messages on your screen.

Running an Assembly-Language Program From BASIC

Now that you know how to create an assembly-language startup disk, we're ready to move on to a new subject: mixing BASIC and assembly language.

We know that an assembly-language program can be loaded and executed from a BASIC program by using a line like

```
10 PRINT CHR$(4);"BRUN PATHNAME"
```

This method, however, is not commonly used for running a machine-language program from a BASIC program because it doesn't offer much in the way of versatility. When you use the BRUN command in a BASIC program, the machine-language program that it calls is always executed as soon as it is loaded. This process is not always desirable, since it relinquishes control immediately to whatever machine-language program has just been loaded. A more common technique is to load a machine-language program during the initialization phase of a BASIC program and to execute it later. Fortunately, there is a BASIC command, BLOAD, that will load a machine-language program into memory without running it. Once a program has been loaded into RAM, it can be executed at any time with the help of two other BASIC instructions: the CALL command and the USR(X) function.

To load a machine-language command using the BLOAD function, you type a line using this format:

```
10 PRINT CHR$(4);"BLOAD PATHNAME"
```

When you put that kind of line in a BASIC program, your computer will look on whatever disk it is using (or on any volume you designate) for a machine-language program that has the requested filename. If it finds such a program, it will load it into RAM but will not run it. Once the machine-language program has been loaded, you can run it whenever you like during the course of your BASIC program, with either a CALL command or a USR(X) function.

Using the CALL Command

It's easy to use Applesoft BASIC's CALL command. All you have to do is load a machine-language program into RAM and then type the word CALL, followed by the decimal address of the machine-language program. CALL can be used either as a direct command or from within a BASIC program. When it is invoked from a BASIC program, this is the syntax that is used:

```
20 CALL 4096
```

If this line were included in a BASIC program and a machine-language program were stored in memory beginning at decimal address 4096 (hexadecimal address $1000), control would be transferred to the machine-language program beginning at $1000.

Let's try that now, using the STARTUP program (formerly the HELLO.OBJ program) that you stored on a startup disk. Make sure

that the startup disk is in your disk drive, and then type this two-line BASIC program:

```
10 PRINT CHR$(4);"BLOAD STARTUP"
20 CALL 4096
```

Then type the word **RUN**. Your STARTUP program—which begins at decimal address 4096 (or hex address $1000)—should now run. It should print "HI!" on your screen, and then, since it ends with an RTS instruction, it should pass control of your computer back to BASIC.

Before we move on to the USR(X) function, there is one special feature of the CALL function that is worth mentioning. In Applesoft BASIC, a CALL statement can be expressed as either a positive number or a negative number. If a CALL statement is expressed as a negative number, Applesoft BASIC will automatically add 65536 to it to obtain an equivalent positive address. This unusual feature of Applesoft BASIC can come in handy, since negative CALL addresses are often shorter and easier to remember than their positive counterparts. For example, you can access the Apple IIc/IIe monitor by typing either CALL 65385 or CALL −151. The second statement is much easier to remember, so it is the one most often used. On the other hand, the negative counterpart of the number 4096 (hexadecimal $1000) is the unwieldy negative number −61440, so you probably wouldn't want to use it.

The USR(X) Function

The USR(X) function is a little more complicated than the CALL command—and much more powerful. USR stands for "user-supplied routine," and the purpose of the USR function is to give expert BASIC programmers a method for running high-speed machine-language routines from within BASIC programs. When you know how to program in both BASIC and assembly language, you can use the USR(X) function to perform mathematical computations that would run too slowly if they were programmed in BASIC—or computations that BASIC might not be able to handle at all. Suppose, for example, that you had access to a series of assembly-language routines for performing complex character-animation sequences in high-resolution graphics. You could use the USR(X) function to call those sequences and run them—at machine-language speeds—from within BASIC programs!

When you use USR(X), you can substitute any value you like for the X that appears between the parentheses in the function. Then, when you invoke the USR(X) function, three things will happen.

First, the value that you have assigned to X will be stored automatically in a specific series of addresses in your computer's memory.

Second, the BASIC program that is in progress will be temporarily interrupted and control of your computer will be turned over to a machine-language program—usually a user-written machine-language program—that has been selected in advance.

Finally, once this preselected machine-language routine takes over, it can retrieve the value that has been passed to it via the USR(X) function and can perform any desired computation using that value. The result of that computation can then be stored in the same series of memory registers that the original value came from. Control can then be passed back to BASIC. Note that once a machine-language computation has been performed on the X variable in the USR(X) function, that variable will have a new value when control is passed back to BASIC.

To clarify this procedure, we will examine a couple of illustrative programs. These two programs—one written in BASIC and one written in assembly language—have been interfaced with the USR(X) function. Program 5-3, which I've called DECHEX.BA2, is a BASIC program that uses the USR(X) function to call a machine-language program. Program 5-4, which I've named DECHEX.SR2, is the source-code listing of the machine-code program that the BASIC program calls.

Before we start analyzing these two programs, you might want to type and save the DECHEX.BA2 program and type, assemble, and save the DECHEX.SR2 program. When you save the object-code version of the DECHEX.SR2 program, I suggest that you follow the same convention that we've followed up to now and assign it the pathname DECHEX.OB2.

When used together, Programs 5-3 and 5-4 will convert any signed 16-bit decimal number into a hexadecimal number and will print the hex numbers on your computer screen. As you may notice, these two programs, even when they are combined, are considerably shorter than the BASIC program for converting decimal numbers to hexadecimal numbers that was presented in Chapter 2. In addition, these programs are capable of handling signed numbers, while the DECHEX program in Chapter 2 could handle only positive numbers. There are also other differences between the program in Chapter 2 and Programs 5-3 and 5-4. Before we examine these differences, let's take a close look at these two new programs.

Program 5-3
USING THE USR(X) FUNCTION TO CALL
A MACHINE-LANGUAGE PROGRAM

```
10  REM     *** DECHEX.BA2 ***
20  REM
30  POKE 10,76: REM     POKE JUMP INSTRUCTION ($4C) INTO
    MEMORY REGISTER 10 ($0A)
40  REM     NOW POKE DECHEX.0B2 ADDRESS INTO REGISTERS 11
    AND 12 ($0B AND $0C)
50 HI =  INT (4096 / 256):LO = 4096 - HI * 256: POKE 11,LO:
    POKE 12,HI
60  PRINT  CHR$ (4);"BLOAD DECHEX.0B2"
70  TEXT : HOME : PRINT "DECIMAL-TO-HEXADECIMAL CONVERTER"
80  PRINT : PRINT "TYPE A DECIMAL NUMBER"
90  PRINT "BETWEEN -32767 AND 32767:": PRINT
100  INPUT NR1$: IF  VAL (NR1$) <  -32767 OR  VAL (NR1$) >
    32767 THEN 80
110 X =  VAL (NR1$)
120  PRINT "HEX: ";
130 Y =  INT ( USR (X))
140  PRINT : GOTO 80
```

Program 5-4
A SOURCE-CODE PROGRAM CALLED BY THE USR(X) FUNCTION

```
   1 *
   2 * DECHEX.SR2
   3 *
   4  ORG $1000
   5 *
   6 PRNTAX EQU $F941 ;ROUTINE TO PRINT A AND X IN HEX ON
     SCREEN
   7 FLPINT EQU $E10C ;FLPT/INT CONVERSION ROUTINE
   8 *
   9 DECHEX JSR FLPINT ;CONVERT VALUE IN FLT PT ACCUM TO
     INTEGER
  10  LDA $A1 ;LOW BYTE OF RESULT
  11  TAX  ;PRNTAX GETS LOW BYTE FROM X REG
  12  LDA $A0 ;HIGH BYTE OF RESULT
  13  JSR PRNTAX ;PRNTAX GETS HI BYTE FROM ACCUMULATOR
  14  RTS
x
```

The DECHEX.BA2 Program Line by Line Let's examine Program 5-4, DECHEX.BA2, line by line. We'll start with lines 30 through 50.

```
30  POKE 10,76: REM     POKE JUMP INSTRUCTION ($4C) INTO
    MEMORY REGISTER 10 ($0A)
40  REM     NOW POKE DECHEX.0B2 ADDRESS INTO REGISTERS 11
    AND 12 ($0B AND $0C)
50  HI =  INT (4096 / 256):LO = 4096 - HI * 256: POKE 11,LO:
    POKE 12,HI
```

These three lines carry out an operation that must always be performed before the USR(X) function is invoked. When the USR(X) function is used, it first looks in memory registers 10, 11, and 12 ($0A through $0C in hex notation). If it finds a set of machine-language instructions in those three registers, it will carry them out. If it finds no machine-language instructions in $0A through $0C, it will either return an error message or produce unpredictable and potentially disastrous results. So, before you invoke the USR(X) function, you always have to store three executable machine-language instructions in memory registers $0A through $0C.

In line 30 of the DECHEX.BA2 program, the machine-language op code $4C (76 in decimal notation) is stored in memory register $0A. $4C is a machine-language equivalent of the assembly-language mnemonic JMP. JMP, as you may remember, is similar to the BASIC instruction GOTO. When JMP is used in an assembly-language program, it is always followed by either a memory address or a label that equates to an address. When the JMP instruction is encountered in a program, it causes the program to jump to the specified memory address.

In lines 40 and 50 of the DECHEX.BA2 program, a 16-bit value is stored in memory locations $0B and $0C (11 and 12 in decimal notation). As you can see, the number stored in those two registers is $1000 (or 4096 in decimal notation). (Take a look at Program 5-4, and you'll see what the value $1000 means when it follows a JMP instruction; it's the starting address of the assembly-language program in the DECHEX.SR2 program.)

Now you can see what lines 30 through 50 of the DECHEX.BA2 program do: they set things up so that when the USR(X) function is called later on in the DECHEX.BA2 program, it will transfer control of your computer to the DECHEX.SR2 program.

Next, let's look at lines 60 through 130 of the DECHEX.BA2 program.

```
60  PRINT  CHR$ (4);"BLOAD DECHEX.OB2"
70  TEXT : HOME : PRINT "DECIMAL-TO-HEXADECIMAL CONVERTER"
80  PRINT : PRINT "TYPE A DECIMAL NUMBER"
90  PRINT "BETWEEN -32767 AND 32767:": PRINT
100  INPUT NR1$: IF  VAL (NR1$) <  -32767 OR  VAL (NR1$)
    > 32767 THEN 80
110 X =  VAL (NR1$)
120  PRINT "HEX: ";
```

Line 60 loads a program called DECHEX.OB2. (If you've assembled and saved the DECHEX.SR2 program in the way that I've suggested,

DECHEX.OB2 will be the filename of the object-code version of DECHEX.SR2.)

In lines 70 through 90 of the DECHEX.BA2 program, a title and two lines of instructions are displayed on the screen. The instructions request that a number be typed in and state that the number must be between −32767 and 32767. Note that the original DECHEX program, presented in Chapter 2, was capable of handling only positive numbers, but the DECHEX.BA2 program can convert both positive and negative numbers into hexadecimal notation.

When you use the DECHEX.BA2 program, however, you must pay a certain price for this extra capability. As you may remember, the original DECHEX program could convert any positive decimal number up to 65,535 into hexadecimal notation. The DECHEX.BA2 program can handle positive numbers only half that large; that is, up to 32767. Since it can also handle negative numbers down to −32768, the *total* number of values that the two programs can translate into hexadecimal numbers is the same. (We'll learn more about negative numbers in Chapter 10, which is about assembly-language math.)

In line 100 of the DECHEX.BA2 program, an INPUT command is used to accept a typed number as a string. If the number is less than −32767 or more than 32767, the program refuses to accept it and calls for another input. Once a legal number has been entered, it is converted into a numeric value and is then defined as the value of the X variable in the USR(X) function. In line 120, the message "HEX: " is printed on the screen.

Using the USR(X) Function When the USR(X) function is finally invoked in line 130, it first deposits the value of X in a special set of memory registers called a *floating-point accumulator.* A floating-point accumulator is a block of memory that is used for storing numbers during operations that involve a procedure called *floating-point arithmetic.* In the Apple IIc and the Apple IIe, the floating-point accumulator is situated in memory registers $9D to $A3.

In Chapter 10, which deals specifically with 6502B/65C02 math, floating-point arithmetic will be discussed in more detail. Now we're going to make use of two more of the machine-language subroutines that are built into every Apple IIc and Apple IIe. One of these subroutines, which I call FLPINT, is designed to convert floating-point numbers to 16-bit binary numbers. The other subroutine, which I've named INTFLP, works the other way around: it converts 16-bit binary numbers to floating-point numbers. Here are brief explanations of these two subroutines.

The FLPINT subroutine, which begins at memory address $E10C, will take whatever value is the floating-point accumulator and convert it into a floating-point number, depositing the high-order byte of the floating point number in memory register $A0 and the low-order byte in memory register $A1. To use this subroutine, you use the assembly-language instruction JSR to jump to the subroutine at memory address $E10C. The contents of the floating-point accumulator will then be converted into a two-byte binary number and can be retrieved from memory registers $A0 and $A1.

(Please note that on page 174 of the 1982 edition of the Apple IIe *Applesoft BASIC Programmer's Reference Manual*, the address of the FLPINT routine is given as $E01C. That address is incorrect; if you use it, the FLPINT routine won't work. The correct address is the one given above: $E10C.)

To use the INTFLP routine, you have to place a two-byte binary integer into the accumulator and the Y register of your computer's 6502B/65C02 microprocessor. The low byte of your two-byte number should be in the accumulator and the high byte should be in the Y register. Then just do a JSR to jump to the INTFLP subroutine at memory address $E2F2. Upon return from that subroutine, the floating-point equivalent of the number that you stored in the A and Y registers will be in your computer's floating-point accumulator.

Analyzing the DECHEX.SR2 Program Now we're ready to examine line 130 of the DECHEX.BA2 program.

```
130 Y =  INT ( USER (X))
```

This line invokes the USR(X) function. As we have seen, the USR(X) function first deposits the value of X in your computer's floating-point accumulator. Then it transfers control to whatever machine-language program starts at the address that has been loaded into memory locations $0B and $0C. Since the address of the DECHEX.SR2 program is now stored in those locations (it was placed there in line 50 of the DECHEX.BA2 program), the machine-language program that the USR(X) function now jumps to is the DECHEX.SR2 program.

Now we'll take a line-by-line look at the DECHEX.SR2 program, beginning with lines 6 and 7.

```
6 PRNTAX EQU $F941 ;ROUTINE TO PRINT A AND X IN HEX ON
  SCREEN
7 FLPINT EQU $E10C ;FLPT/INT CONVERSION ROUTINE
```

Here we have another symbol table, a table that assigns labels to the starting addresses of important routines. In the two-line symbol table that appears in lines 6 and 7 of the DECHEX.SR2 program, only two addresses are listed. In line 6, the address of a routine called PRNTAX is defined as $F941. Then, in line 7, there's an address for the FLPINT routine, which converts floating-point numbers into hexadecimal numbers. Since we've already discussed the FLPTINT subroutine, we can now direct our attention to the PRNTAX subroutine in the DECHEX .SR2 symbol table.

PRNTAX, like FLPINT, is built into the Apple IIc and the Apple IIe. PRNTAX is an important routine in a program like DECHEX.SR2, since it can automatically display hexadecimal numbers on your computer's screen. To use PRNTAX, you have to place a 16-bit hexadecimal number in your 6502B/65C02 microprocessor's A and X registers, with the high-order byte in the accumulator and the low-order byte in the X register. Then you can do a JSR to memory address $F941, and the number stored in the A and X registers will be displayed on your computer screen.

Running the DECHEX.SR2 Program Before we start discussing the rest of the DECHEX.SR2 program, it's important to remember what has happened up to now: a number has been typed in on your computer's keyboard and has been stored in its floating-point accumulator. Then, with the help of the USR(X) function, control of your computer has been transferred to DECHEX.SR2.

```
 9 DECHEX JSR FLPINT ;CONVERT VALUE IN FLT PT ACCUM TO
   INTEGER
10  LDA $A1 ;LOW BYTE OF RESULT
11  TAX  ;PRNTAX GETS LOW BYTE FROM X REG
12  LDA $A0 ;HIGH BYTE OF RESULT
13  JSR PRNTAX ;PRNTAX GETS HI BYTE FROM ACCUMULATOR
14  RTS
```

Now we're ready to proceed. The FLPINT subroutine is called in line 9 of the DECHEX program, and the value in the floating-point accumulator—that is, the value that has been typed in—is converted into a two-byte integer. FLPINT stores the low byte of that integer in $A1 and stores the high byte in $A0. Then the FLPINT subroutine ends and transfers control back to the DECHEX.SR2 program.

In line 9 of the DECHEX.SR2 program, the accumulator is loaded with the value in $A1 (that is, the low byte of the two-byte integer returned by FLPINT). Then, in line 10, there's an assembly-language

mnemonic that we haven't encountered before: TAX, which stands for "transfer the value in the accumulator to the X register." In line 10, the TAX mnemonic does just what you'd expect it to do. It moves the value that has just been loaded into the accumulator into the X register. When that operation has been carried out, the low byte of the value returned by FLPINT is in the X register.

There are now only three more lines in the DECHEX.SR2 program. In line 10, the accumulator is loaded with the value stored in memory register $A0 (that is, the low byte of the value returned by FLPINT). Then in line 13 there's a jump to the PRNTAX routine, which combines the value in the accumulator and the value in the X register and then displays the values as a four-digit hexadecimal number on the screen.

The USR(X) Function as a Programming Tool

The USR(X) function is one of the most complicated functions available in BASIC, but it is also one of the most powerful. Once you know how to use it, you can compile whole libraries of machine-language functions and use them at will in BASIC programs. As you now know, it is much easier to program in BASIC than in assembly language. Unfortunately, however, BASIC operates so slowly that it is simply incapable of handling many kinds of routines, especially those involving the use of high-resolution graphics. The USR(X) function is powerful because it allows you, as a programmer, to move back and forth between BASIC and assembly language. If you know how to use the USR(X) function, you can take advantage of the best features of BASIC and assembly language. You can use BASIC for writing the kinds of routines for which BASIC is best suited, and any time you like, you can also use the USR(X) function to call up high-speed, high-performance assembly-language routines.

6

The 6502B/65C02 Instruction Set

Up to now, we've mainly discussed the syntax and the grammar of 6502B/65C02 assembly language, and we'll be covering those topics in more detail in later chapters. But this chapter is important because it is about the *vocabulary* of Apple IIc/Apple IIe assembly language—the 6502B/65C02 instruction set.

The chapter was not designed to be read in one sitting; it's a reference chapter, so you should probably browse through it to get an idea of what kind of material the chapter contains. Then you can refer to it when necessary.

Most of this chapter is devoted to an alphabetical listing of the 6502B/65C02 instruction set. The listing includes all 56 assembly-language instructions that are used by the 6502B microprocessor, plus eight additional instructions that can be used with the 65C02 processor. Each of these 65C02-specific instructions is marked with an asterisk. All of the addressing modes used by the 6502B and the 65C02 are also listed, and those that are applicable only to the 65C02 chip are identified.

Instructions for the 65802 and the 65816—the two 16-bit chips that are now members of the 6502 family—are not included in the listing in this chapter. However, a complete listing of the 65802/65816 instruction set can be found in the Appendices.

Under each mnemonic listed, you'll find a brief explanation of how that mnemonic is used in Apple IIc and Apple IIe assembly-language programs. In addition, the flags and registers affected by each mnemonic are listed.

At the end of this chapter, following the alphabetical listing of the 6502B/65C02 instruction set, you'll find a special bonus: a BASIC program called "The Byte Simulator" that will help you understand how your computer's microprocessor processes the instruction set. The program is written in BASIC, so you can type and execute it without using an assembler. When you run it, you'll see how it got its name.

Abbreviations Used in This Chapter

The following section contains a complete listing of the 6502B/65C02 microprocessor instruction set and includes all of the instruction mnemonics used in Apple IIc and Apple IIe assembly-language programming. Of course, the list does not include pseudo-operations (also called pseudo-ops or directives), which vary from assembler to assembler. It does include addressing modes, which will be covered in Chapter 7. For a listing of the pseudo-ops used by your assembler, you should consult your assembler's instruction manual.

Tables 6-1 through 6-3 define the abbreviations used in the instruction set.

Table 6-1. Processor Status (P) Register Flags

N	Negative (sign) flag
V	Overflow flag
B	Break flag
D	Decimal flag
I	Interrupt flag
Z	Zero flag
C	Carry flag

Table 6-2. 6502B/65C02 Memory Registers

A	Accumulator
X	X register
Y	Y register
M	Memory register

Table 6-3. 6502B/65C02 Addressing Modes

A	Absolute addressing
AC	Accumulator addressing
AI	Absolute indexed indirect addressing (JMP instruction, 65C02 only)
Z	Zero-page addressing (65C02 only)
IMM	Immediate addressing
IND	Absolute indirect addressing
IMP	Implied addressing
AX	Absolute,X (X-indexed) addressing
AY	Absolute,Y (Y-indexed) addressing
IX	Indexed indirect (Indirect,X) addressing
IY	Indirect indexed (Indirect,Y) addressing
R	Relative addressing
ZPG	Zero-page indirect addressing
ZX	Zero-page X-indexed (Zero-page,X) addressing
ZY	Zero-page Y-indexed (Zero-page,Y) addressing

The Instruction Set

ADC (Add with carry) Adds the contents of the accumulator to the contents of a specified memory location or literal value. If the P register's carry flag is set, a carry is also added. The result of the addition operation is then stored in the accumulator.

Flags affected: N, V, Z, C
Registers affected: A
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX, ZPG*

AND (Logical AND) Performs a binary logical AND operation on the contents of the accumulator and the contents of a specified memory location or an immediate value. The result of the operation is stored in the accumulator.

Flags affected: N, Z
Registers affected: A
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX, ZPG*

ASL (Arithmetic shift left) Moves each bit in the accumulator or a specified memory location one position to the left. A 0 is deposited into bit 0 position, and bit 7 is forced into the carry bit of the P register. The result of the operation is left in the accumulator or the affected memory register.

Flags affected: N, Z, C
Registers affected: A, M
Addressing modes: AC, A, Z, AX, ZX

BCC (Branch if carry clear) Executes a branch if the carry flag of the P register is clear. Results in no operation if the carry flag is set. The destination of the branch is calculated by adding a signed displacement, ranging from −128 to +127, to the address of the first instruction that follows the BCC instruction. This calculation results in an effective displacement of +129 bytes to −126 bytes from the branch instruction. An attempt to use a BCC instruction for a longer branch will result in an "out of range" error.

Flags affected: None
Registers affected: None
Addressing modes: R

BCS (Branch if carry set) Executes a branch if the carry flag of the P register is set. Results in no operation if the carry flag is clear. The destination of the branch is calculated by adding a signed displacement, ranging from −128 to +127, to the address of the first instruction that follows the BCS instruction. This results in an effective displacement of +129 bytes to −126 bytes from the branch instruction. An attempt to use

a BCS instruction for a longer branch will result in an "out of range" error.

Flags affected: None
Registers affected: None
Addressing modes: R

BEQ (Branch if equal) Executes a branch if the 0 flag of the P register is set. Results in no operation if the 0 flag is clear. Can be used to jump to cause a branch if the result of a calculation is 0 or if two numbers are equal. The destination of the branch is calculated by adding a signed displacement, ranging from −128 to +127, to the address of the first instruction that follows the BEQ instruction. This calculation results in an effective displacement of +129 bytes to −126 bytes from the branch instruction. An attempt to use a BEQ instruction for a longer branch will result in an "out of range" error.

Flags affected: None
Registers affected: None
Addressing modes: R

BIT (Compare bits in accumulator with bits in a specified memory register) Performs a binary logical AND operation on the contents of the accumulator and the contents of a specified memory address. The contents of the accumulator are not affected, but three flags in the P register are.

If any bits that are set in the accumulator match any bits that are set in the value being tested, the Z flag is cleared. If there are no set bits in the accumulator that match any set bits in the value being tested, the Z flag is set. Therefore, by using a BIT instruction followed by a BNE or BEQ instruction, you can determine whether any set bits in the accumulator and the tested value match. If there is a match, a BIT/BNE test will succeed. If there is no match, a BIT/BEQ test will succeed.

The BIT instruction also has another effect: bits 6 and 7 of the value in memory being tested are transferred directly into the V and N bits of the status register. This feature provides a handy method for testing bit 6 and bit 7 of any desired value in a single operation, and the BIT instruction is therefore used frequently in arithmetic operations involving signed numbers.

Flags affected: N, V, Z
Registers affected: None

Addressing modes: A, Z, IMM*, AX*, ZX*

BMI (Branch on minus) Executes a branch if the N flag of the P register is set. Results in no operation if the N flag is clear. The destination of the branch is calculated by adding a signed displacement, ranging from −128 to +127, to the address of the first instruction that follows the BMI instruction. This results in an effective displacement of +129 bytes to −126 bytes from the branch instruction. An attempt to use a BMI instruction for a longer branch will result in an "out of range" error.

Flags affected: None
Registers affected: None
Addressing modes: R

BNE (Branch if not equal) Executes a branch if the 0 flag of the P register is clear (that is, if the result of an operation is non-0). Results in no operation if the 0 flag is set. Can be used to jump to cause a branch if the result of a calculation is not 0 or if two numbers are not equal. The destination of the branch is calculated by adding a signed displacement, ranging from −128 to +127, to the address of the first instruction that follows the BNE instruction. This calculation results in an effective displacement of +129 bytes to −126 bytes from the branch instruction. An attempt to use a BNE instruction for a longer branch will result in an "out of range" error.

Flags affected: None
Registers affected: None
Addressing modes: R

BPL (Branch on plus) Executes a branch if the N flag is clear (that is, if the result of a calculation is positive). Results in no operation if the N flag is set. The destination of the branch is calculated by adding a signed displacement, ranging from −128 to +127, to the address of the first instruction that follows the BPL instruction. This results in an effective displacement of +129 bytes to −126 bytes from the branch instruction. An attempt to use a BPL instruction for a longer branch will result in an "out of range" error.

Flags affected: None
Registers affected: None
Addressing modes: R

*** BRA (Branch always)** Executes a branch to a specified memory address. The destination of the branch is calculated by adding a signed displacement, ranging from −128 to +127, to the address of the first instruction that follows the BRA instruction. This calculation results in an effective displacement of +129 bytes to −126 bytes from the branch instruction. An attempt to use a BRA instruction for a longer branch will result in an "out of range" error.

Flags affected: None
Registers affected: None
Addressing modes: R

BRK (Break) The BRK instruction is a software-controlled interrupt that is often used to halt a program at a desired spot during debugging operations. When a BRK instruction is encountered in a program, the program counter is incremented by two and the break (B) flag of the P register is set. Next, the program counter is pushed onto the stack, high byte first. Then the contents of the P register, with the B flag set, are also pushed onto the stack.

When these operations are done, the interrupt flag (I) is set, disabling interrupts. Then a 16-bit address stored in a special 16-bit pointer is placed in the program counter. In the Apple IIc and the Apple IIe, this pointer is contained in memory addresses $3F0 and $3F1, a pair of registers designed to be used only with the BRK instruction. In older 6502-based computers, the pointer is contained in memory addresses $FFFE and $FFFF—the same addresses that are used as a vector by other types of interrupt instructions.

When a BRK instruction is issued, the BRK pointers used by the Apple IIc and the Apple IIe will usually halt whatever program is being executed and will pass control of the computer to its built-in machine-language monitor.

When control of the Apple IIe/IIc has been returned to the monitor, the contents of memory registers and microprocessor registers can be examined, and debugging of the program being executed can proceed. However, because the BRK instruction causes such a complex series of operations to take place, use of the instruction can sometimes have unforeseen results. To prevent unpleasant surprises from taking place when the BRK instruction is invoked, the Apple IIc is equipped with a built-in interrupt handler that can handle BRK instructions in various ways, depending upon whether the computer is in 80-column mode and whether auxiliary memory and bank-switched memory are being used.

By changing the contents of the BRK vector at memory addresses $3F0 and $3F1, Apple IIc owners can write their own BRK-handling routines if they wish. But for most debugging operations, there is probably no reason to try to override the BRK-handler that is built into the Apple IIc.

Flags affected: B
Registers affected: Stack pointer
Addressing modes: IMP

BVC (Branch if overflow clear) Executes a branch if the P register's overflow (V) flag is clear. Results in no operation if the overflow flag is set. The destination of the branch is calculated by adding a signed displacement, ranging from −128 to +127, to the address of the first instruction that follows the BVC instruction. This calculation results in an effective displacement of +129 bytes to −126 bytes from the branch instruction. An attempt to use a BVC instruction for a longer branch will result in an "out of range" error. This instruction is used primarily in operations involving signed numbers.

Flags affected: None
Registers affected: None
Addressing modes: R

BVS (Branch if overflow set) Executes a branch if the P register's overflow (V) flag is set. Results in no operation if the overflow flag is clear. The destination of the branch is calculated by adding a signed displacement, ranging from −128 to +127, to the address of the first instruction that follows the BVC instruction. This calculation results in an effective displacement of +129 bytes to −126 bytes from the branch instruction. An attempt to use a BVC instruction for a longer branch will result in an "out of range" error. This instruction is used primarily in operations involving signed numbers. However, it also provides an easy way to test bit 6 of any value.

Flags affected: None
Registers affected: None
Addressing modes: R

CLC (Clear carry) Clears the carry bit of the processor status register.

Flags affected: C
Registers affected: None
Addressing modes: IMP

CLD (Clear decimal flag) Clears the decimal flag of the P register, putting the Apple IIc/IIe into binary mode (its default mode) rather than into an alternate mode called BCD (or decimal mode). When the Apple IIc is in binary mode, it can carry out ordinary binary operations on ordinary binary numbers. When the computer is in BCD mode, it is capable of working with a special kind of numbers, called BCD numbers, which are more accurate than ordinary binary numbers but are more difficult to use. BCD numbers (covered in more detail in Chapters 2 and 10) are often used in business-related programs in which a high degree of accuracy in arithmetic is important. In most other kinds of 6502B/65C02 programs, the decimal flag is usually left clear and ordinary binary numbers are generally used.

Flags affected: D
Registers affected: None
Addressing modes: IMP

CLI (Clear interrupt mask) Clears the interrupt flag of the P register, enabling interrupts to take place. Until the advent of the Apple IIe, interrupts were not supported by Apple II-series computers and therefore were not used in Apple programs. However, the Apple IIe does support interrupts, and the Apple IIc even has an extremely sophisticated built-in interrupt handler. Interrupts are a bit beyond the scope of this book, but they are becoming increasingly important, particularly in programs designed to be used with sophisticated peripherals such as the Apple mouse. Details on how interrupts are used in Apple IIc/IIe programs can be found in the Apple IIc and Apple IIe reference manuals and in manuals covering the operations of peripherals such as the Apple mouse.

Flags affected: I
Registers affected: None
Addressing modes: IMP

CLV (Clear overflow flag) Clears the P register's overflow flag by setting it to 0. This instruction is used primarily in operations involving signed numbers. However, it can also be used to clear bit 6 of any value.

Flags affected: V
Registers affected: None
Addressing modes: IMP

CMP (Compare with accumulator) Compares a specified literal number, or the contents of a specified memory location, with the contents of the accumulator. The N, Z, and C flags of the status register are affected by this operation, and a branch instruction usually follows it. The branch instruction that follows the CMP instruction can cause a program to branch to a given routine under certain conditions. For example, a branch might take place if the value in the accumulator is less than, equal to, or more than the value being tested. The CMP instruction, and the branching instructions that are used with it, are covered in more detail in Chapter 8.

Flags affected: N, Z, C
Registers affected: None
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX, ZPG*

CPX (Compare with X register) Compares a specified literal number, or the contents of a specified memory location, with the contents of the X register. The N, Z, and C flags of the status register are affected by this operation, and a branch instruction usually follows. The branch instruction that follows the CPX instruction can, under certain conditions, cause a program to branch to a given routine. For example, a branch might take place if the value in the X register is less than, equal to, or more than the value being tested.

Flags affected: N, Z, C
Registers affected: None
Addressing modes: A, IMM, Z

CPY (Compare with Y register) Compares a specified literal number, or the contents of a specified memory location, with the contents of the Y register. The N, Z, and C flags of the status register are affected by this operation, and a branch instruction usually follows. The branch instruction that follows the CPY instruction can cause a program to branch to a given routine under certain conditions; for example, a branch might take place if the value in the Y register is less than, equal to, or more than the value being tested.

Flags affected: N, Z, C

Registers affected: None
Addressing modes: A, IMM, Z

DEA or DEC A (Decrement accumulator) Decrements the contents of the accumulator by one. If the value of the accumulator is $00, the result of a DEA operation will be $FF, since there is no carry.

Flags affected: N, Z
Registers affected: A
Addressing modes: AC

DEC (Decrement a memory location) Decrements the contents of a specified memory location by one. If the value in the location is $00, the result of a DEC operation will be $FF, since there is no carry.

Flags affected: N, Z
Registers affected: M
Addressing modes: ACC*, A, Z, AX, ZX

DEX (Decrement X register) Decrements the X register by one. If the value in the location is $00, the result of the DEX operation will be $FF, since there is no carry.

Flags affected: N, Z
Registers affected: X
Addressing modes: IMP

DEY (Decrement Y register) Decrements the Y register by one. If the value in the location is $00, the result of the DEY operation will be $FF, since there is no carry.

Flags affected: N, Z
Registers affected: Y
Addressing modes: IMP

EOR (Exclusive-OR with accumulator) Performs an Exclusive-OR operation on the contents of the accumulator and a specified literal value or memory location. The N and Z flags are conditioned in accordance with the result of the operation, and the result is stored in the accumulator.

Flags affected: N, Z
Registers affected: A

Addressing modes: A, Z, I, AX, AY, IX, IY, ZX, ZPG*

INA or INC A (Increment accumulator) The content of the accumulator is incremented by one. If the content of the accumulator is $FF, the result of the INA operation will be $00, since there is no carry.

Flags affected: N, Z
Registers affected: A
Addressing modes: AC

INC (Increment memory) The contents of a specified memory location are incremented by one. If the value in the location is $FF, the result of the INC operation will be $00, since there is no carry.

Flags affected: N, Z
Registers affected: M
Addressing modes: ACC*, A, Z, AX, ZX

INX (Increment X register) The contents of the X register are incremented by one. If the value of the X register is $FF, the result of the INX operation will be $00, since there is no carry.

Flags affected: N, Z
Registers affected: X
Addressing modes: IMP

INY (Increment Y register) The contents of the Y register are incremented by one. If the value of the Y register is $FF, the result of the INY operation will be $00, since there is no carry.

Flags affected: N, Z
Registers affected: Y
Addressing modes: IMP

JMP (Jump to address) Causes program execution to jump to a specified address.

Flags affected: None
Registers affected: None
Addressing modes: A, IX*, IND

JSR (Jump to subroutine) Causes program execution to jump to the address that follows the instruction. That address should be the starting

address of a subroutine that ends with the instruction RTS. When the program reaches that RTS instruction, execution of the program returns to the next instruction after the JSR instruction that caused the jump to the subroutine.

Flags affected: None
Registers affected: Stack pointer
Addressing modes: A

LDA (Load the accumulator) Loads the accumulator with either a specified value or the contents of a specified memory location. The N flag is conditioned if a value with the high bit set is loaded into the accumulator, and the Z flag is set if the value loaded into the accumulator is 0.

Flags affected: N, Z
Registers affected: A
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX, ZPG*

LDX (Load the X register) Loads the X register with either a specified value or the contents of a specified memory location. The N flag is conditioned if a value with the high bit set is loaded into the X register, and the Z flag is set if the value loaded into the X register is 0.

Flags affected: N, Z
Registers affected: X
Addressing modes: A, Z, IMM, AY, ZY

LDY (Load the Y register) Loads the Y register with either a specified value or the contents of a specified memory location. The N flag is conditioned if a value with the high bit set is loaded into the Y register, and if the Z flag is set, the value loaded into the Y register is 0.

Flags affected: N, Z
Registers affected: Y
Addressing modes: A, Z, IMM, AX, ZX

LSR (Logical shift right) Each bit in the accumulator is moved one position to the right. A 0 is deposited into the bit 7 position, and bit 0 is deposited into the carry. The result is left in the accumulator or in the affected memory register.

Flags affected: N, Z, C

Registers affected: A, M
Addressing modes: AC, A, Z, AX, ZX

NOP (No operation) Causes the computer to do nothing for two clock cycles. Used in delay loops and to synchronize the timing of computer operations.

Flags affected: None
Registers affected: None
Addressing modes: IMP

ORA (Inclusive-OR with the accumulator) Performs a binary inclusive-OR operation on the value in the accumulator and a literal value or the contents of a specified memory location. The N and Z flags are conditioned in accordance with the result of the operation, and the result of the operation is deposited in the accumulator.

Flags affected: N, Z
Registers affected: A, M
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX, ZPG*

PHA (Push accumulator) The contents of the accumulator are pushed on the stack. The accmulator and the P register are left unchanged.

Flags affected: None
Registers affected: Stack pointer
Addressing modes: IMP

PHP (Push processor status) The contents of the P register are pushed on the stack. The P register itself is left unchanged and no other registers are affected.

Flags affected: None
Registers affected: Stack pointer
Addressing modes: IMP

*** PHX (Push X register on the stack)** The contents of the X register are pushed on the stack. The X register and the P register are left unchanged.

Flags affected: None
Registers affected: Stack pointer
Addressing modes: IMP

*** PHY (Push Y register on the stack)** The contents of the Y register are pushed on the stack. The Y register and the P register are left unchanged.

Flags affected: None
Registers affected: Stack pointer
Addressing modes: IMP

PLA (Pull accumulator) One byte is removed from the stack and deposited in the accumulator. The N and Z flags are conditioned, just as if an LDA operation had been carried out.

Flags affected: N, Z
Registers affected: A, Stack pointer
Addressing modes: IMP

PLP (Pull processor status) One byte is removed from the stack and deposited in the P register. This instruction is used to retrieve the status of the P register after it has been saved by pushing it onto the stack. All of the flags are thus conditioned to reflect the original status of the P register.

Flags affected: N, V, B, D, I, Z, C
Registers affected: Stack pointer
Addressing modes: IMP

*** PLX (Pull X register)** One byte is removed from the stack and deposited in the X register. The N and Z flags are conditioned, just as if an LDX operation had been carried out.

Flags affected: N, Z
Registers affected: X, Stack pointer
Addressing modes: IMP

*** PLY (Pull Y register)** One byte is removed from the stack and deposited in the Y register. The N and Z flags are conditioned, just as if an LDY operation had been carried out.

Flags affected: N, Z
Registers affected: Y, Stack pointer
Addressing modes: IMP

ROL (Rotate left) Each bit in the accumulator or a specified memory location is moved one position to the left. The carry bit is deposited into the bit 0 location and is replaced by bit 7 of the accumulator or the affected memory register. The N and Z flags are conditioned in accordance with the result of the rotation operation.

Flags affected: N, Z, C
Registers affected: A, M
Addressing modes: AC, A, Z, AX, ZX

ROR (Rotate right) Each bit in the accumulator or a specified memory location is moved one position to the right. The carry bit is deposited into the bit 7 location and is replaced by bit 0 of the accumulator or the affected memory register. The N and Z flags are conditioned in accordance with the result of the rotation operation.

Flags affected: N, Z, C
Registers affected: A, M
Addressing modes: AC, A, Z, AX, ZX

RTI (Return from interrupt) The RTI instruction is used to end an interrupt in much the same way that an RTS instruction is used to end a subroutine. When an RTI instruction is issued, the program counter and the P register are pulled from the stack, and the stack pointer is adjusted to reflect the new state of the stack. Then the interrupt ends, and the program jumps back to where it left off before the interrupt began (that is, to the instruction following the instruction that began the interrupt). When using the RTI instruction, it is important to remember that although RTI restores the P register to its original state, RTI does not restore the original states of the X, Y, and A registers. That job is left to the programmer.

Flags affected: N, V, B, D, I, Z, C
Registers affected: PC, Stack pointer
Addressing modes: IMP

RTS (Return from subroutine) The RTS instruction serves two functions. When used at the end of a machine-language program, it terminates the program and passes control of the Apple IIc/IIe to whatever system program was running when the machine-language program began; usually, this results in a return either to BASIC or to the Apple IIc/IIe monitor.

When RTS is used at the end of a subroutine, it has a completely different function; it pulls a 16-byte address from the top of the stack and loads that address into the 6502B/65C02 program counter. This operation ends the subroutine, and the program in progress then jumps back to where it was before the subroutine began (that is, to the instruction following the instruction that called the subroutine).

Flags affected: None
Registers affected: Stack pointer
Addressing modes: IMP

SBC (Subtract with carry) Subtracts a literal value or the contents of a specified memory location from the contents of the accumulator. The opposite of the carry is also subtracted—in other words, there is a borrow. The N, V, Z, and C flags are all conditioned by this operation, and the result of the operation is deposited in the accumulator.

Flags affected: N, V, Z, C
Registers affected: A
Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX, ZPG*

SEC (Set carry) The carry flag is set. This instruction usually precedes an SBC instruction. Its primary purpose is to set the carry flag so that there can be a borrow.

Flags affected: C
Registers affected: None
Addressing modes: IMP

SED (Set decimal mode) Prepares the computer for operations using BCD (binary coded decimal) numbers. BCD arithmetic is more accurate than binary arithmetic (the usual type of 6502B/65C02 arithmetic) but is slower and more difficult to use and consumes more memory. BCD arithmetic is usually used in accounting and bookkeeping programs and in floating-point arithmetic operations.

Flags affected: D
Registers affected: None
Addressing modes: IMP

SEI (Set interrupt disable) Sets the interrupt (I) flag of the P register, disabling all maskable interrupts (IRQs). Setting the interrupt flag does

not disable non-maskable interrupts (NMIs), which are essential to the operation of the Apple IIc/Apple IIe.

Flags affected: I
Registers affected: None
Addressing modes: IMP

STA (Store accumulator) Stores the contents of the accumulator in a specified memory location. The contents of the accumulator are not affected.

Flags affected: None
Registers affected: M
Addressing modes: A, Z, AX, AY, IX, IY, ZX, ZPG*

STX (Store X register) Stores the contents of the X register in a specified memory location. The contents of the X register are not affected.

Flags affected: None
Registers affected: M
Addressing modes: A, Z, ZY

STY (Store Y register) Stores the contents of the Y register in a specified memory location. The contents of the Y register are not affected.

Flags affected: None
Registers affected: M
Addressing modes: A, Z, ZX

*** STZ (Store 0 in memory)** Stores a 0 in a specified memory location.

Flags affected: None
Registers affected: None
Addressing modes: A, Z, AX, ZX

TAX (Transfer accumulator to X register) The value in the accumulator is deposited in the X register. The N and Z flags are conditioned in accordance with the result of this operation. The contents of the accumulator are not changed.

Flags affected: N, Z
Registers affected: X
Addressing modes: IMP

TAY (Transfer accumulator to Y register) The value in the accumulator is deposited in the Y register. The N and Z flags are conditioned in accordance with the result of this operation. The contents of the accumulator are not changed.

Flags affected: N, Z
Registers affected: Y
Addressing modes: IMP

*** TRB (Test and reset bits)** A logical AND operation is performed on a memory location and the inverse of the accumulator value. The result is stored in the memory location; the Z flag is conditioned.

Flags affected: D, Z, C, N
Registers affected: M
Addressing modes: A, Z

*** TSB (Test and set bits)** A logical OR operation is performed on a memory location and the value of the accumulator. The result is stored in the memory location and the Z flag is conditioned.

Flags affected: D, Z, C, N
Registers affected: M
Addressing modes: A, Z

TSX (Transfer stack to X register) The value in the stack pointer is deposited in the X register. The N and Z flags are conditioned in accordance with the result of this operation. The value of the stack pointer is not changed.

Flags affected: N, Z
Registers affected: X
Addressing modes: IMP

TXA (Transfer X register to accumulator) The value in the X register is deposited in the accumulator. The N and Z flags are conditioned in accordance with the result of this operation. The value of the X register is not changed.

Flags affected: N, Z
Registers affected: A
Addressing modes: IMP

TXS (Transfer X register to stack) The value in the X register is deposited in the stack pointer. No flags are conditioned by this operation. The value of the X register is not changed.

Flags affected: None
Registers affected: Stack pointer
Addressing modes: IMP

TYA (Transfer Y register to accumulator) The value in the Y register is deposited in the accumulator. The N and Z flags are conditioned by this operation. The value of the Y register is not changed.

Flags affected: N, Z
Registers affected: A
Addressing modes: IMP

The Byte Simulator

Program 6-1 is a printout of "The Byte Simulator," a program that will give you an inside look at what your computer's microprocessor does when it processes the above instructions.

The Byte Simulator is not a real assembler, so you won't be able to assemble and run programs with it. It has some other limitations, too; for example, it doesn't allow the use of labels or indirect addressing (a topic that will be covered in detail in a later chapter). And, since it was designed to be compatible with all Apple IIe computers as well as the Apple IIc, it won't accept the eight instructions that are unique to the 65C02 microprocessor.

One important feature of The Byte Simulator is that it can't freeze your computer system, making everything you're doing come to a crashing halt. The reason is that The Byte Simulator will not actually poke values into your computer's memory registers; it can *read* the contents of any memory register in your computer, but it can't *write* to registers in RAM. So, although you can use the program to check the contents of any memory register in your computer, you won't be courting disaster every time you key in an assembly-language instruction.

After you've typed and saved The Byte Simulator, you can see how it works by using it to type any of the programs presented so far in this book. Once you know how it works, you can use it to test any Apple IIe or Apple IIc assembly-language program.

When you load the program, you will first see a status line across the top of your screen. This line will show you the current status of the four most important registers inside the 65C02/6502B chip: the processor status register, the accumulator, and the X and Y registers. Beneath this status line, you'll see a cursor. Starting at the cursor location, you can type an assembly-language program and see exactly how each instruction in the program would affect each register in your Apple's main microprocessor if the program were actually running. Each time you type a line of assembly language using The Byte Simulator, it will show you how that line would affect your CPU's internal registers and keep a running update of the contents of your CPU's A, X and Y registers. The information will be displayed in both hex and binary notation.

As you continue to learn 6502B/65C02 assembly language, The Byte Simulator should be very helpful. Every instruction in 6502 assembly language has some effect on the registers inside your Apple's CPU, and you can use The Byte Simulator to get an inside look at those effects.

Program 6-1
THE BYTE SIMULATOR

```
100  REM  ******************** THE BYTE SIMULATOR **********
     **********
110  DATA   ASL,BRK,CLC,CLD,CLI,CLV,DEX,DEY,INX,INY,LSR,NOP,
     PHA,PHP,PLA
120  DATA  PLP,ROL,ROR,RTI,RTS,SEC,SED,SEI,TAX,TAY,TSX,TXA,
     TXS,TYA
130  DATA  ADC,AND,CMP,CPX,CPY,EOR,LDA,LDX,LDY,ORA,SBC
140  DATA  ADC,AND,ASL,BCC,BCS,BEQ,BIT,BMI,BNE,BPL,BVC,BVS,
     CMP,CPX,CPY
150  DATA  DEC,EOR,INC,JMP,JSR,LDA,LDX,LDY,LSR,ORA,ROL,ROR,
     SBC,STA,STX
160  DATA  STY,ASL,LSR,ROL,ROR
170  DATA  0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
180  DATA  0000,0001,0010,0011,0100,0101,0110,0111
190  DATA  1000,1001,1010,1011,1100,1101,1110,1111
200  DIM HEX$(8),BIT$(8),H$(16),B$(16),TEMP$(2),BIT(8),N$(75)
210 AH$ = "00":XH$ = "00":YH$ = "00":Z$ = "0":AD = 0:SC =
     0:YD = 0
220 AB$ = "00000000":XB$ = "00000000":YB$ = "00000000"
230 N = 0:V = 0:B = 0:D = 0:I = 0:Z = 0:C = 0
240  IF  RIGHT$ (A$,1) = " " THEN 300
250  FOR L = 1 TO 75: READ N$(L): NEXT L
260  FOR L = 1 TO 16: READ H$(L): NEXT L: FOR L = 1 TO 16:
     READ B$(L): NEXT L
270  PRINT  CHR$ (4);"PR#3": TEXT : REM   CLEAR SCREEN, START
     PROGRAM
280  PRINT "NV-BDIZC   A: ";AH$;"      X: ";XH$;"      Y: "
     ;YH$
290  PRINT N;V;"-";B;D;I;Z;C;"   ";AB$;"   ";XB$;"   ";YB$:
     PRINT : GOTO 310
300  PRINT  CHR$ (7): REM  RING BELL
```

```
310 B = 0:A$ = "": INPUT " ";A$: REM  TYPE SPACE BETWEEN
    LAST PAIR OF QUOTES
320  IF  LEN (A$) < 3 OR  LEN (A$) > 10 THEN 300
330  IF  LEN (A$) = 3 THEN 420: REM  GOTO ONE-BYTE MNEMONIC
    ROUTINES
340  IF  MID$ (A$,4,1) <  >  CHR$ (32) THEN 300
350  IF  RIGHT$ (A$,1) =  CHR$ (32) THEN 300
360  GOTO 450: REM    GO TO MULTIPLE-BYTE MNEMONIC ROUTINES
370  REM    *** ROUTINE TO CONVERT OP$ & AD$ TO BINARY
    NUMBERS ******
380 OD$ = OP$: GOSUB 1130: FOR L = 1 TO 8:BIT$(L) = Z$:
    NEXT L
390  FOR L = 1 TO 8:B1$(L) =  MID$ (OB$,L,1): NEXT L
400  FOR L = 1 TO 8:B2$(L) =  MID$ (AB$,L,1): NEXT L: RETURN
410  REM  ************** IMPLIED ADDRESSING *****************
    *********
420  FOR L = 1 TO 29: IF A$ = N$(L) THEN OC$ = A$:OC = L:
    GOTO 1350
430  NEXT L: GOTO 300
440  REM  **** IMMEDIATE, ABSOLUTE & ACCUMULATOR ADDRESSING
    *********
450 OC$ =  LEFT$ (A$,3):OP$ =  MID$ (A$,5)
460  IF  LEFT$ (OP$,1) = "#" AND  MID$ (OP$,2,1) = "$" THEN
    FLAG$ = "AH": GOTO 530
470  IF  LEFT$ (OP$,1) = "#" THEN FLAG$ = "AD": GOTO 620
480  IF  LEFT$ (OP$,1) = "$" THEN FLAG$ = "IH": GOTO 690
490  IF OP$ = "A" THEN 790
500  IF  LEFT$ (OP$,1) < "0" AND  LEFT$ (OP$,1) > "9" THEN
    300: REM   TRY AGAIN
510 FLAG$ = "ID": GOTO 840
520  REM  ********* HEX OPERAND, ABSOLUTE ADDRESSING ******
    **************
530 OP$ =  MID$ (OP$,3):BA$ = "H": REM  HEX ADDRESS
540  FOR L = 30 TO 40: IF OC$ = N$(L) THEN OC = L: GOTO 560
550  NEXT L: GOTO 300
560  IF  LEN (OP$) > 2 THEN 300
570  FOR L = 1 TO  LEN (OP$):X$ =  MID$ (OP$,L,1): IF X$ <
    Z$ OR X$ > "F" THEN 300
580  IF X$ > "9" AND X$ < "A" THEN 300
590  IF  LEN (OP$) = 1 THEN OP$ = Z$ + OP$
600 OH$ = OP$: GOSUB 1030:OP$ = OD$: GOTO 670
610  REM  ********* DECIMAL OPERAND, ABSOLUTE ADDRESSING **
    ************
620 OP$ =  MID$ (OP$,2):BA$ = "D": REM   DECIMAL ADDRESS
630  FOR L = 30 TO 40: IF OC$ = N$(L) THEN OC = L: GOTO 650
640  NEXT L: GOTO 300
650  IF  VAL (OP$) > 255 THEN 300
660  FOR L = 1 TO  LEN (OP$):X$ =  MID$ (OP$,L,1): IF  ASC
    (X$) < 48 OR  ASC
(X$) > 57 THEN 420
670 OC = OC - 29: GOTO 1420
680  REM  ********* HEX OPERAND, IMMEDIATE ADDRESSING *****
    ***********
690 OP$ =  MID$ (OP$,2):BA$ = "H": REM  HEX ADDRESS
700  FOR L = 41 TO 71: IF OC$ = N$(L) THEN OC = L: GOTO 720
710  NEXT L: GOTO 300
720  IF  LEN (OP$) > 4 THEN 300
```

```
730  FOR L = 1 TO  LEN (OP$):X$ =  MID$ (OP$,L,1): IF X$ <
     Z$ OR X$ > "F" THEN 300
740  IF X$ > "9" AND X$ < "A" THEN 300
750  NEXT L
760 OH$ = OP$: GOSUB 1030:OP$ = OD$:OP$ =  STR$ ( PEEK ( VAL
     (OP$)))
770 BA$ = "D":OC = OC - 40: GOTO 1430
780  REM   *************** ACCUMULATOR ADDRESSING *********
     ************
790 OP$ = "A": REM   ACCUMULATOR ADDRESSING
800  FOR L = 72 TO 75: IF OC$ = N$(L) THEN OC = L - 71: GOTO
     820
810  NEXT L: GOTO 300
820  ON OC GOTO 1510,1760,1790,1810
830  REM  *********** DECIMAL OPERAND, IMMEDIATE ADDRESSING
     **********
840 BA$ = "D": REM   DECIMAL ADDRESS
850  IF  VAL (OP$) > 65535 THEN 300
860  FOR L = 1 TO  LEN (OP$):X$ =  MID$ (OP$,L,1): IF  ASC
     (X$) < 48 OR  ASC
(X$) > 57 THEN 300
870  FOR L = 41 TO 71: IF OC$ = N$(L) THEN OC = L: GOTO 890
880  NEXT L: GOTO 300
890 OP$ =  STR$ ( PEEK ( VAL (OP$)))
900 OC = OC - 40: GOTO 1430
910  REM  ************ DECIMAL-TO-HEXADECIMAL CONVERSION **
     ************
920  FOR L = 1 TO 4:HEX$(L) = "": NEXT L
930  FOR L = 1 TO 5:T$ =  RIGHT$ (OD$,L): NEXT L
940 NR =  VAL (OD$):X = 4
950 TMP = NR:NR =  INT (NR / 16):TMP = TMP - NR * 16
960  IF TMP < 10 THEN HEX$(X) =  RIGHT$ ( STR$ (TMP),1): GOTO 980
970 HEX$(X) =  CHR$ (TMP - 10 +  ASC ("A"))
980  IF NR <  > 0 THEN X = X - 1: GOTO 950
990 OH$ = HEX$(1) + HEX$(2) + HEX$(3) + HEX$(4)
1000  IF  LEN (OH$) = 1 THEN OH$ = Z$ + OH$
1010  RETURN
1020  REM  ********** HEXADECIMAL-TO-DECIMAL CONVERSION ***
     **********
1030 NR = 0: FOR L = 1 TO  LEN (OH$):HEX$(L) =  MID$
     (OH$,L,1)
1040  IF HEX$(L) <  = "9" THEN NR = NR * 16 +  VAL (HEX$(L)):
     GOTO 1060
1050 NR = NR * 16 +  ASC (HEX$(L)) -  ASC ("A") + 10
1060  NEXT L:OD$ =  STR$ (NR): RETURN
1070  REM  ************** BINARY-TO-DECIMAL CONVERSION ****
     **********
1080  FOR L = 8 TO 1 STEP  - 1:B$(L) =  MID$ (OB$,L,1):
     NEXT L
1090  FOR L = 1 TO 8:BIT(L) =  VAL (B$(L)): NEXT L:OD =
     0:M = 256
1100  FOR L = 1 TO 8:M = M / 2:OD = OD + BIT(L) * M: NEXT L
1110 OD$ =  STR$ (OD): RETURN
1120  REM  ********** DECIMAL-TO-BINARY CONVERSION ********
     **************
1130 OD =  VAL (OP$): FOR L = 8 TO 1 STEP  - 1:Q = OD / 2:R
     = Q -  INT (Q)
```

```
1140  IF R = 0 THEN BT$(L) = Z$: GOTO 1160
1150 BT$(L) = "1"
1160 OD =  INT (Q): NEXT L
1170 OB$ = BT$(1) + BT$(2) + BT$(3) + BT$(4) + BT$(5) +
     BT$(6) + BT$(7) + BT$(8): RETURN
1180  REM  ******** HEXADECIMAL-TO-BINARY CONVERSION ******
     **************
1190 HEX$(1) = "":HEX$(2) = "": FOR L = 1 TO  LEN (OH$):
     HEX$(L) =  MID$ (OH$,L,1)
1200  NEXT L: IF HEX$(2) = "" THEN HEX$(2) = HEX$(1):HEX$(1)
     = "0"
1210  FOR L = 1 TO 16: IF HEX$(1) = H$(L) THEN BIT$(1) =
     B$(L)
1220  NEXT L
1230  FOR L = 1 TO 16: IF HEX$(2) = H$(L) THEN BIT$(2) =
     B$(L)
1240  NEXT L
1250 OB$ = BIT$(1) + BIT$(2): PRINT : RETURN
1260  REM  ************ BINARY TO HEXADECIMAL CONVERSION **
     *************
1270  FOR L = 1 TO 8:BIT$(L) =  MID$ (OB$,L,1): NEXT L
1280 BIT$ = BIT$(1) + BIT$(2) + BIT$(3) + BIT$(4) + BIT$(5)
     + BIT$(6) + BIT$(7) + BIT$(8)
1290 T1$ =  LEFT$ (BIT$,4):T2$ =  RIGHT$ (BIT$,4): FOR L =
     1 TO 16
1300  IF T1$ = B$(L) THEN HEX$(1) = H$(L)
1310  NEXT L: FOR L = 1 TO 16: IF T2$ = B$(L) THEN HEX$(2) =
     H$(L)
1320  NEXT L: IF HEX$(1) = "" THEN HEX$(1) = Z$: IF HEX$(2)
     = "" THEN HEX$(2) = Z$
1330 OH$ = HEX$(1) + HEX$(2): RETURN
1340  REM    ****************** ON/GOTO DATA **************
     ************
1350  ON OC GOTO 1510,1560,1570,1580,1590,1600,1610,1650,
     1690,1730
1360 NR = OC - 10
1370  ON NR GOTO 1760,1780,1780,1780,1780,1780,1790,1810,
     1830,1840
1380 NR = NR - 10
1390  ON NR GOTO 1850,1860,1870,1880,1900,1920,1930,1940,
     1950,1840
1400 NR = NR - 10
1410  ON NR GOTO 1850,1860,1870,1880,1900,1920,1930,1940,
     1950,1950
1420  ON OC GOTO 1970,2100,2170,2250,2320,2390,2450,2550,
     2650,2750,2810
1430  ON OC GOTO 1970,2350,3250,1780,1780,1780,3200,1780,
     1780,1780
1440 NR = OC - 10
1450  ON NR GOTO 1780,1780,3250,2250,2320,3260,2390,3290,
     1780,1780
1460 NR = NR - 10
1470  ON NR GOTO 2450,2550,2650,3320,2750,3340,3360,2810,
     1780,1780
1480 NR = NR - 10
1490  ON NR GOTO 1780,3390
1500  REM  *********** OP-CODE ROUTINES START HERE ********
     **********
```

```
1510 C =  VAL ( LEFT$ (AB$,1)):AB$ =  MID$ (AB$,2) + Z$:
     REM  *** ASL ***
1520 OB$ = AB$: GOSUB 1270:AH$ = OH$:OP$ = OH$: GOSUB 1030:N
     = 0:Z = 0
1530  IF  LEFT$ (OB$,1) = "1" THEN N = 1
1540  IF  VAL (OD$) = 0 THEN Z = 1
1550  GOTO 3390
1560 B = 1: GOTO 3390: REM  *** BRK ***
1570 C = 0: GOTO 3390: REM  *** CLCL ***
1580 D = 0: GOTO 3390: REM  *** CLD ***
1590 I = 0: GOTO 3390: REM  *** CLI ***
1600 V = 0: GOTO 3390: REM  *** CLV ***
1610 OH$ = XH$: GOSUB 1030:XD =  VAL (OD$): REM  *** DEX ***
1620 XD = XD - 1: IF XD < 0 THEN XD = 255
1630 OD$ =  STR$ (XD): GOSUB 920:XH$ = OH$: GOSUB 1190:XB$
     = OB$
1640 TMP = XD: GOSUB 3410: GOTO 280
1650 OH$ = YH$: GOSUB 1030:YD =  VAL (OD$): REM  *** DEY ***
1660 YD = YD - 1: IF YD < 0 THEN YD = 255
1670 OD$ =  STR$ (YD): GOSUB 920:YH$ = OH$: GOSUB 1190:YB$
     = OB$
1680 TMP = YD: GOSUB 3410: GOTO 280
1690 OH$ = XH$: GOSUB 1030:XD =  VAL (OD$): REM  *** INX ***
1700 OD$ =  STR$ (XD): GOSUB 920:XH$ = OH$: GOSUB 1190:XB$ =
     OB$
1710 XD = XD + 1: IF XD > 255 THEN XD = 0
1720  GOTO 1630
1730 OH$ = YH$: GOSUB 1030:YD =  VAL (OD$): REM  *** INY ***
1740 YD = YD + 1: IF YD > 255 THEN YD = 0
1750  GOTO 1670
1760 C =  VAL ( RIGHT$ (AB$,1)):AB$ = Z$ +  LEFT$ (AB$,7):
     REM  *** LSR ***
1770  GOTO 1520
1780  GOTO 3390: REM  *** NOP, PHA, PHP, PLA AND PLP ***
1790 TMP = C:C =  VAL ( LEFT$ (AB$,1)):AB$ =  MID$ (AB$,2) +
     STR$ (TMP): REM
*** ROL ***
1800  GOTO 1520
1810 TMP = C:C =  VAL ( RIGHT$ (AB$,1)):AB$ =  STR$ (TMP) +
     LEFT$ (AB$,7):
REM  *** ROR ***
1820  GOTO 1520
1830 N = 0:V = 0:B = 0:D = 0:I = 0:Z = 0:C = 0: GOTO 280:
     REM  *** RTI ***
1840  GOTO 3390: REM  *** RTS ***
1850 C = 1: GOTO 3390: REM  *** SEC ***
1860 D = 1: GOTO 3390: REM  *** SED ***
1870 I = 1: GOTO 3390: REM  *** SEI ***
1880 XH$ = AH$:XB$ = AB$:OP$ = AH$: GOSUB 1030:TMP =  VAL
     (OD$): REM
***              TAX ***
1890  GOSUB 3410: GOTO 3390
1900 YH$ = AH$:YB$ = AB$:OP$ = AH$: GOSUB 1030:TMP =  VAL
     (OD$): REM  *** TAY
***
1910  GOSUB 3410: GOTO 3390
1920 XH$ = "00":XB$ = "00000000": GOSUB 3410: GOTO 3390: REM
     *** TSX ***
```

```
1930 AH$ = XH$:AB$ = XB$: GOTO 1520: REM    *** TXA ***
1940  GOTO 3390: REM  *** TXS ***
1950 AH$ = YH$:AB$ = YB$: GOTO 1520: REM   *** TYA ***
1960  REM  ************** ABSOLUTE-ADDRESS OPERANDS *******
     *******
1970  IF D THEN 2950: REM  *** ADC ***
1980 OP =  VAL (OP$):TMP$ = AB$
1990  GOSUB 1130:PLUS$ = OB$
2000 OH$ = AH$: GOSUB 1030:AD$ = OD$:AD =  VAL (AD$):TMP = AD
2010 AD = AD + OP + C:C = 0: IF AD > 255 THEN  GOSUB 2090
2020 AD$ =  STR$ (AD):OD$ = AD$: GOSUB 920:AH$ = OH$
2030  GOSUB 1190:AB$ = OB$
2040 N = 0: IF AD > 127 THEN N = 1
2050 Z = 0: IF AD = 0 THEN Z = 1
2060 V = 0
2070  IF  LEFT$ (TMP$,1) =  LEFT$ (PLUS$,1) AND  LEFT$
     (TMP$,1) <  >  LEFT$ (AB$,1) THEN V = 1
2080 OD$ = AD$: GOSUB 1030:AH$ = OH$: GOTO 280
2090 C = 1:AD = AD - 256: RETURN
2100  GOSUB 380: REM   *** AND ***
2110  FOR L = 1 TO 8:BIT$(L) = "0": NEXT L
2120  FOR L = 1 TO 8: IF B1$(L) = B2$(L) THEN BIT$(L) =
     B1$(L)
2130  NEXT L
2140 AB$ = BIT$(1) + BIT$(2) + BIT$(3) + BIT$(4) + BIT$(5) +
     BIT$(6) + BIT$(7) + BIT$(8)
2150 OB$ = AB$: GOSUB 1270:AH$ = OH$
2160  GOSUB 1030:TMP =  VAL (OD$): GOSUB 3410: PRINT :
     GOTO 280
2170 OH$ = AH$: GOSUB 1030:AD$ = OD$:AD =  VAL (AD$):OP =
     VAL (OP$): REM  *** CMP ***
2180  IF AD = OP THEN Z = 1: GOTO 2200
2190 Z = 0
2200  IF OP > AD THEN N = 1: GOTO 2220
2210 N = 0
2220  IF AD > OP OR AD = OP THEN C = 1: GOTO 2240
2230 C = 0
2240  GOTO 3390
2250 OH$ = XH$: GOSUB 1030:XD$ = OD$:XD =  VAL (XD$):OP =
     VAL (OP$): REM  CPX
2260  IF XD = OP THEN Z = 1: GOTO 2280
2270 Z = 0
2280  IF OP > XD THEN N = 1: GOTO 2300
2290 N = 0
2300  IF XD > OP OR XD = OP THEN C = 1: GOTO 2320
2310 C = 0: GOTO 3380
2320 OH$ = YH$: GOSUB 1030:YD$ = OD$:YD =  VAL (YD$):OP =
     VAL (OP$): REM  *** CPY ***
2330  IF YD = OP THEN Z = 1: GOTO 2350
2340 Z = 0
2350  IF OP > YD THEN N = 1: GOTO 2370
2360 N = 0
2370  IF YD > OP OR YD = OP THEN C = 1: GOTO 3110
2380 C = 0: GOTO 3380
2390  GOSUB 400: REM  *** EOR ***
2400  FOR L = 1 TO 8:BIT$(L) = "0": NEXT L: FOR L = 1 TO 8
2410  IF B1$(L) = "1" OR B2$(L) = "1" THEN 2440
```

```
2420  IF B1$(L) = B2$(L) THEN 2440
2430 BIT$(L) = "1"
2440  NEXT L: GOTO 2140
2450  IF D = 1 THEN 2480: REM   *** LDA ***
2460 OD$ = OP$: GOSUB 920:AH$ = OH$: GOSUB 1190:AB$ = OB$
2470 TMP =  VAL (OP$): GOSUB 3410: GOTO 280
2480  IF FLAG$ = "AD" AND  VAL (OP$) > 99 THEN 300
2490  IF FLAG$ <  > "AD" THEN OD$ = OP$: GOSUB 920:AH$ =
     OH$: GOTO 2530
2500  IF  LEN (OP$) = 1 THEN OP$ = Z$ + OP$
2510 AH$ = OP$:OH$ = AH$: GOTO 2530
2520 OD$ = OP$: GOSUB 920:AH$ = OH$
2530  GOSUB 1190:AB$ = OB$
2540 TMP =  VAL (OP$): GOSUB 3410: GOTO 2470
2550  IF D = 1 THEN 2580: REM    *** LDX ***
2560 OD$ = OP$: GOSUB 920:XH$ = OH$: GOSUB 1190:XB$ = OB$
2570 TMP =  VAL (OP$): GOSUB 3410: GOTO 280
2580  IF FLAG$ = "AD" AND  VAL (OP$) > 99 THEN 300
2590  IF FLAG$ <  > "AD" THEN OD$ = OP$: GOSUB 920:XH$ =
     OH$: GOTO 2630
2600  IF  LEN (OP$) = 1 THEN OP$ = Z$ + OP$
2610 XH$ = OP$:OH$ = XH$: GOTO 2630
2620 OD$ = OP$: GOSUB 920:XH$ = OH$
2630  GOSUB 1190:XB$ = OB$
2640 TMP =  VAL (OD$): GOSUB 3410: GOTO 2570
2650  IF D = 1 THEN 2680: REM  *** LDY ***
2660 OD$ = OP$: GOSUB 920:YH$ = OH$: GOSUB 1190:YB$ = OB$
2670 TMP =  VAL (OP$): GOSUB 3410: GOTO 280
2680  IF FLAG$ = "AD" AND  VAL (OP$) > 99 THEN 300
2690  IF FLAG$ <  > "AD" THEN OD$ = OP$: GOSUB 920:YH$ =
     OH$: GOTO 2730
2700  IF  LEN (OP$) = 1 THEN OP$ = Z$ + OP$
2710 YH$ = OP$:OH$ = YH$: GOTO 2730
2720 OD$ = OP$: GOSUB 920:YH$ = OH$
2730  GOSUB 1190:YB$ = OB$
2740 TMP =  VAL (OD$): GOSUB 3410: GOTO 2670
2750  GOSUB 380: REM  *** ORA ***
2760  FOR L = 1 TO 8: IF B1$(L) = "1" OR B2$(L) = "1" THEN
     BIT$(L) = "1"
2770  NEXT L
2780 AB$ = "": FOR L = 1 TO 8:AB$ = AB$ + BIT$(L): NEXT L
2790 OB$ = AB$: GOSUB 1270:AH$ = OH$
2800  GOSUB 1030:TMP =  VAL (OD$): GOSUB 3410: GOTO 3390
2810  IF D THEN 3060: REM  *** SBC ***
2820 OP =  VAL (OP$):TMP$ = AB$
2830  GOSUB 1130:MI$ = OB$
2840 OH$ = AH$: GOSUB 1030:AD$ = OD$:AD =  VAL (AD$):TMP = AD
2850 AD = AD - OP: IF C = 0 THEN AD = AD - 1
2860  IF AD < 0 THEN AD = 256 + AD:C = 0
2870 AD$ =  STR$ (AD):OD$ = AD$: GOSUB 920:AH$ = OH$
2880  GOSUB 1190:AB$ = OB$
2890 N = 0: IF AD > 127 THEN N = 1
2900 Z = 0: IF AD = 0 THEN Z = 1
2910 V = 0: IF  LEFT$ (TMP$,1) =  LEFT$ (MI$,1) THEN 2930
2920  IF  LEFT$ (AB$,1) =  LEFT$ (TMP$,1) THEN V = 1
2930 OD$ = AD$: GOSUB 1030:AH$ = OH$: GOTO 280
2940  REM   ***** BCD ADDITION ROUTINE *****
```

```
2950  IF FLAG$ <  > "AD" THEN 1980
2960  IF  LEFT$ (AH$,1) > "9" OR  RIGHT$ (AH$,1) > "9" THEN
     3030
2970 AD =  VAL (AH$)
2980 OP =  VAL (OP$):AD = AD + OP + C:C = 0
2990  GOSUB 1030:TMP =  VAL (OD$): GOSUB 3410: GOTO 3390
3000  IF AD > 99 THEN  GOSUB 3040
3010 AH$ =  STR$ (AD): IF  LEN (AH$) = 1 THEN AH$ = Z$ + AH$
3020 OH$ = AH$: GOSUB 1190:AB$ = OB$: GOTO 280
3030 OH$ = AH$: GOSUB 1030:AD$ = OD$:AD =  VAL (AD$): GOTO
     2980
3040 C = 1:AD = AD - 100: RETURN
3050  REM  ******* BCD SUBTRACTION ROUTINE ***************
     *******
3060  IF FLAG$ <  > "AD" THEN 2820
3070  IF  LEFT$ (AH$,1) > "9" OR  RIGHT$ (AH$,1) > "9" THEN
     3120
3080 AD =  VAL (AH$)
3090 OP =  VAL (OP$):AD = AD - OP: IF C = 0 THEN AD = AD - 1
3100  IF AD < 0 THEN  GOSUB 3130
3110  GOTO 3010
3120 OH$ = AH$: GOSUB 1030:AD$ = OD$:AD =  VAL (AD$): GOTO
     3090
3130 C = 0:AD = 100 + AD: RETURN
3140  REM  ******** IMMEDIATE ADDRESS OPERANDS ************
     ****
3150 OD$ = OP$: GOSUB 920: GOSUB 1190: REM  ASL
3160 C =  VAL ( LEFT$ (OB$,1)):OB$ =  MID$ (OB$,2) + Z$
3170  GOSUB 1270: GOSUB 1030:X =  VAL (OD$): IF X < 0 THEN N
3180  IF X = 0 THEN Z = 1
3190  GOTO 280
3200  GOSUB 1030:AD =  VAL (OD$): REM    *** BIT ***
3210  GOSUB 1130:N =  VAL ( LEFT$ (OB$,1)):V =  VAL ( MID$
     (OB$,2,1))
3220  GOSUB 400:Z = 1: FOR L = 1 TO 8
3230  IF B1$(L) = "1" AND B2$(L) = "1" THEN Z = 0
3240  NEXT L: GOTO 3390
3250  GOTO 2170
3260 OP =  VAL (OP$):OP = OP - 1: IF OP < 0 THEN N: REM  DEC
3270  IF OP = 0 THEN Z = 1
3280  GOTO 3390
3290 OP =  VAL (OP$):OP = OP + 1: IF OP < 0 THEN N: REM  INC
3300  IF OP = Z THEN Z
3310  GOTO 3390
3320 OD$ = OP$: GOSUB 920: GOSUB 1190
3330 C =  VAL ( RIGHT$ (OB$,1)):OB$ = Z$ +  LEFT$ (OB$,7):
     GOTO 3170
3340 OD$ = OP$: GOSUB 920: GOSUB 1190: REM  ROL
3350 TMP = C:C =  VAL ( LEFT$ (OB$,1)):OB$ =  MID$ (OB$,2) +
     STR$ (TMP): GOTO 3170
3360 OD$ = OP$: GOSUB 920: GOSUB 1190: REM  ROR
3370 TMP = C:C =  VAL ( RIGHT$ (OB$,1)):OB$ =  STR$ (TMP) +
     LEFT$ (OB$,7): GOTO 3170
3380  REM  ********** PRINT LINE SPACE & GET ANOTHER LINE
3390  PRINT : GOTO 280
3400  REM  ************** SET Z AND N FLAGS **************
     ***
```

```
3410 N = 0: IF TMP > 127 THEN N = 1
3420 Z = 0: IF TMP = 0 THEN Z = 1
3430  RETURN
3440  IF FLAG$ = "AH" AND ( LEFT$ (AH$,1)) < "A" AND ( MID$
     (AH$,2,1)) < "A" THEN 2480
```

7

Addressing Your Apple

In Chapter 1, I mentioned that there is a one-to-one correlation between assembly language and machine language: for every mnemonic in an assembly-language program, there's a numeric machine-language instruction that means exactly the same thing.

Actually, there are many assembly-language mnemonics that have more than one equivalent instruction in machine language. For example, when you see the mnemonic ADC in an assembly-language program, there are eight different numeric instructions that it can be converted into when it is assembled into machine language. To understand why this is true, it is necessary to know something about how addressing modes are used in 6502B/65C02 assembly language.

An addressing mode is a technique for locating and using information stored in a computer's memory. The 6502 chip has 13 addressing modes, the 65C02 has 15. So your Apple IIc or IIe has either 13 or 15 addressing modes, depending on which chip is installed. In this chapter, we'll examine all fifteen 6502/65C02 addressing modes.

First let's look at Table 7-1, which shows that there are nine addressing modes that can be used with the mnemonic ADC.

If you examine Columns 2 and 3 in Table 7-1, you may notice a curious relationship between the assembly-language statements in Column 2 and their machine-language equivalents in Column 3. In Column 2, labeled "Assembly-Language Statements," each addressing mode uses the same mnemonic but a different operand. In Column 3, labeled "Machine-Code Equivalents,"each statement has the same operand but a different op code.

This relationship illustrates an important difference between assembly language and machine language. When you look at a program written in 6502B/65C02 assembly language, the address mode that is used in each statement in the program is determined by the statement's operand. When a 6502B/65C02 program is translated into machine language, however, the address mode that is used for each statement can be determined by looking at the statement's op code.

Now let's look at Table 7-2, which illustrates the addressing modes that can be used with either the 65C02 or the 6502B chip.

Of the 15 addressing modes illustrated in this table, only two have been used so far in this book: the *immediate* mode (such as LDA #2) and the *absolute* mode (such as LDA $0207). As you may recall from Chapter 2, the operand of a statement written in the immediate addressing mode

Table 7-1. Differences in Assembly-Language and Machine-Language Addressing Modes

COLUMN 1: ADDRESSING MODE	COLUMN 2: ASSEMBLY-LANGUAGE STATEMENTS	COLUMN 3: MACHINE-CODE EQUIVALENTS	COLUMN 4: NO. OF BYTES
Immediate	ADC #$03	69 03	2
Zero Page	ADC #$03	65 03	2
Zero Page,X	ADC $03,X	75 03	2
Absolute	ADC $0300	6D 00 03	3
Absolute Indexed,X	ADC $0300,X	7D 00 03	3
Absolute Indexed,Y	ADC $0300,Y	79 00 03	3
Indexed Indirect	ADC ($03,X)	61 03	2
Indirect Indexed	ADC ($03),Y	71 03	2
Zero-Page Indirect*	ADC ($03)	72 03	2

*65C02 only

Table 7-2. The Addressing Modes of the 65C02/6502B Microprocessor

	ADDRESSING MODE	FORMAT
1.	Implicit (Implied)	RTS
2.	Accumulator	ASL A
3.	Immediate	LDA #2
4.	Absolute	LDA $02A7
5.	Zero Page	STA $33
6.	Relative	BCC LABEL
7.	Absolute Indexed,X	LDA $9000,X
8.	Absolute Indexed,Y	LDA $9000,Y
9.	Zero Page,X	LDA $33,X
10.	Zero Page,Y	STX $33,Y
11.	Indexed Indirect	LDA ($33,X)
12.	Indirect Indexed	LDA ($33),Y
13.	Absolute Indirect	JMP ($089A)
14.	Zero-Page Indirect*	ADC ($0A)
15.	Absolute Indexed Indirect*	JMP ($089A,X)

*65C02 only

is always a literal value, while the operand of a statement written in the absolute mode is always a memory address. If the immediate-mode statement LDA #2 were encountered during the running of an assembly-language rogram, the literal value 2 would be loaded into the 6502B/65C02 accumulator. However, if the absolute-mode statement LDA $0207 were encountered during the course of a program, the value stored in memory register $0207 would be loaded into the accumulator.

Another Look at the ADDRS Program

Program 7-1 is a printout of ADDNRS.SRC, an assembly-language program we have already used in this book. The program is repeated here because it not only contains several addressing modes, it also illustrates how those modes are used in 6502B/65C02 assembly language. This version of the ADDNRS.SRC program was written using the Merlin Pro assembler-editor system; however, if you own an Apple ProDOS assembler or an ORCA/M assembler, you should be able by now to alter the program to meet your assembler's requirements without too many problems, since the major differences in the formats used by these assemblers were described in previous chapters.

Program 7-1
THE ADDNRS PROGRAM

```
 1 *
 2 * ADDNRS.SR2
 3 *
 4  ORG $8000
 5 *
 6 ADDNRS CLD
 7  CLC          ;Implied addressing
 8  LDA #2       ;Immediate addressing
 9  ADC #2       ;Immediate addressing
10  STA $0300    ;Absolute addressing
11  RTS
```

Three address modes are used in the ADDNRS.SRC program, and all three are identified in the comments column of Program 7-1. We'll now examine each of the three address modes used in the ADDNRS routine. (An asterisk indicates modes recognized by 65C02 only.)

Implied (Implicit) Addressing

The implied addressing mode is a mode that does not require—or permit—an operand. When you use implied addressing, all you have to type is a three-letter mnemonic. You never have to specify an operand since an implied operand is contained within each mnemonic used in the implied addressing mode.

From a memory-management point of view, it's a good idea to use as many implied address mnemonics as you can. Since an implied address mnemonic uses no operand, it requires only one byte of memory, rather than the two or three bytes that are consumed by statements written using other types of addressing.

Examples of implied addressing are

```
CLC ;(CLEAR THE CARRY BIT OF THE P REGISTER)
DEX ;(DECREMENT THE X REGISTER)
INY ;(INCREMENT THE Y REGISTER)
```

Op-code mnemonics that can be used in the implicit addressing mode are BRK, CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, PHA, PHP, PHX*, PHY*, PLA, PLP, PLX*, PLY*, RTI, RTS, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS, and TYA.

Immediate Addressing

The immediate addressing mode always requires an operand, and that operand is always a literal number. In a statement that uses immediate addressing, then, a "#" sign—the symbol for a literal number—always appears in front of the operand.

When an immediate address is used in an assembly-language statement, the assembler does not have to peek into a memory location to find a value. Instead, the value itself is placed directly into the accumulator. Then whatever operation the statement calls for can be performed immediately.

A statement written in the immediate addressing mode always requires two bytes of memory: one byte for the op code and one byte for the operand. An immediate mode statement never uses a two-byte operand, since the 6502B/65C02 chip cannot handle any literal number larger than one byte.

Here are some examples of immediate addressing.

```
LDA #2
ADC #$33
SBC #253
```

Instructions that can be used in the immediate address mode are ADC, AND, BIT*, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA, and SBC.

Absolute Addressing

Absolute addressing is another addressing mode that you've encountered in this book. In a statement that uses absolute addressing, the operand is a memory location, not a literal number. The mnemonic in an absolute address statement, then, always calls for an operation to be performed on a value stored in a specified memory location, not on the operand itself.

When a statement is written in the absolute addressing mode, the operand, being a memory address, always requires two bytes of memory. Since your Apple computer can handle addresses up to 16 bits long, two bytes are always reserved for operands that are used in the absolute addressing mode.

Here are some examples of assembly-language statements written in the absolute addressing mode.

```
LDA $0300
STA 768
CMP $FFDO
```

Mnemonics that can be used in the absolute addressing mode are ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, STY, STZ*, TSB*, and TRB*.

Zero-Page Addressing

Zero-page addressing is very similar to absolute addressing. When a statement is written using a zero-page address, a value is retrieved from a specified memory address, and that is the value on which the operand does its work. There is one important difference, though, between an absolute address and a zero-page address. When a statement is written in the absolute address mode, the address that is accessed by the statement can lie in any part of available RAM. When a statement is written using page-zero addressing, however, the address that it accesses must lie in a special area of memory called, logically enough, Page Zero.

Specifically, the memory block in your computer known as Page Zero occupies the 256 memory registers that extend from memory address $00 through memory address $FF. You could also say that Page Zero extends $0000 to $00FF, but it isn't really necessary to use those extra pairs of 0's when you want to refer to a zero-page address. Usually, when you follow an assembly-language instruction with a one-byte address—or with a four-digit hex number that begins with two 0's—your assembler will automatically interpret the address you've specified as a Page Zero address.

Since it only takes one byte to specify a Page Zero address, a statement that uses zero-page addressing requires only two bytes of memory: one for the op code and one for the operand. Unfortunately, however, there often isn't much free memory space on Page Zero, so it isn't usually possible to use a lot of zero-page addressing in assembly-language programs.

In Chapter 11, which is devoted to memory management, there will be more information on Page Zero space and how it's used. For now, here are a few examples of what zero-page addressing looks like in assembly-language programs.

If your computer is equipped with a 6502B chip, the instructions that you can use in the zero-page addressing mode are ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, and STY. If your Apple has a 65C02 chip, you can also use zero-page addressing for several new instructions: STZ ("store a zero in memory"), TRB ("test and reset memory bits with accumulator"), and TSB ("test and set memory bits with accumulator").

Accumulator Addressing

The *accumulator addressing* mode can be used with several mnemonics that perform operations on values stored in the 6502B/65C02 accumulator. The command ASL A, for example, is used to shift each bit in the accumulator by one bit position, with a 0 taking the place of the far-right bit, and with the far-left bit (bit 7) dropping into the carry bit of the processor status (P) register.

If your computer is equipped with a 6502B chip, other instructions that can be used in the accumulator addressing mode are LSR, ROL, and ROR. If your Apple has a 65C02 chip, the accumulator addressing mode can also be used with the mnemonics INA or INC A ("increment accumulator") and DEA ("decrement accumulator").

If you own an ORCA/M assembler, you must use the letter A as an operand when you use the accumulator addressing mode. The Apple ProDOS assembler and the Merlin Pro assembler, however, will accept accumulator addressing mnemonics without their A operands.

Relative Addressing

Relative addressing is used with a programming technique called *conditional branching.* Conditional branching instructions are similar to IF ... GOTO instructions in BASIC; they are used to make programs jump from one sequence of instructions to another when certain specific conditions are fulfilled.

There are eight conditional branching instructions that can be used with the 6502B chip, and there is one additional instruction that can be used with the 65C02. All nine branching instructions begin with B, which stands for "branch to." The extra instruction that can be used with the 65C02 is BRA, which stands for "branch always." As its name implies, the mnemonic BRA will cause a jump to another segment in a program under any condition.

The other eight conditional branching instructions that use relative addressing are BCC (branch to a specified address if the carry flag is clear), BCS (branch if the carry flag is set), BEQ (branch if the result of an operation is equal to 0), BNE (branch if the result of an operation is not equal to 0), BMI (branch if the result of an operation is minus), BPL (branch if the result of an operation is plus), BVC (branch if the overflow flag is clear), and BVS (branch if the overflow flag is set).

How Branching Instructions Are Used The nine conditional branching instructions in 6502B/65C02 assembly language are most often used with threother instructions called *comparison instructions*. Typically, a comparison instruction is used to compare two values with each other, and the conditional branch instruction is then used to determine what should be done if the comparison turns out a certain way.

Usually, a branch instruction causes a program to jump to a specified address if certain conditions are met or not met. A branch might be made, for example, if one number is larger than another, if the two numbers are equal, or if a certain operation results in a positive, negative, or 0 value.

Conditional branching can also be based on the results of arithmetic or logical operations or can be invoked after various kinds of tests on bits, bytes, and other numerical values.

The three comparison instructions in 6502B/65C02 assembly language are

- CMP ("compare the number in the accumulator with ...")
- CPX ("compare the value in the X register with ...")
- CPY ("compare the value in the Y register with ...").

An Example of Conditional Branching Program 7-2, titled ADDCHK, is an example of an assembly-language routine that uses conditional branching. (The routine was typed using a Merlin Pro assembler but can be adapted easily to suit the Apple ProDOS assembler or the ORCA/M.)

Program 7-2
ADDCHK
An Addition Program with Error-Checking

```
 1 *
 2 * ADDCHK
 3 *
 4   ORG $8000
 5 *
 6 ADDCHK LDA #0
 7   STA $9000
 8 *
 9   CLD
10   CLC
11   LDA $0300
12   ADC $0301
13   BCS ERROR
14   RTS
15 *
16 ERROR LDA #1
17   STA $9000
18   RTS
```

The ADDCHK program is an 8-bit addition routine with a simple error-checking utility built in. It adds two 8-bit values, using absolute addressing. If this calculation results in a 16-bit value (a number larger than 255), there will be an overflow error in addition and the carry bit of the processor status register will be set.

Here's how the ADDCHK program works. In the two-line sequence labeled ADDCHK, a 0 is loaded into memory register $9000. Then the carry and decimal flags are cleared and the value in memory register $0300 is added to the value in memory register $0301. If this addition operation results in a carry, the carry bit of the processor status register will be set automatically.

After the addition operation is carried out, a BCS (branch if carry set) instruction is used to test the carry bit of the P register. If the test succeeds, the carry bit is set and the program branches to a routine labeled ERROR. If the carry bit is not set, the program ends.

In the routine labeled error, a flag—the number 1—is loaded into memory register $9000. Then the program ends.

Absolute Indexed Addressing

An indexed address, like a relative address, is calculated by using an offset. In an indexed address, however, the offset is determined by the current contents of the 6502B/65C02's X register or Y register.

A statement containing an indexed address can be written using either of these formats:

```
LDA $02A7,X
LDA $02A7,Y
```

How Absolute Indexed Addressing Works When indexed addressing is used in an assembly-language statement, the contents of either the X register or the Y register (depending upon which index register is being used) are added to the address given in the instruction to determine the final address.

Program 7-3, entitled PRINTIT, is an illustration of a program that makes use of indexed addressing. The program moves byte by byte through a string of ASCII characters, using a built-in Apple routine called COUT to display each character in the string on the screen. When the complete string has been displayed, the program ends.

The starting address of COUT, plus the ASCII code number for a carriage return, are defined in a symbol table that precedes the program.

More details on how programs like this one work will be provided in

Chapter 8. The primary purpose of this example is to present an illustration of indirect addressing.

Program 7-3
PRINTIT
A Screen-Printing Program

```
 1 *
 2 * PRINTIT
 3 *
 4 COUT EQU $FDED
 5 EOL EQU 13
 6 *
 7  ORG $8000
 8 *
 9  JMP PRINTIT
10 *
11 TEXT DB 198,204,193,211,200,160,173,173,160,193,208,208,
   204,197,160
12  DB 207,215,206,197,210,160,194,210,197,193,203,211,160
13  DB 205,193,195,200,201,206,197,160,195,207,196,197,
   161,13
14 *
15 PRINTIT  LDX  #0
16 LOOP LDA TEXT,X
17  JSR COUT
18  CMP #EOL
19  BEQ FINI
20  INX
21  JMP LOOP
22 FINI RTS
23  END
```

Testing for a Carriage Return When Program 7-3 begins, we know that the string ends with a carriage return (ASCII $0D), as strings often do in Apple programs.

As the program proceeds through the string, it tests each character to see whether it is a carriage return. If the character is not a carriage return, the program moves on to the next character. If the character is a carriage return, that means there are no more characters in the string, and the routine ends.

Absolute indexed addressing can be used with these 6502B/65C02 instructions: ADC, AND, ASL (X only), CMP, DEC (X only), EOR, INC (X only), LDA, LDX (Y only), LDY (X only), LSR (X only), ORA, ROL (X only), ROR (X only), SBC, and STA. If your computer is equipped with a 65C02 chip, you can use two additional mnemonics—BIT and STZ (store 0)—in the absolute indexed (X only) addressing mode.

Zero-Page, X Addressing

Zero-Page,X addressing is used just like Absolute Indexed,X addressing. However, the address used in the Zero-Page,X addressing mode must be located on Page Zero. Therefore, this form of addressing uses only one byte of memory as an operand when it is assembled into machine language.

Instructions that can be used in the Zero-Page,X addressing mode are ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA, and STY. If your computer has a 65C02 chip, you can also use the mnemonics BIT and STZ (store 0) in the Zero-Page,X addressing mode.

Zero-Page, Y Addressing

Zero-Page,Y addressing works just like Zero-Page,X addressing but can be used with only two mnemonics: LDX and STX. If it weren't for the Zero Page,Y addressing mode, it wouldn't be possible to use absolute indexed addressing with the instructions LDX and STX.

Indirect Addressing

Indirect addressing can be divided into two subcategories: *indexed indirect addressing* and *indirect indexed addressing.* Those names can be confusing, but there's a trick that you can use to differentiate them. Indexed indirect addressing—which has an X in the first word of its name—is an addressing mode that makes use of the 6502B/65C02 chip's X register. Indirect indexed addressing—which *doesn't* have an X in the first word of its name—uses the 6502B/65C02's Y register.

Both indexed indirect addressing and indirect indexed addressing are used primarily to look up data stored in tables.

Indexed Indirect Addressing Three things happen when a program uses indexed indirect addressing. First, the contents of the X register are added to a zero-page address—not to the contents of the address, but to the address itself. Next, the result of this calculation is interpreted as another zero-page address. Finally, when this second address has been calculated, the value that it contains, as well as the contents of the next address, are combined to form a third address. That third address is the address that will finally be interpreted as the operand of the statement in question.

Here is an example that might help clarify this process. Suppose that memory address $A0 in your computer holds the number $00, that

memory address $A1 holds the number $80, and that the X register holds the number 00.

Here is a short chart illustrating those values.

```
$A0 = #$00
$A1 = #$80
  X = #$00
```

Now let's suppose you are running a program that contains the indexed indirect instruction LDA ($A0,X).

If all of those conditions exist when your computer encounters the instruction LDA ($A0,X), the computer will add the contents of the X register (a 0) to the number $A0. The sum of $A0 and 0 is, of course, $A0. Your computer will then go to memory addresses $A0 and $A1. It will find the number $00 in memory address $A0 and the number $80 in address $A1.

Since 6502B/65C02-based computers store 16-bit numbers in reverse order—low byte first—your computer will interpret the number found in $A0 and $A1 as $8000. So it will load the accumulator with whatever number it finds stored in address $8000.

Now let's imagine that when your computer encounters the statement LDA ($A0,X), its 6502B/65C02 X register holds the number 04 instead of the number 00.

Here is a chart illustrating those values, plus a few more equivalents that we'll be using shortly.

```
$A0 = #$00
$A1 = #$80
$A2 = #$0D
$A3 = #$FF
$A4 = #$FC
$A5 = #$1C
  X = #$04
```

If these conditions exist when your computer encounters the instruction LDA ($A0,X), your computer will add the number $04 (the value in the X register) to the number $A0 and then go to memory addresses $A4 and $A5. In those two addresses, it will find the final address (low byte first, of course) of the data it is looking for—in this case, $1CFC.

Indexed indirect addressing is a rare addressing mode not used in many assembly-language programs. When it is used, its purpose is to locate a 16-bit address in a table of addresses that is stored on Page Zero. Space on Page Zero is hard to find, though, so you probably won't be able to store many address tables there. It's not too likely, then, that you'll find much use for this addressing mode.

Indirect Indexed Addressing Indirect indexed addressing is used much more often than indexed indirect addressing in 6502B/65C02 assembly language.

Indirect indexed addressing uses the Y register (never the X register) as an offset to calculate the final address of the start of a table. The starting (or *base*) address of the table has to be stored on Page Zero but the table itself does not.

When an assembler encounters an indirect indexed address in a program, it first looks into the Page Zero address that is enclosed in the parentheses preceding the Y. The 16-bit value stored in that address and the following address are then added to the contents of the Y register. The value that results is a 16-bit address—the address the statement is looking for.

Here's an example of indirect indexed addressing. If your computer is running a program and comes to the instruction ADC ($A0),Y. The computer will then look into memory addresses $A0 and $A1. Suppose that it finds the number $00 in memory register $A0 and the number $50 in $A1. Suppose also that the Y register contains a 4.

Here is a list that illustrates those conditions.

```
$A0 = #$00
$A1 = #$50
  Y = #$04
```

If these conditions exist when your computer encounters the instruction ADC ($A0),Y, the computer will combine the numbers $00 and $50 and will find (low bit first) the address $5000. It will then add the contents of the Y register (4 in this case) to the number $5000 to arrive at a total of $5004.

The number $5004 is the final value of the operand ($A0,Y). Therefore, the contents of the accumulator will be added to whatever number is stored in memory address $5004.

Once you understand indirect indexed addressing, it can be a very valuable tool in assembly-language programming. Only one address—the starting address of a table—has to be stored on Page Zero, where space is always scarce. Yet that address, added to the contents of the Y register, can be used as a pointer to locate any other address in your computer's memory.

Absolute Indirect Addressing There is only one instruction—JMP—that you can use in the *absolute indirect addressing* mode. When absolute indirect addressing is used, a 16-bit number is placed inside a pair of parentheses that follow the JMP instruction, as shown here.

```
JMP ($0900)
```

This number serves as a pointer to a pair of memory registers which, taken together, contain the address to which the desired jump will be made. Let's suppose, for example, that the memory address $02A7 contains the value $00 and that the address $02A8 holds the value $06. Let's also suppose that the statement JMP ($02A7) is included in an Apple assembly-language program. In this case, the program being executed will jump to the address $0600—*not* to the address $02A7, as it would if the jump instruction were simply JMP $02A7.

Absolute Indexed Indirect Addressing *Absolute indexed indirect addressing* is an address mode that can be used only with the JMP instruction, and only on a computer equipped with a 65C02 chip. This is the format for writing a statement using absolute indexed indirect addressing:

```
JMP ($8000,X)
```

When absolute indexed indirect addressing is used in an assembly-language program, the contents of the address that follows the instruction are added to the X register. The sum of this operation points to a memory address containing the lower byte of the effective address. The next memory location will contain the high byte of the effective address.

Zero-Page Indirect Addressing *Zero-page indirect addressing* is a new address mode that can be used only on a computer equipped with a 65C02 chip. This is the format for using zero-page indirect addressing mode:

```
LDA ($0A)
```

When the zero-page indirect addressing mode is used in an assembly-language program, the byte that follows the instruction is interpreted as zero-page address. This byte contains the low-order byte of the effective address, and the next address on Page Zero contains the high-order byte of the affected address.

In a 65C02-equipped Apple IIc or Apple IIe, zero-page addressing can be used with the following instructions: ADC, AND, CMP, EOR, LDA, ORA, SBC, and STA.

A Pseudo-Address: The Stack

While we're on the subject of 6502B/65C02 addressing modes, let's take a look at a programming tool that's related very closely to addressing: the *hardware stack.*

The hardware stack—often referred to simply as the stack—occupies the 256 bytes of memory from $0100 to $01FF in RAM.

The stack is what programmers sometimes call a LIFO (last-in, first-out) block of memory. It is often compared to a spring-loaded stack of plates in a diner; when you put a number in the memory location on top of the stack, it covers up the number that was previously on top. The number on top of the stack must then be removed before the number under it can be accessed.

Although the plate-stack illustration is a useful technique for describing how the stack works, it is not completely accurate; actually, the stack is nothing but a block of RAM. Let's see what really happens when you place a number on your Apple's hardware stack.

The block of memory in your computer's hardware stack (extending from memory register $0100 to memory register $01FF) is used from high memory down: the first number that is stored on the stack will be in register $01FF, the next number will be placed in register $01FE, and so on. Because of this storage system, the last stack address that can be used is memory register $0100.

Your Apple's 6502B or 65C02 chip keeps track of stack manipulations with the help of a special register called the stack pointer (described briefly in Chapter 3). When there is nothing stored on the stack, the value of the stack pointer is $FF. If you add $100 to that number, you get $01FF—the highest memory address on the stack. This is the address that will be used for the next (or in this case, the first) value that is stored on the stack.

As soon as a value is stored on the stack, your computer's 6502B or 65C02 chip will automatically decrement the stack pointer by one. Each time another value is stored on the stack, the stack pointer will be decremented again. Therefore, the stack pointer will always point to the address of the next value that will be stored on the stack.

Let's suppose that several numbers have been stored on the stack and that we now want to retrieve one of those values. When a number that has been stored on the stack is retrieved, the value of the stack pointer

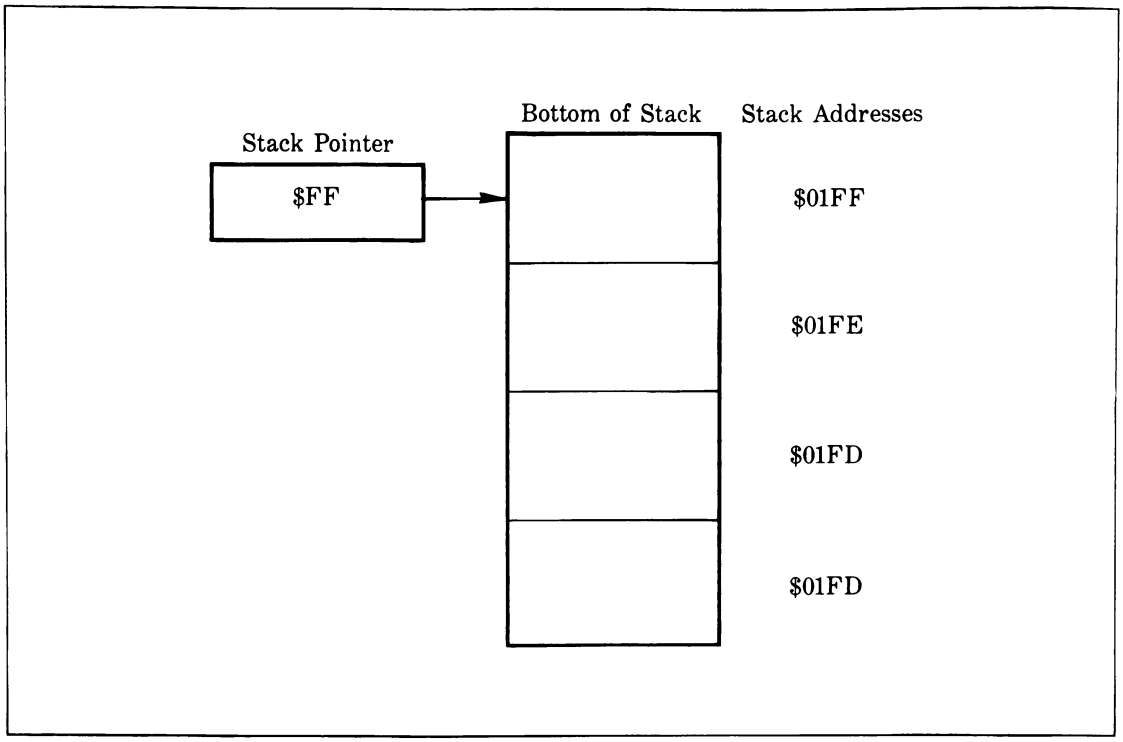

Figure 7-1. How the stack pointer works

is incremented by one. That effectively removes one value from the stack, since it means that the next value stored on the stack will have the same position on the stack as the one that was removed. If you examine Figure 7-1, you'll get an idea of how this process works. Figure 7-1 shows an empty stack, with the stack pointer pointing to the first available address on the stack: $01FF.

In Figure 7-2, a number (whose value is arbitrary) has been placed on the stack. Notice that the value of the stack pointer has been decremented. The number we have placed on the stack is now stored at the highest address in the stack, memory register $01FF.

In Figure 7-3, another number (also with an arbitrary value) has been placed on the stack. As you can see, the stack pointer has been decremented again and a second number is now on the stack.

In Figure 7-4 we'll remove one number from the stack. Stack address $01FE still holds the value $33, but the value of the stack pointer has been decremented and now points to memory address $01FE. The next number that is placed on the stack, then, will be stored at

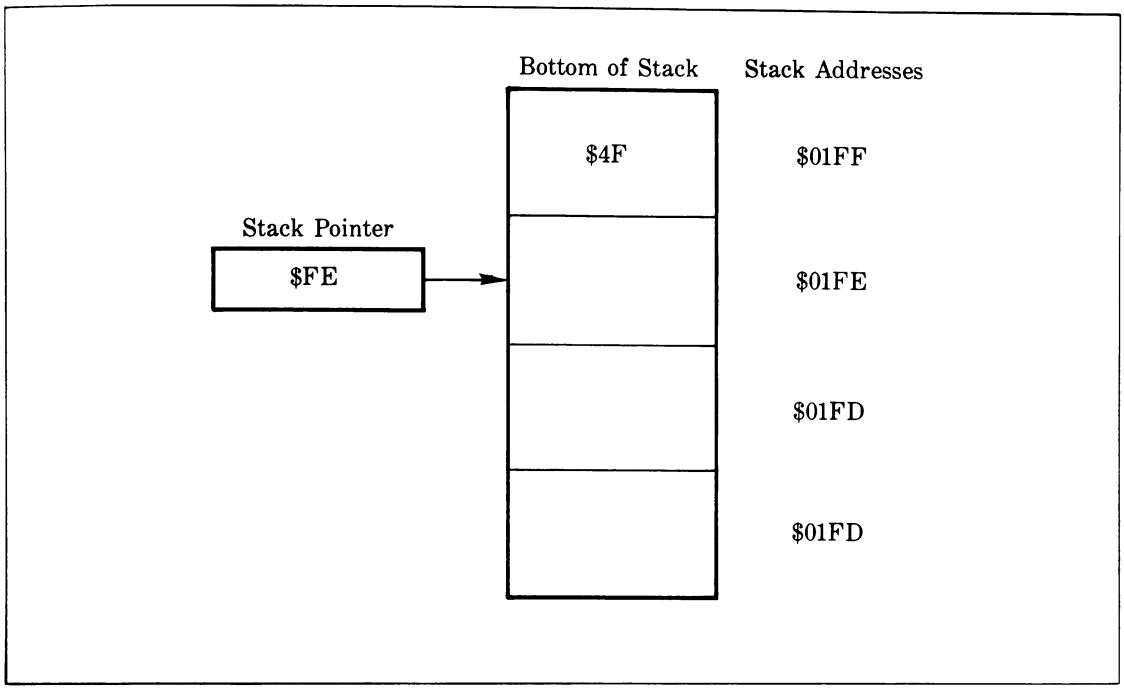

Figure 7-2. Placing a number on the stack

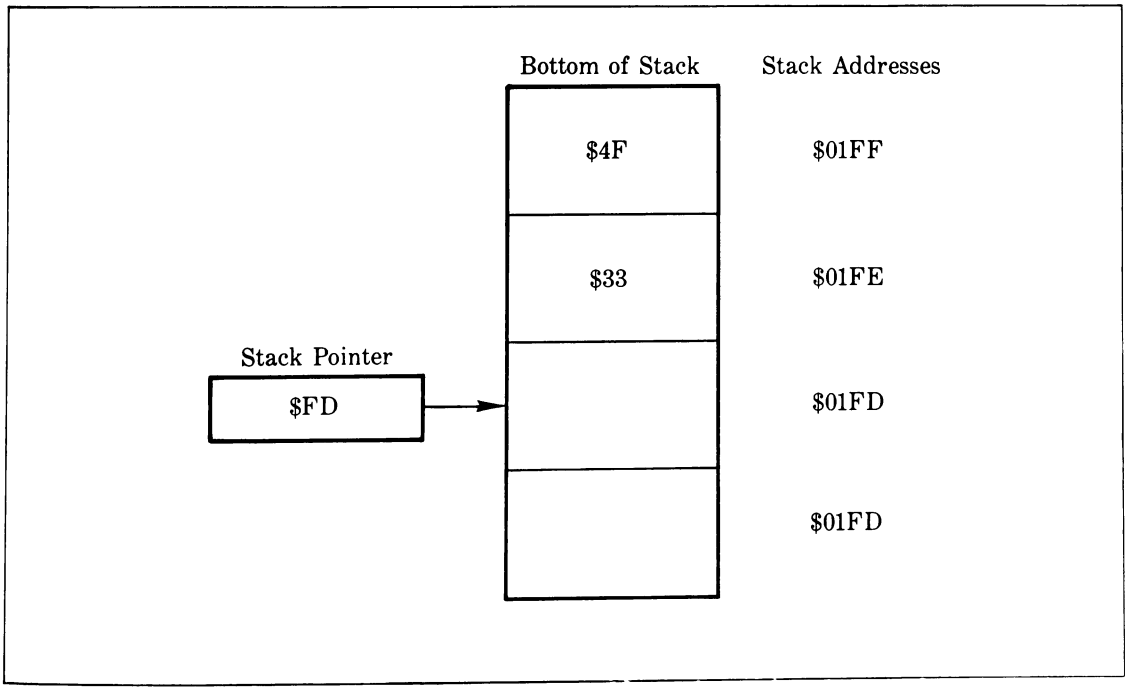

Figure 7-3. Placing another number on the stack

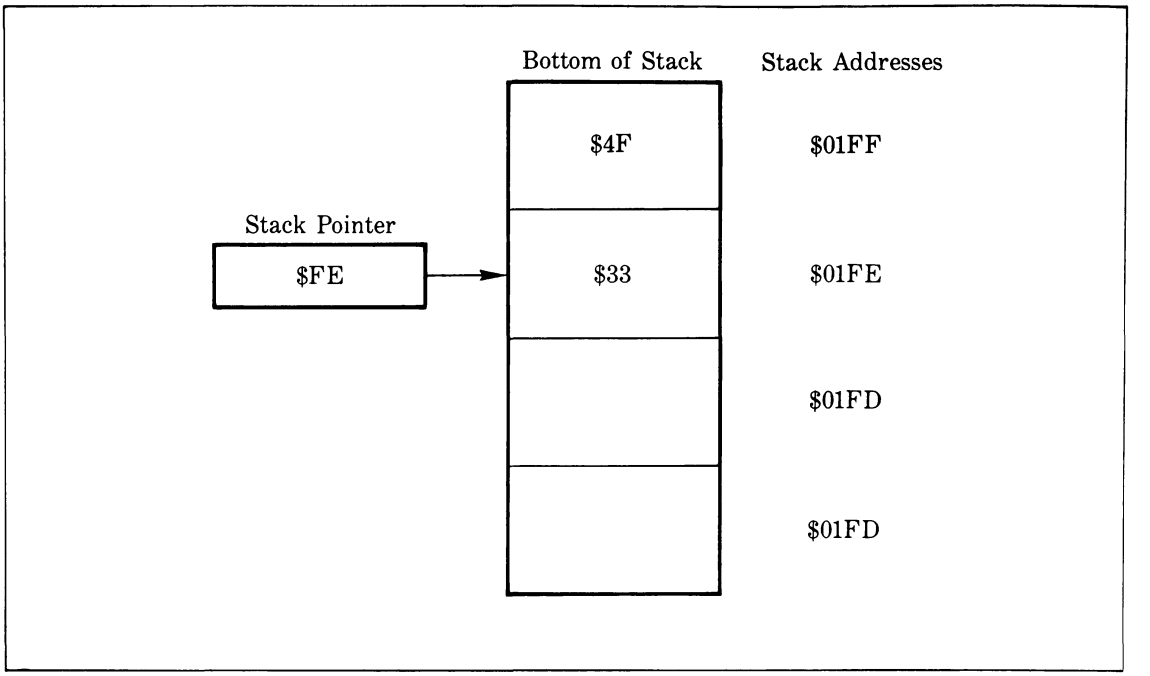

Figure 7-4. Pulling a number off the stack

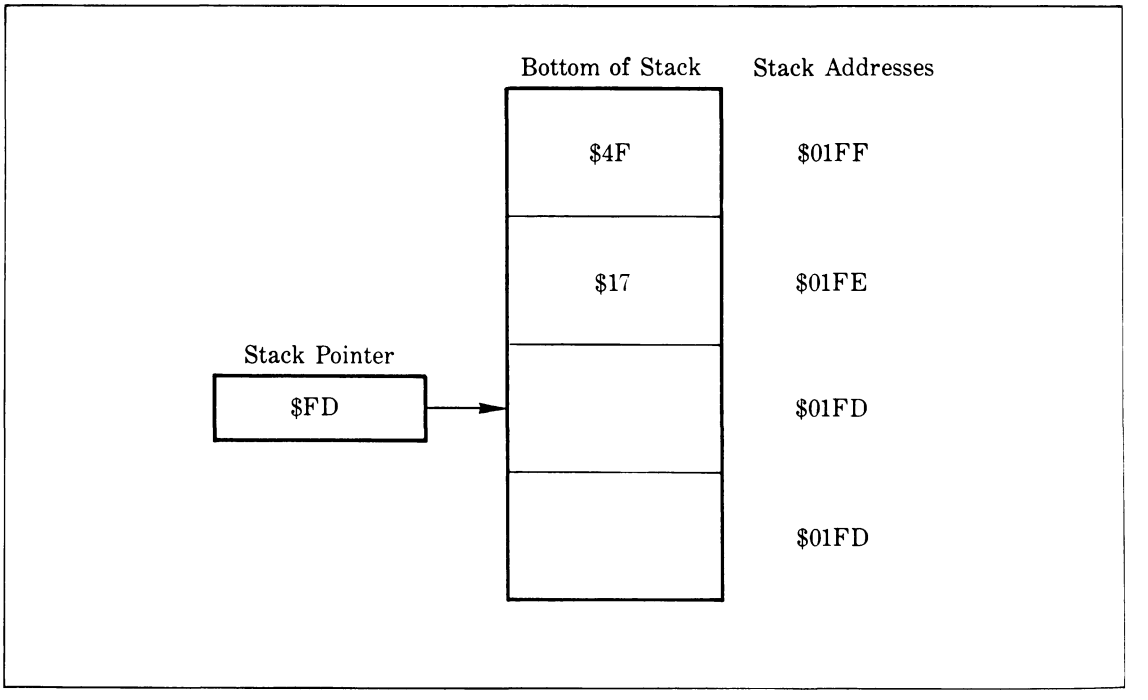

Figure 7-5. One last stack manipulation

memory address $01FE. When that number is stored, the number previously stored in that stack position—$33—will be erased.

Let's store one more number on the stack. This time the value of the number placed on the stack will be $17, as shown in Figure 7-5. Memory register $01FE now holds the value $17. The value of the stack pointer has been decremented, the value $33 has been erased by the value $17, and the next number placed on the stack will be stored in memory register $01FD.

How Your Operating System Uses the Stack

The 6502B/65C02 processor often uses the stack for temporary data storage during the operation of a program. When a program jumps to a subroutine, for example, the 6502B/65C02 chip takes the memory address that the program will later have to return to and pushes that address to the top of the stack. Then, when the subroutine ends with an RTS instruction, the return address is pulled from the top of the stack and loaded into the 6502B/65C02's program counter. The program can then return to the proper address and normal processing can resume.

The stack is also used quite often in user-written programs. Here is an example of a routine that makes use of the stack. You may recognize it as a variation on the 8-bit addition program that we've been using throughout this book.

A Two-Part Program

In Program 7-4, we'll put two 8-bit numbers on the stack. In Program 7-5, we'll take the numbers off the stack and add them. Program 7-4 should be typed first but, before the program is assembled and executed, Program 7-5 should be appended to Program 7-4.

Program 7-4
STACKADD, PART 1
(Putting two number on the stack)

```
1 *
2 *STACKADD
3 *
4   ORG $8000
5 *
6   LDA #35 ;(OR ANY OTHER 8-BIT NUMBER)
7   PHA
8   LDA #49 ;(OR ANY OTHER 8-BIT NUMBER)
9   PHA
```

Now add Program 7-5 to the end of Program 7-4.

Program 7-5
STACKADD, PART 2
(Taking two numbers off the stack and adding them)

```
 1 *
 2 *WHEN THIS PROGRAM BEGINS, TWO
 3 *NUMBERS ARE ON THE STACK
 4   CLD
 5   CLC
 6   PLA
 7   STA $FD
 8   PLA
 9   ADC $FD
10   STA $FE
11   RTS
```

When you add Program 7-5 to Program 7-4, you get a straightforward 8-bit addition routine that shows how convenient it can be to use the stack in assembly-language programs. First, a value is pulled from the stack and stored in the accumulator; then the value is stored in memory address $FD.

Next, another value is pulled from the stack and added to the value now stored in $FD. The result of this calculation is then stored in $FE and the routine ends.

As you can see, the stack can be a very convenient temporary storage area for data. The stack is very memory-efficient, too, since it doesn't require the use of dedicated storage registers. It can also save time, since it takes only one instruction to push a value onto the stack and only one instruction to retrieve a value that has been stored there.

An Important Warning

You must be very careful when using the hardware stack in an assembly-language routine. When the routine ends, it's extremely important to leave the stack exactly as you found it. If you've placed a value on the stack during the course of a routine, it must be removed from the stack before the routine ends and normal processing resumes. Failure to remove the value could result in program crashes, memory wipeouts, and other programming disasters. Nevertheless, if you take care to manage the stack properly, it can be a very powerful programming tool.

Mnemonics that make use of the stack are PHA ("push the contents of the accumulator onto the stack"), PLA ("pull the top value off the stack and deposit it in the accumulator"), PHP ("push the contents of the P register onto the stack"), and PLP ("pull the top value off the

stack and deposit it into the P register"). The instructions JSR and RTS and the 65C02 instructions PHX, PHY, PLX, and PLY also make use of the stack.

The PHP and PLP operations are often included in assembly-language subroutines so that the contents of the P register won't be deleted during subroutines. When you jump to a subroutine that might change the status of the P register, you should start the subroutine by pushing the contents of the P register onto the stack. Then, just before the subroutine ends, you can restore the P register's previous state with a PHP instruction. If you follow this procedure, the contents of the P register won't be destroyed during the course of the subroutine.

8

Looping and Branching

Now that you're familiar with the 6502B/65C02 instruction set and addressing modes, we can start doing some actual programming in Apple IIc/IIe assembly language. In this chapter, you'll learn how to display messages on the screen, encode and decode ASCII characters, and perform a number of other useful procedures in assembly language, including:

- Incrementing and decrementing the X and Y registers.
- Using comparison and branching instructions together.
- Looping and branching.
- Using Apple assembler directives DFB ("define byte"), DB (which means the same thing), and DC ("define constant," a directive used with the ORCA/M assembler package). DFB, DB, and DC are all used to reserve memory space for data in assembly-language programs.

In this chapter we again use the COUT routine that's built into the Apple IIc and the Apple IIe beginning at call address $FDED. We used the COUT routine in Chapter 1 to display the word "HI" on your computer screen in the HI.TEST program.

A New Program

Program 8-1, entitled BULLETIN, is the longest type-and-run program that has been presented in this book. It was written using the Merlin Pro assembler, but it will also run on the Apple ProDOS assembler. An ORCA/M version of the program is provided in Program 8-2.

Program 8-1
THE BULLETIN PROGRAM
(Merlin Pro assembler version)

```
 1 *
 2 * BULLETIN
 3 *
 4  ORG $8000
 5 *
 6 BUFLEN EQU 29
 7 COUT EQU $FDED
 8 *
 9  JMP BEGIN
10 TEXT DB 195,197,204,197,211,212,201,193,204,160
11  DB 193,195,195,201,196,197,206,212,160
12  DB 210,197,208,207,210,212,197,196,173,173
13 *
14 BEGIN LDX #0
15 *
16 LOOP LDA TEXT,X
17  JSR COUT
18  INX
19  CPX #BUFLEN
20  BNE LOOP
21  RTS
```

When you've typed, assembled, and executed the BULLETIN program, you'll see half of a cryptic message on your computer screen. The other half of the message will be presented later in this chapter, in a program called BULLETIN.B.

Saving the BULLETIN Program

Once your BULLETIN program is running properly, save it on a disk in both its source-code and object-code versions. Then we can take a look at how the program works.

Let's start with an explanation of the assembly-language directive DB, which means "define byte." It appears in lines 10 through 12 of the Merlin/Apple assembler version of the program. (In the ORCA/M listing, a slightly different directive, DC, for "define constant," is used. We'll discuss that directive in a moment.)

Directives like DB and DC are pseudo-operation codes (pseudo-ops) because they appear in the op-code column of assembly-language source-code listing but are not actually included in the standard 6502B/65C02 assembly-language instruction set. As mentioned in Chapter 4, the main difference between an op code and a pseudo-op is that an op code tells a microprocessor what to do, while a pseudo-op tells an assembler what to do. When a program is run through an assembler, each op code that program contains is translated into machine language, while each pseudo-op is used as an instruction by the assembler. When an assembler encounters the directive DB, for example, it interprets the numbers that follow the directive as literal numbers that are to be deposited into a consecutive series of registers in a computer's memory. You'll learn about many other kinds of pseudo-ops in later chapters.

Now let's look at the directive DC, which is used in place of DB in the ORCA/M version of the BULLETIN program.

Program 8-2
THE BULLETIN PROGRAM
(ORCA/M Version)

```
          KEEP  BULLETIN
MAIN      START

BUFLEN    EQU   29
COUT      EQU   $FDED

          JMP   BEGIN

TEXT      DC    I1'195,197,204,197,211,212,201,193,204,160'
          DC    I1'193,195,195,201,196,197,206,212,160'
          DC    I1'210,197,208,207,210,212,197,196,173,173'

BEGIN     LDX   #0

LOOP      LDA   TEXT,X
          JSR   COUT
          INX
          CPX   #BUFLEN
          BNE   LOOP
          RTS

          END
```

The directive DC, unlike DB, has to be told how to interpret any string of values that follows it. In the ORCA/M listing of the BULLETIN program, the letter I that follows each DC directive stands for "integer," and the numeral 1 that follows each I means that each value that follows is to be interpreted as a one-byte integer.

Notice that in the ORCA/M program, the numbers following the DC directives are enclosed in single quotes. This procedure is necessary because ORCA/M's DC directive always expects subsequent numbers to be enclosed in delimiters.

Using the X and Y Registers as Counters

As pointed out in Chapter 3, the X and Y registers in the 6502B/65C02 chip can be progressively incremented and decremented during loops in a program. In the BULLETIN program, the X register is incremented from 0 to 29 during a loop to keep track of a string of text characters being displayed on a screen. The characters are expressed as ASCII code numbers, and those code numbers are the values following the DB and DC directives in the BULLETIN program.

It isn't difficult to see how the X-register loop in this program works. First, the statement LDX #0 is used to load the X register with a 0. Then the loop begins. The first statement in the loop is "LDA TEXT,X". This instruction uses indexed addressing to load the accumulator with an ASCII code for a text character. The COUT routine is then used to display each character on the screen. By the time the loop ends, all 29 of the characters that follow the DB or DC directive have been displayed on the screen.

(Incidentally, there's no need for "#" symbols in front of the numbers that follow directives like DB and DC, since they are automatically interpreted as literal numbers by the assemblers that they're designed for.)

Incrementing and Decrementing the X and Y Registers

In line 18 of the BULLETIN program, the mnemonic INX means "increment the X register."

The first time the program progresses through the loop that starts at line 16, the X register will hold a 0. As soon as the COUT routine has displayed its first character on the screen, though, the INX instruction in line 18 will increment that 0 to a 1.

In line 19 we see the instruction CPX #BUFLEN. If you look back at line 6, you'll see that BUFLEN is a constant that has been equated to the number 29. The instruction CPX #BUFLEN, then, means "compare the value in the X register to the literal number 29."

This comparison determines whether all 29 characters have been displayed on the screen. Once you've displayed all 29 characters in the text string, press a carriage return to end the program.

Comparing Values in Assembly Language

There are three comparison instructions in 6502B/65C02 assembly language: CMP, CPX, and CPY.

CMP means "compare to a value in the accumulator." When the instruction CMP is used, followed by an operand, the value expressed by the operand is subtracted from the value in the accumulator. This subtraction operation is not performed to determine the exact difference between these two values, but merely to see whether they are equal, and if they are not equal, which is the larger.

If the value in the accumulator is equal to the tested value, the zero (Z) flag of the processor status (P) register will be set to 1. If the value in the accumulator is not equal to the tested value, the Z flag will be left in a cleared state.

If the value in the accumulator is less than the tested value, the carry (C) flag of the P register will be left in a cleared state. If the value in the accumulator is greater than the tested value, the carry flag will be set.

CPX and CPY work exactly like CMP, except that they are used to compare values with the contents of the X and Y registers. They have the same effects as CMP on the status flags of the P register.

Using Comparison and Branching Instructions Together

The three comparison instructions in Apple assembly language are usually used in conjunction with eight other assembly language instructions —the conditional branching instructions that I mentioned in Chapter 6.

The sample program called BULLETIN contains a conditional branching instruction in line 20. That instruction is BNE LOOP, which means "branch to the statement labeled loop if the zero flag (of the processor status register) is not set." Remember that in the 6502B/65C02 processor status register, the zero flag is *set* (equals 1) if the result of an operation that has just been performed is 0, and the zero flag is *cleared* (equals 0) if the result of an operation that has just been performed is *not* 0.

When your computer encounters the BNE LOOP instruction in line 20, it will keep branching back to line 16 (labeled LOOP) as long as the value of the X register has not yet been decremented to 0.

Once the value of the X register has been decremented to 0, the

statement "BNE LOOP" in line 210 will be ignored and the program will move on to the next line. That line contains an RTS instruction that terminates the program.

Conditional Branching Instructions

We know that there are eight conditional branching instructions in 6502B/65C02 assembly language. They all begin with the letter B, and they're also called relative addressing or branching instructions. Table 8-1 shows these eight instructions and their meanings.

How Branching Differs From Jumping

There is another category of 6502/6502B/65C02 instructions, called *jump instructions*. There are some important differences between jump instructions and branching instructions.

There are two jump instructions in 6502 assembly language: *JMP* and *JSR*. The JMP mnemonic is used much like the GOTO instruction in BASIC; when a JMP instruction is encountered in an assembly-language program, the program jumps to whatever memory address is specified by the operand that follows the JMP instruction.

The assembly-language instruction JSR is used much like BASIC's GOSUB instruction. When a JSR instruction is encountered in an assembly-language program, the memory address of the next instruc-

Table 8-1. Conditional Branching Instructions

BCC—Branch if the carry (C) flag of the processor status (P) register is clear. (If the carry flag is set, the operation will have no effect.)

BCS—Branch if the carry (C) flag is set. (If the carry flag is clear, the operation will have no effect.)

BEQ—Branch if the result of an operation is zero (if the zero [Z] flag is set).

BMI—Branch on minus (if an operation results in a set N [negative] flag).

BNE—Branch if not equal to zero (if the zero [Z] flag isn't set).

BPL—Branch on plus (if an operation results in a cleared negative [N] flag).

BVC—Branch if the overflow (V) flag is clear.

BVS—Branch if overflow (V) flag is set.

tion in the program is stored on the hardware stack. Then the program jumps to whatever memory address is specified by the operand following the JSR instruction.

The mnemonic JSR is designed primarily for use with subroutines. In 6502/6502B/65C02 assembly language, subroutines almost always end with RTS instructions.

In assembly language, RTS is the exact opposite of JSR. When an RTS instruction is encountered in a program, a memory address is removed from the stack and processing immediately jumps to that address. If the RTS instruction has been used to end a subroutine, the address pulled from the stack will usually be the one that was deposited there by the JSR instruction used to invoke the subroutine. Therefore, the processing of the program will resume at the line following the JSR instruction that was used to invoke the subroutine.

We have learned that branching instructions are conditional; jump instructions, however, are *unconditional.* When a jump instruction is encountered in a program, it will always be carried out. When a branching instruction is encountered in a program, it will be carried out only if certain specific conditions are fulfilled.

There is another important difference between a jump instruction and a branching instruction. In machine language, the operand that follows a jump instruction is always expressed as a 2-byte value and is always interpreted as the actual starting address of the destination of the jump instruction. However, when a branching instruction is assembled into machine language, the operand that follows the branching instruction is always converted to a signed 1-byte number. Then, when the program is executed, this signed 1-byte number is interpreted as an *offset* that points to the starting address of the destination of the branch instruction.

Let's look at a sample statement containing a jump instruction.

```
JMP $C000
```

If this statement were assembled into machine language and then executed, the result would be quite straightforward: the value $C000 would be loaded into your computer's program counter and a jump to memory address $C000 would then occur.

Unfortunately, branching instructions are a little more complicated than jump instructions. Program 8-3 is a sample program that uses the branching instruction BCC, which means "branch if carry set." The program is called BRANCHIT.S.

Program 8-3
THE BRANCHIT.S PROGRAM (SOURCE-CODE VERSION)

```
 1  ORG $8000
 2 *
 3 WHAZIS EQU $0300
 4 *
 5  LDA #5
 6  CLC
 7  ADC WHAZIS
 8  BCS RETURN
 9  TAX
10 RETURN RTS
```

Program 8-3 is a very straightforward subroutine. In line 5, the literal number 5 is loaded into the accumulator. Then the 6502B/65C02 carry flag is cleared, and the value stored in memory address $0300 (which has been labeled WHAZIS) is added to the value stored in the accumulator (now 5). Next, in line 8, a branching instruction is invoked. If adding 5 to the value of WHAZIS has resulted in a carry (if the sum of 5 and WHAZIS is greater than 255), the routine will branch to line 10 and will end. However, if the sum of 5 and WHAZIS does not result in a carry—that is, if the sum is less than 255—the sum will be transferred to the X register before the routine ends.

Assembling the BRANCHIT.S Routine

Now take a look at an assembled listing of the BRANCHIT.S program as shown in Table 8-2. Lines 8 through 10 of Table 8-2 show how the branching instruction in the BRANCHIT.S program works. In line 8, to the left of the line number, you see

```
8006:B0 01
```

The first figure in this line, $8006, is the memory address in which the instruction BCS will be stored when it has been assembled into machine language. The second figure in the line, $B0, is the actual machine-language equivalent of the BCS instruction. The third number, $01, is an offset value that must be computed by your computer's 6502B or 65C02 chip before it can carry out the BCS instruction.

Offset Values

In a 6502B/65C02 branching instruction, an offset value is a signed number that must be added to a given memory address in order to com-

Table 8-2. The BRANCHIT.S Program (Assembled Version)

```
8000:                1          ORG       $8000
8000:                2 *
8000:                3 WHAZIS   EQU       $0300
8000:                4 *
8000:A9 05           5          LDA       #5
8002:18              6          CLC
8003:6D A7 02        7          ADC       WHAZIS
8006:B0 01           8          BCS       RETURN
8008:AA              9          TAX
8009:60             10 RETURN   RTS
```

pute the destination address of the branching instruction. The address to which it must be added is always the address that *follows* the statement containing the branching instruction. Therefore, the offset in line 8 of the BRANCHIT.S program is 1. When that 1 is added to the address of the instruction following the branching instruction—$8008—the sum is $8009. That number is the address of the RTS instruction that ends the BRANCHIT.S program.

When you write a branching instruction in assembly language, you can follow it with either a literal address or a label that equates to an address. When your program is assembled into machine language, your assembler will convert that literal address or label into an offset value. From then on, each time your 6502B or 65C02 chip encounters a branching instruction during the execution of the assembled program, it will automatically use the offset that follows each branching instruction to compute the destination address of the branch.

Remember that an offset that follows a branching instruction can never be longer than one byte. Since this one byte is always interpreted by the 6502B/65C02 chip as a signed number, a branching offset can be no smaller than −128 and no larger than +127. Also, since this displacement is always added to the address of the first instruction that follows a branching instruction, the effective displacement of a branching instruction can range only between −126 and +129. Thus, branches that occur as the result of branching instructions are subject to certain

length limitations: the destination address of a branching instruction cannot be more than 126 bytes lower or more than 129 bytes higher than the address of the first instruction that follows the branching instruction.

What if you want to write an instruction that will branch to an address that does not fall within these limitations? If you want to exceed the distance limitations of a branching instruction, you simply use the instruction to branch to a jump instruction that has no such restrictions. Program 8-4 is an example of how that can be done.

Program 8-4
THE BRANCHIT.S PROGRAM (WITH A JUMP INSTRUCTION ADDED)

```
 1   ORG $8000
 2 *
 3 WHAZIS EQU $0300
 4 *
 5   LDA #5
 6   CLC
 7   ADC WHAZIS
 8   BCC CONT
 9   JMP FARJMP ;(CAN BE ANYWHERE IN MEMORY)
10 CONT TAX
11   RTS
```

Long-Distance Branching

In the version of the BRANCHIT.S program shown in Program 8-4, the BCS instruction that appeared in the original program has been replaced by a BCC instruction. A new line, containing a JMP instruction that can jump to any address in memory, has been inserted following the line containing the BCC instruction. In this version of the program, if the addition of 5 to the value of WHAZIS results in a carry, the program will jump to an address labeled FARJMP that can be situated anywhere. Otherwise, the program jumps to line 10, labeled CONT (for "continue"), and proceeds as before.

How Conditional Branching Instructions Are Used

You have seen that the usual way to use a conditional branching instruction in 6502B/65C02 assembly language is to load the X or Y register with a 0 or some other value and then to load the A register (or a memory register) with a value to be used for a comparison. Next you use a conditional branching instruction to tell the computer what P register flags to test and what to do if these tests succeed or fail.

Once you understand the general concept of conditional branching, you can use a simple table, such as Table 8-3, for writing conditional branching instructions.

Table 8-3. Uses of Conditional Branching Instructions

TO TEST FOR:	DO THIS:	AND THEN THIS:
A = VALUE	CMP #VALUE	BEQ
A <> VALUE	CMP #VALUE	BNE
A >= VALUE	CMP #VALUE	BCS
A > VALUE	CMP #VALUE	BEQ and then BCS
A < VALUE	CMP #VALUE	BCC
A = (ADDR)	CMP $ADDR	BEQ
A <> (ADDR)	CMP $ADDR	BNE
A >= (ADDR)	CMP $ADDR	BCS
A > (ADDR)	CMP $ADDR	BEQ and then BCS
A < (ADDR)	CMP $ADDR	BCC
X = VALUE	CPX #VALUE	BEQ
X <> VALUE	CPX #VALUE	BNE
X >= VALUE	CPX #VALUE	BCS
X > VALUE	CPX #VALUE	BEQ and then BCS
X < VALUE	CPX #VALUE	BCC
X = (ADDR)	CPX $ADDR	BEQ
X <> (ADDR)	CPX $ADDR	BNE
X >= (ADDR)	CPX $ADDR	BCS
X > (ADDR)	CPX $ADDR	BEQ and then BCS
X < (ADDR)	CPX $ADDR	BCC
Y = VALUE	CPY #VALUE	BEQ
Y <> VALUE	CPY #VALUE	BNE
Y >= VALUE	CPY #VALUE	BCS
Y > VALUE	CPY #VALUE	BEQ and then BCS

Assembly-Language Loops

In 6502B/65C02 assembly language, comparison instructions and conditional branch instructions are usually used together. In the sample program called BULLETIN, the comparison instruction CPX and the branch instruction BNE are used together in a loop controlled by the incrementation of a value in the X register.

Each time the loop in the program goes through a cycle, the value in the X register is progressively incremented or decremented. Each time the program comes to the line containing the instruction INX, the value in the X register is compared to the literal number 29. When that number is reached, the loop ends.

The program will therefore keep looping back to the line containing the statement JSR COUT until 29 characters have been printed on the screen.

The BULLETIN.B Program

Now we're ready to take a look at a new program entitled BULLETIN.B. Program 8-5 will provide you with the second half of the cryptic message that was presented in the original BULLETIN program. It also contains some improvements that make it more versatile than its predecessor—and easier to understand.

Program 8-5
BULLETIN.B
(Merlin Pro Assembler Version)

```
 1 *
 2 * BULLETIN.B
 3 *
 4  ORG $8000
 5 *
 6 EOL EQU 13
 7 BUFLEN EQU 40
 8 FILLCH EQU $20
 9 COUT EQU $FDED
10 *
11  JMP START
12 *
13 TEXT ASC "A DOGMA GOT HIT BY A KARMA"
14  DB EOL
15 *
16 * CLEAR TEXT BUFFER
17 *
18 START LDA #FILLCH
19  LDX #BUFLEN
20 STUFF DEX
21  STA TXTBUF,X
22  BNE STUFF
23 *
24 * STORE MESSAGE IN BUFFER
25 *
26  LDX #0
27 LOOP1 LDA TEXT,X
28  STA TXTBUF,X
29  CMP #EOL
30  BEQ PRINT
31  INX
32  CPX #BUFLEN
33  BCC LOOP1
34 *
35 * PRINT MESSAGE
36 *
37 PRINT LDX #0
38 LOOP2 LDA TXTBUF,X
39  PHA
40  JSR COUT
41  PLA
```

```
42   CMP #EOL
43   BNE NEXT
44   JMP FINI
45 NEXT INX
46   CPX #BUFLEN
47   BCC LOOP2
48 *
49 FINI RTS
50 *
51 TXTBUF DS BUFLEN
```

Program 8-5 was created on a Merlin Pro assembler but with minor modifications will also work on an Apple ProDOS assembler system. Program 8-6 is the same program typed on an ORCA/M system.

Program 8-6
BULLETIN.B
(ORCA/M Assembler Version)

```
          KEEP BULLETIN

MAIN      START

EOL       EQU   13
BUFLEN    EQU   40
FILLCH    EQU   $20
COUT      EQU   $FDED

          LDA   #FILLCH
          LDX   #BUFLEN
STUFF     DEX
          STA   TXTBUF,X
          BNE   STUFF

          LDX   #0
LOOP1     LDA   TEXT,X
          STA   TXTBUF,X
          CMP   #EOL
          BEQ   PRINT
          INX
          CPX   #BUFLEN
          BCC   LOOP1

PRINT     LDX   #0
LOOP2     LDA   TXTBUF,X
          PHA
          JSR   COUT
          PLA
          CMP   #EOL
          BNE   NEXT
          JMP   FINI
NEXT      INX
          CPX   #BUFLEN
          BCC   LOOP2

FINI      RTS
```

```
TEXT     DC     C'A DOGMA GOT HIT BY A KARMA'
         DC     H'0D'

TXTBUF   DS     40

         END
```

As you can see, the BULLETIN.B program is quite similar to the original program. After you've typed, assembled, and run the BULLETIN.B program, you'll see that it performs essentially the same kind of operation as BULLETIN, but in a slightly more elegant way. The most obvious difference between the two programs is the way they handle text strings. The original BULLETIN program made use of a text string composed of ASCII codes. You'll see when you type BULLETIN.B that there's usually no need to convert text strings into ASCII code numbers in order to use them in assembly-language programs. All three of the assemblers that were used to create the programs in this book are equipped with features that will do that job automatically.

Another important difference between BULLETIN.B and its predecessor is the way the loop that reads the characters is written. In BULLETIN, the loop counted the number of characters that had been printed on the screen and ended when the count reached 29. That's a perfectly good system for printing text strings that are 29 characters long. It won't display strings of other lengths, however, so it isn't a very versatile routine for displaying characters on a screen.

Testing for a Carriage Return

BULLETIN.B is more versatile than BULLETIN because it can display strings of almost any length on a screen. The BULLETIN.B program doesn't keep track of the number of characters it has printed by maintaining a running count of how many letters have been displayed. Rather, when the program encounters a character, it tests the character to see if its value is $0D—the ASCII code for a carriage return or end-of-line (EOL) character. If the character is not an EOL, the computer displays it and goes on to the next character in the string. If the character is an EOL character, the computer displays a carriage return on the screen and the routine ends.

Another difference between BULLETIN and BULLETIN.B is that the latter program doesn't read characters and display them in the same step. Instead, the characters are first placed in a *buffer*, and then the contents of the buffer are printed on the screen.

Text buffers are often used in assembly-language programs because they are both versatile and easy to use. Text can be loaded into a buffer

from a keyboard, for example, or from a telephone modem, or even directly from a computer's memory. Conversely, once a string is in a buffer, it can be removed from the buffer in just as many different ways—no matter how the characters got into the buffer in the first place, and no matter what characters they are. Thus, once a few subroutines have been written to fill a buffer and then to process it in some manner, those subroutines can be used for many different purposes. A buffer can therefore serve as a central repository for text strings, making them easily accessible.

Clearing a Text Buffer

Before you use a text buffer, it's always a good idea to clear it of leftover characters, so a buffer-clearing routine has been written into the BULLETIN.B program. It's a short and simple routine that will clear a text buffer—or any other block of memory that doesn't exceed its length limitations—and will fill the buffer with spaces, 0's, or any other value you choose. In the BULLETIN.B program, the routine fills the buffer with a string of spaces; they will appear as blank spaces on your screen.

As you continue to work with assembly language, you'll find that memory-clearing routines such as this one are useful in many kinds of programs. Word processors, telecommunications programs, and many other kinds of software packages make extensive use of routines that can clear values from blocks of memory and replace them with other values.

The memory-clearing routine in the BULLETIN.B program uses indexed (direct) addressing and an X register countdown. It will fill each memory address in a text buffer (TXTBUF) with a designated "fill character" (FILLCH). Then the routine ends.

The buffer-clearing routine in BULLETIN.B will work with any 8-bit fill character and with any buffer length (BUFLEN) up to 255 characters. Later on in this book, you'll find some 16-bit routines that can fill longer blocks of RAM with values.

One More Program: THE NAME GAME

The final program in this chapter will make use of many of the programming techniques we've learned so far. Program 8-7 is called THE NAME GAME. It was written on an Apple IIe using a Merlin Pro assembler. With minor modifications, you can also type, assemble, and run it with an Apple ProDOS assembler. If you own an ORCA/M system,

you should be able by now to make the modifications needed to type and assemble the program using the ORCA/M.

Program 8-7
THE NAME GAME

```
 1 *
 2 * THE NAME GAME
 3 *
 4  ORG $8000
 5 *
 6 EOL EQU $0D ;RETURN KEY
 7 EOF EQU $03
 8 FILLCHR EQU $20 ;SPACE KEY
 9 BUFLEN EQU 40
10 GETLN1 EQU $FD6F ;ROUTINE TO GET A LINE OF TEXT FROM
   KEYBOARD
11 COUT EQU $FDED ;ROUTINE TO PRINT A CHARACTER ON THE
   SCREEN
12 TEMPTR EQU $FB
13 OSBUF EQU $200
14 *
15  JMP BEGIN
16 *
17 COUNT DS 1
18 INPBUF DS 80
19 *
20 TITLE ASC "THE NAME GAME"
21  HEX 0D
22 HELLO ASC "HELLO, "
23  HEX 03
24 QUERY ASC "WHAT IS YOUR NAME?"
25  HEX 0D
26 NAME ASC "GEORGE"
27  HEX 0D
28 REBUFF ASC "GO AWAY, "
29  HEX 03
30 DEMAND ASC "BRING ME GEORGE!"
31  HEX 0D
32 GREET ASC "HI, GEORGE!"
33  HEX 0D
34 *
35 * CLEAR TEXT BUFFER
36 *
37 FILL LDA #FILLCHR
38  LDX #BUFLEN
39 FILLOOP DEX
40  STA INPBUF,X
41  BNE FILLOOP
42  RTS
43 *
44 PRINT LDY #0
45 SHOW LDA (TEMPTR),Y
46  CMP #EOF
47  BEQ DONE
```

```
 48  PHA
 49  JSR COUT
 50  PLA
 51  CMP #EOL
 52  BNE NEXT
 53  JMP DONE
 54 NEXT INY
 55  CPY #BUFLEN
 56  BCC SHOW
 57 DONE RTS
 58 *
 59 BEGIN LDA #0
 60  STA $C00A ;TURN ON 80-COLUMN FIRMWARE
 61  JSR $C300 ;TRANSFER CONTROL TO 80-COL CARD
 62  STA $C00D ;TURN ON 80-COLUMN DISPLAY
 63  JSR $FC58 ;CLEAR SCREEN & HOME CURSOR
 64 *
 65 * PRINT 'THE NAME GAME'
 66 *
 67  LDA #EOL
 68  JSR COUT
 69  LDA #<TITLE
 70  STA TEMPTR
 71  LDA #>TITLE
 72  STA TEMPTR+1
 73  JSR PRINT
 74  LDA #EOL
 75  JSR COUT
 76 *
 77 * PRINT 'HELLO . . .'
 78 *
 79  LDA #<HELLO
 80  STA TEMPTR
 81  LDA #>HELLO
 82  STA TEMPTR+1
 83  JSR PRINT
 84 *
 85 * PRINT 'WHAT IS YOUR NAME?'
 86 *
 87 ASK LDA #<QUERY
 88  STA TEMPTR
 89  LDA #>QUERY
 90  STA TEMPTR+1
 91  JSR PRINT
 92  LDA #EOL
 93  JSR COUT
 94 *
 95 * INPUT A TYPED LINE
 96 *
 97  JSR GETLN1
 98  STX COUNT
 99  LDY #0
100 PRLOOP LDA OSBUF,Y
101  STA INPBUF,Y
102  INY
103  DEX
```

```
104  BNE PRLOOP
105 *
106 * CHECK TO SEE IF NAME IS GEORGE
107 *
108  LDX COUNT
109  CPX #6
110  BNE NOGOOD
111  DEX
112 CHECK LDA INPBUF,X
113  CMP NAME,X
114  BNE NOGOOD
115  DEX
116  BNE CHECK
117  JMP DUNIT
118 *
119 * NO; PRINT 'GO AWAY . . .'
120 *
121 NOGOOD LDA #EOL
122  JSR COUT
123  LDA #<REBUFF
124  STA TEMPTR
125  LDA #>REBUFF
126  STA TEMPTR+1
127  JSR PRINT
128 *
129 * PRINT PLAYER'S NAME
130 *
131  LDA #<INPBUF
132  STA TEMPTR
133  LDA #>INPBUF
134  STA TEMPTR+1
135  JSR PRINT
136  LDA #EOL
137  JSR COUT
138 *
139 * PRINT 'BRING ME GEORGE!'
140 *
141  LDA #<DEMAND
142  STA TEMPTR
143  LDA #>DEMAND
144  STA TEMPTR+1
145  JSR PRINT
146  LDA #EOL
147  JSR COUT
148  JMP ASK
149 *
150 * YES; PRINT GREETING
151 *
152 DUNIT JSR $FBDD ;SOUND A BEEP
153  LDA #EOL
154  JSR COUT
155  LDA #<GREET
156  STA TEMPTR
157  LDA #>GREET
158  STA TEMPTR+1
159  JSR PRINT
160  RTS
```

If you've typed, assembled, and executed the programs called BULLETIN and BULLETIN.B, you shouldn't have much trouble understanding how THE NAME GAME works. Using several fairly simple subroutines, it displays a short message on your screen and then waits for you to type a response. If the program considers your response incorrect, it prompts you to try again. When you finally enter the line the program is looking for, you get a "reward" message and the program ends.

The GETLN1 Routine

In addition to the kernel routine COUT, which was used in the BULLETIN and BULLETIN.B programs to display characters on the screen, THE NAME GAME makes use of a built-in routine called GETLN1 that can *read* lines of text that have been entered on your Apple's keyboard. The call address of the GETLN1 routine is $FD6F, as you can see in line 10 of THE NAME GAME program. To use the GETLN1 routine in an assembly-language program, you list the routine's address in the program's symbol table, and then write a statement like

```
JSR GETLN1
```

If you look at line 97 of THE NAME GAME program, you'll see that statement. When you run THE NAME GAME, that line is where your computer will stop and wait for you to type a name. As soon as you type a name, plus a carriage return, your Apple will store the line you've typed in a special buffer that begins at memory address 200. The number of characters in the line that you've typed will be left in the X register of your computer's 6502B/65C02 microprocessor.

In THE NAME GAME's symbol table, the buffer that starts at memory register 200 is called OSBUF. The contents of this buffer can change very quickly. Therefore, as soon as the buffer is used in THE NAME GAME program, its contents are copied into another buffer called INPBUF. (This procedure is carried out in lines 98 through 104.)

In lines 108 through 117, a check is made to see whether the name typed into OSBUF is George. If it isn't, the program will jump to a routine that demands to see George. If the name typed in turns out to be George, the computer will respond with a beep and a greeting.

Using Your Apple's 80-Column Display

Lines 59 and 60 of THE NAME GAME initialize the 80-column firmware that enables the Apple IIc and the Apple IIe to generate an 80-

column screen. In line 61, control is transferred to the 80-column firmware with the assembly-language statement JSR $C300, which accomplishes the same task as the BASIC statement PR#3. In line 62, your computer's 80-column display is actually turned on with the statement STA $C00D. Finally, in line 63, the program calls a built-in routine that starts at $FC58. This routine clears the screen display and places a cursor in the upper-left corner of the screen.

Full details of how these routines work can be found in the Apple IIc reference manual and the Apple IIe reference manual. If, however, you just want to know how to use your computer's 80-column capabilities in an assembly-language program, lines 59 through 63 of THE NAME GAME contain everything you need to know.

Low-Byte and High-Byte Symbols

The technique used in THE NAME GAME for storing 16-bit numbers in high-order and low-order 8-bit memory locations is worth special mention. The technique first appears in lines 69 through 72.

```
69  LDA #<TITLE
70  STA TEMPTR
71  LDA #>TITLE
72  STA TEMPTR+1
```

You can probably see what that sequence does. In source code recognized by most assemblers, the string "#<HELLO" means "the low byte of the address labeled 'HELLO'", and the string "#>HELLO" means "the high byte of the address labeled 'HELLO'". (This string refers not to the contents of the address, by the way, but to the address itself since, in 6502/6510/8502 assembly language, the symbol "#" is used to identify a literal number.)

The above sequence of code first stores the low-order byte of the 16-bit address of the string "HELLO" into the memory location labeled "TEMPTR" (which, as you can see by looking at the program's symbol table, is memory address $FB). Then it stores the high-order byte of the address of the string labeled "HELLO" into memory location TEMPTR+1, or $FC.

Playing THE NAME GAME

When THE NAME GAME has called your Apple's 80-column firmware into action, the program will continue by displaying its title on the screen. The next two routines display the line "Hello, what is your name?"

Then the program clears the text buffer and waits for you to type a response. As you type your answer, each character you enter will be placed in the text buffer. That process will continue until you stop typing characters and press a carriage return.

Next the program will examine the characters that you've entered to see whether they spell the name George. If they don't, the program will demand: "BRING ME GEORGE!" When you finally type the name George, your computer will reward you with a beep and proclaim: "HI, GEORGE!"

You will find that the principles used to create the input and output routines for THE NAME GAME are used in many assembly-language programs. So, if you know how THE NAME GAME works, you've learned quite a bit—by George!

9

Single-Bit Manipulations of Binary Numbers

There are 65,536 bytes of memory in an unexpanded Apple IIe computer and 131,072 bytes of memory in an Apple IIc or a fully expanded IIe. Since there are eight bits in every byte, there are 524,288 bits in a no-frills Apple IIe and 1,048,576 bits in a 128K Apple IIe or an off-the-rack Apple IIc. What does that mean to an assembly-language programmer? If you know how to perform single-bit operations on binary numbers, you can control every binary bit in your Apple automatically. That's a tremendous amount of control to have over a computer.

In the first chapter of this book, you learned how to control one of the most important bits in your Apple's central microprocessor: the carry bit of the 6502B/65C02 processor status register. Manipulating the P register's carry bit is one of the most important bit-manipulation techniques in 6502B/65C02 assembly language. You've also had considerable

experience in using the carry bit in addition programs.

In this chapter, you'll have an opportunity to learn some new techniques using the carry bit of the P register.

Using the Carry Bit In Bit-Shifting Operations

You know that your Apple's 6502B/65C02 microprocessor is an 8-bit chip; it cannot perform operations on numbers larger than 255 without going through a number of steps.

The 6502B/65C02 must split a larger number into 8-bit chunks and then perform the requested operations on each part of the number. The number must then be made whole again.

Once you're familiar with this process, it isn't nearly as difficult as it sounds. In fact, the "scissors" that are used for this electronic cutting and pasting are actually contained in one tiny bit—the carry bit of the 6502B/65C02's P register.

Bit-Shifting Instructions

You've seen how how carry operations work in several programs in this book. In order to get a clearer look at how the carry works in 6502B/65C02 arithmetic, though, it would be useful to examine four very specialized machine-language instructions: *ASL* (arithmetic shift left), *LSR* (logical shift right), *ROL* (rotate left), and *ROR* (rotate right). These four instructions are used very extensively in 6502B/65C02 assembly language.

ASL (Arithmetic Shift Left)

As you will recall from Chapter 2 (the chapter on binary arithmetic), every number that ends in 0 in binary notation is double the preceding binary number that ends in 0. For example, 1000 0000 ($80) is double the number 0100 0000 ($40), which is double the number 0010 0000 ($20), which is double the number 0001 0000 ($10).

Therefore, it is extremely easy to multiply a binary number by 2. You just shift every bit in the number one space to the left and place a 0 in the bit that has been emptied by this shift (bit 0, or the far-right bit of the number). If bit 7 (the far-left bit) of the number to be doubled is a 1, then provision must be made for a carry.

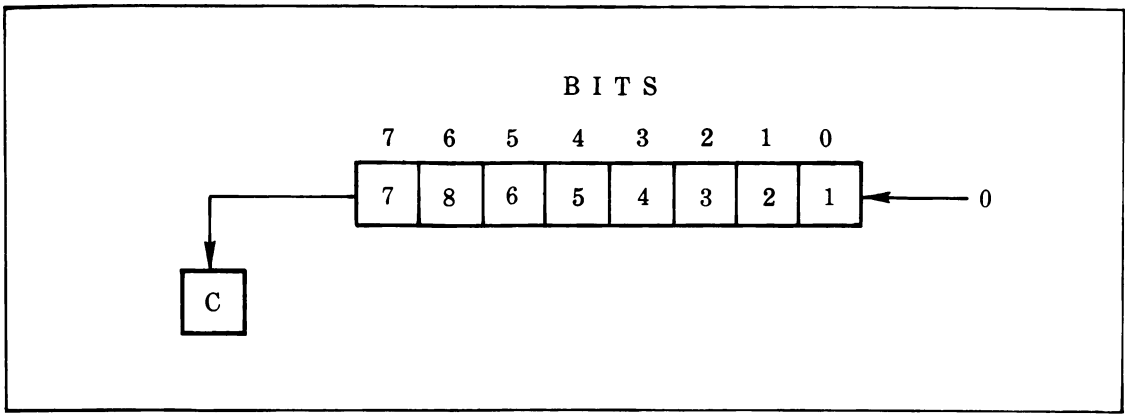

Figure 9-1. The ASL instruction

The entire operation we just described—shifting a byte left, with a carry—can be performed by a single instruction in 6502B/65C02 assembly language. That instruction is ASL, which stands for "arithmetic shift left." As illustrated in Figure 9-1, the instruction ASL moves each bit in an 8-bit number one space to the left—each bit, that is, except bit 7. That bit drops into the carry bit of the processor status (P) register.

The ASL instruction has many uses in 6502B/65C02 assembly language. For instance, it is an easy way of multiplying numbers by 2. Program 9-1 is a number-doubling routine as it might look in a program created using a Merlin Pro or Apple ProDOS assembler.

Program 9-1
DOUBLING A NUMBER WITH THE ASL INSTRUCTION

```
1 *
2  ORG $8000
3 *
4  LDA #$40 ;REM 0100 0000
5  ASL A ;SHIFT VALUE IN ACCUMULATOR TO LEFT
6  STA $FB
7  RTS
```

If you run Program 9-1 and then use the machine-language monitor to examine the contents of memory address $FB, you'll see that the number $40 (0100 0000) has been doubled to $80 (1000 0000) before being stored in memory address $FB.

Another use for the ASL instruction is to pack data, which increases a computer's effective memory capacity. Later in this chapter, there will be an example of how to pack data using the ASL instruction.

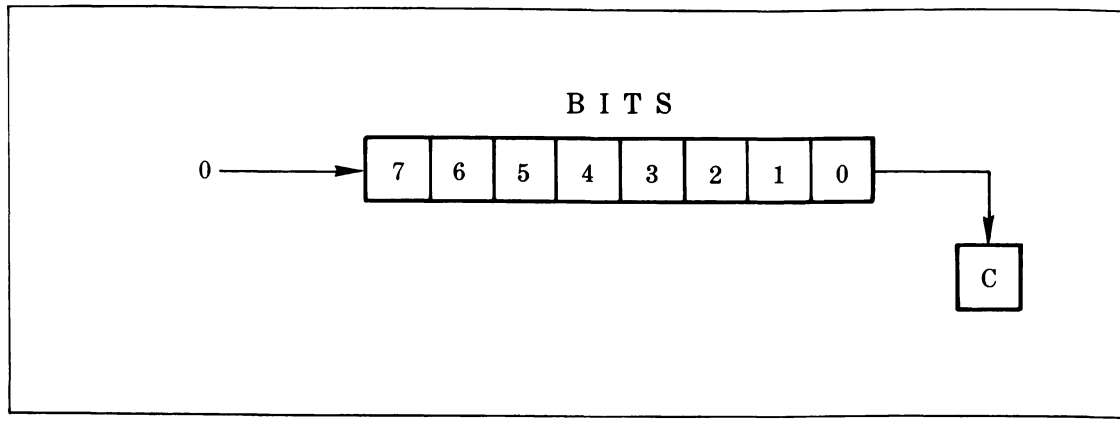

Figure 9-2. The LSR instruction

LSR (Logical Shift Right)

As shown in Figure 9-2, the instruction LSR (logical shift right) is the exact opposite of the instruction ASL.

How the LSR Instruction Works LSR, like ASL, works on whatever binary number is in the 6502B/65C02's accumulator. However, it will shift each bit in the number one position to the *right*. Bit 7 of the new number (left empty by the LSR instruction) will be filled with a 0, and the LSR will be dumped into the carry flag of the P register.

As illustrated in Program 9-2, the LSR instruction can be used to divide any even 8-bit number by 2.

Program 9-2
DIVIDING A NUMBER BY 2 WITH THE LSR INSTRUCTION

```
 1 *
 2 * LSRDIV
 3 *
 4 VALUE1 EQU $FB
 5 VALUE2 EQU $FC
 6 *
 7  ORG $8000
 8 *
 9  LDA #6 ;OR ANY OTHER 8-BIT NUMBER
10  STA VALUE1
11 *
12 *NOW WE'LL DIVIDE BY 2
13 *
14  LDA VALUE1
15  LSR A
16  STA VALUE2
17  RTS
```

In this program, the number stored in VALUE1 is divided by 2 and the result is stored in VALUE2. This division routine will also tell you whether the number it has divided is odd or even. It leaves that piece of information in the carry bit of the 6502B/65C02 P register; if the routine leaves the carry bit clear, the number that was just divided is even. If the carry bit is set, the value is odd.

Later in this chapter, we will learn how to use LSR to *unpack* data.

Another Test for Odd or Even Program 9-3 is another routine that can determine whether a number is even or odd. In lines 12 and 13 of the ODDTEST routine, a memory register called FLGADR (for "flag address") is cleared to 0. Then the contents of the memory register called VALUE1 are shifted to the right one position and stored in a third register, VALUE2. If the value being shifted is even, the shift operation does not set the carry bit, and the subroutine ends. If the value being shifted is odd, the operation does set the carry bit, the program jumps to line 220, and the set carry bit is rotated into the FLGADR register using an instruction called ROL. (You'll learn more about ROL in a moment.) Thus, if the routine leaves a 0 in FLGADR, the number that was divided is even; if the routine ends with a 1 stored in FLGADR, the number that was divided is odd.

Program 9-3
THE ODDTEST ROUTINE

```
 1 *
 2 * ODDTEST
 3 *
 4 VALUE1 EQU $FB
 5 VALUE2 EQU $FC
 6 FLGADR EQU $FD
 7 *
 8  ORG $8000
 9 *
10  LDA #7 ;(ODD)
11  STA VALUE1
12  LDA #0
13  STA FLGADR ;CLEARING FLGADR
14 *
15  LDA VALUE1
16  LSR A ;PERFORM THE DIVISION
17  STA VALUE2 ;DONE
18 *
19  BCS FLAG
20  RTS ;END ROUTINE IF CARRY CLEAR . . .
21 *
22 FLAG
24  ROL FLGADR
25  RTS ;. . . AND END THE PROGRAM
```

The use of LSR to unpack data that has been packed using ASL will be discussed later in this chapter.

ROL (Rotate Left) and ROR (Rotate Right)

The instructions ROL (rotate left) and ROR (rotate right) are also used to shift bits in binary numbers, but they use the carry bit differently from ASL and LSR. Figure 9-3 illustrates how the ROL instruction works.

ROL, like ASL, can be used to shift the contents of the accumulator or a memory register one place to the left. Unlike ASL, however, ROL does not place a 0 in the bit 0 position of the number being shifted into the carry bit. Instead, it rotates the carry bit into bit 0 of the register being shifted and then moves every other bit in that register one place to the left, rotating bit 7 back into the carry bit. If the carry bit is set when that happens, a 1 is placed in the bit 0 position of the byte being shifted. If the carry bit is clear, a 0 goes into the bit 0 position of the shifted register.

Figure 9-4 illustrates the use of the ROR instruction. ROR works just like ROL, but in the opposite direction. It moves each bit of the byte being shifted one position to the right and rotates the carry bit into the bit 7 position of the shifted byte. Bit 0 of the shifted byte is moved into the carry bit of the P register.

ROL and ROR are often used in 6502B/65C02 multiplication and division routines, as well as in other routines in which bits are shifted and tested.

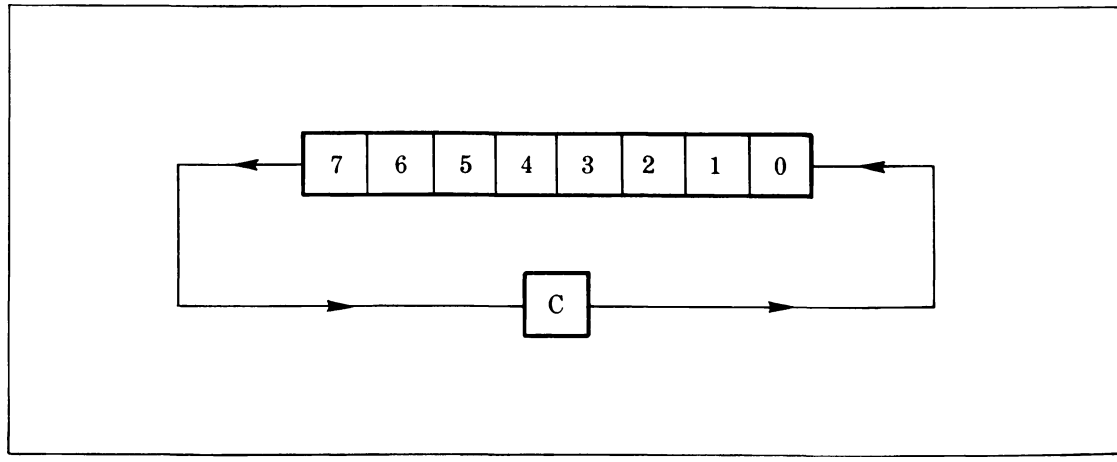

Figure 9-3. The ROL (rotate left) instruction

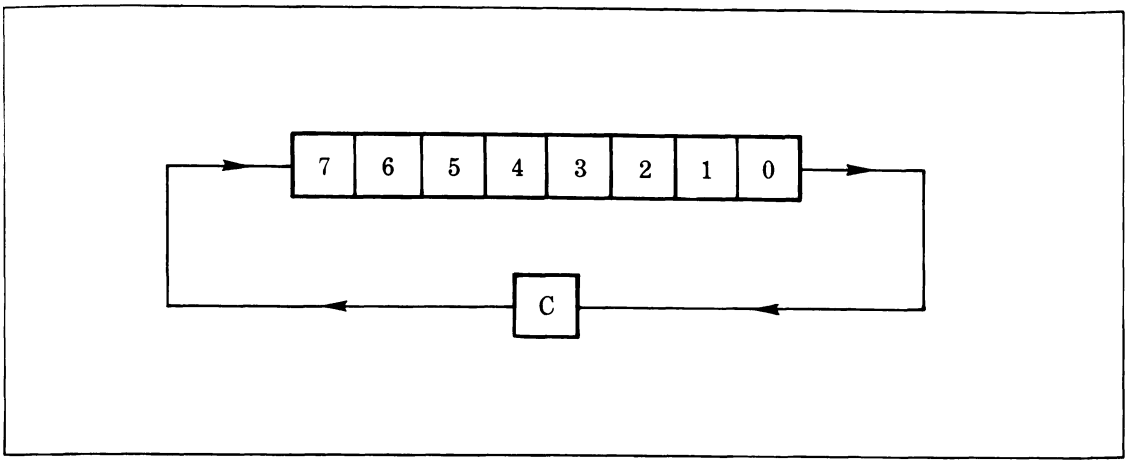

Figure 9-4. The ROR (rotate right) instruction

The Logical Operators

Let's look at four important assembly-language mnemonics called *logical operators.* These instructions are AND (and), ORA (or), EOR (exclusive or), and BIT (bit).

The four 6502B/65C02 logical operators AND, ORA, EOR, and BIT are all used to compare values. They work differently, however, from the comparison operators CMP, CPX, and CPY. The instructions CMP, CPX, and CPY yield very general results. They can determine only whether two values are equal and, if the values aren't equal, which one is larger.

AND, ORA, EOR, and BIT are much more specific instructions. They are used to compare single bits of numbers and thus have many uses in writing assembly-language programs for the Apple IIe/IIc.

Boolean Logic

The four logical operators in assembly language use principles of mathematical science called *Boolean logic.* In Boolean logic, the binary numbers 0 and 1 are used not to express values, but to indicate whether a statement is true or false. If a statement is true, its value in Boolean logic is said to be 1. If the statement is false, its value is said to be 0.

In 6502B/65C02 assembly language, the operator AND has the same meaning that the word "and" has in English.

Table 9-1. Truth Table for AND

0 AND 0 0	0 AND 1 0	1 AND 0 0	1 AND 1 1

If one bit AND another bit have a value of 1 (and are thus "true"), then the AND operator also yields a value of 1. However, if any other condition exists—if one bit is true and the other is false, or if both bits are false—then the AND operator returns a result of 0, or false.

The results of logical operators are often illustrated with diagrams called *truth tables.* Table 9-1 is a truth table for the AND operator.

In 6502B/65C02 assembly language, the AND instruction is often used in an operation called *bit masking.* The purpose of bit masking is to clear or set specific bits of a number. The AND operator can be used, for example, to clear any number of bits by placing a 0 in each bit that is to be cleared:

```
100  LDA #AA ;BINARY 1010 1010
110  AND #F0 ;BINARY 1111 0000
```

If your computer encounters this sequence in a program, it will perform the following AND operation:

```
      1010 1010  (contents of accumulator)
AND   1111 0000
-----------------
      1010 0000  (new value in accumulator)
```

In this example, the AND instruction clears the low nybble of $AA to $0 (with a result of $A0). The same technique would work with any other 8-bit number. No matter what number is being passed through the mask 1111 0000, its lower nybble will always be cleared to $00, and its upper nybble will always emerge from the AND operation unchanged.

The ORA Operator

When the instruction ORA (or) is used to compare a pair of bits, the result of the comparison is 1 (true) if the value of *either* bit is 1. Table 9-2 is the truth table for ORA.

Table 9-2. Truth Table for ORA

0	0	1	1
ORA 0	ORA 1	ORA 0	ORA 1
0	1	1	1

ORA is also used in bit-masking operations. Here is an example of a masking routine using ORA.

```
LDA VALUE
ORA #$0F
STA DEST
```

If the number in VALUE were $22 (binary 0010 0010), the following masking operation would then take place.

```
    0010 0010 (in accumulator)
ORA 0000 1111 (#$0F)
-------------
    0010 1111 (new value in accumulator)
```

The EOR Operator

The instruction EOR (exclusive or) will return a true value (1) if one—and only one—of the bits in the pair being tested is a 1. Table 9-3 is the truth table for the EOR operator.

The EOR instruction is often used for comparing bytes to determine if they are identical. If any bit in two bytes being compared is different, the result of the comparison will be non-0:

Example 1		Example 2
1011 0110		1011 0110
EOR 1011 0110	BUT:	EOR 1011 0111
0000 0000		0000 0001

In Example 1, the bytes being compared are identical, so the result of the comparison is 0. In Example 2, one bit is different, so the result of the comparison is non-0.

The EOR operator is also used to complement values. If an 8-bit value

Table 9-3. Truth Table for EOR

0	0	1	1
EOR 0	EOR 1	EOR 0	EOR 1
0	1	1	0

is EOR'd with $FF, every bit in the value that is a 1 will be complemented to a 0, and every bit that is a 0 will be complemented to a 1:

```
      1110 0101  (in accumulator)
EOR   1111 1111
-------------------
      0001 1010  (new value in accumulator)
```

Still another useful characteristic of the EOR instruction is that when it is performed twice on a number using the same operand, the number will first be changed to another number and then be restored to its original value:

```
      1110 0101  (in accumulator)
EOR   0101 0011
-------------------
      1011 0110  (new value accumulator)
EOR   0101 0011  (same operand as above)
-------------------
      1110 0101  (original value in accumulator restored)
```

This capability of the EOR instruction is often used in high-resolution graphics to place one image over another without destroying the one underneath.

Packing and Unpacking Data in Memory

Now we're ready to discuss the packing and unpacking of data using bit-shifting and bit-testing instructions. First, let's talk about how you can pack data to conserve space in your computer's memory.

To get an idea of how data-packing works, suppose that you had a series of 4-bit values stored in a block of memory in your computer.

These values could be ASCII characters, BCD numbers (more about those later), or any other kind of 4-bit values.

Using the ASL instruction, you can pack two such values into every byte of the block of memory in which they were stored. You can thus store the values in half the memory space that they had occupied in their unpacked form.

Program 9-4 is a routine you can use in a loop to pack each byte of data.

Program 9-4
PACKING DATA USING THE ASL INSTRUCTION

```
 1 *
 2 *PACKDATA
 3 *
 4  ORG $8000
 5 *
 6  NYB1 EQU $FB
 7  NYB2 EQU $FC
 8  PKDBYT EQU $FD
 9 *
10  LDA #$04 ;OR ANY OTHER 4-BIT VALUE
11  STA NYB1
12  LDA #$06 ;OR ANY OTHER 4-BIT VALUE
13  STA NYB2
14 *
15  CLC
16  LDA NYB1
17  ASL A
18  ASL A
19  ASL A
20  ASL A
21  ORA NYB2
22  STA PKDBYT
23  RTS
```

How It Works

The routine in Program 9-4 will load a 4-bit value into the accumulator, shift that value to the high nybble in the accumulator, and then (using the ORA logical operator) place another 4-bit value in the low nybble of the accumulator. The accumulator will thus be packed with two four-bit values—and those two values will then be stored in "PKDBYT", a single 8-byte memory register.

Testing the Results

Type the program into the computer and execute it using your assembler's machine-language monitor. After you've run the program, you can peek into your computer's memory to see what has been done by

using your Apple's built-in monitor. Just call up the machine-language monitor and type the command

```
FB.FD
```

followed by a carriage return. Your computer will then respond with the following line:

```
00FB- 04 06 46
```

This line tells you that the number $04 has been stored in memory address $FB and that the number $06 has been stored in memory address $FC. Both of these values have then been packed into memory address $FC.

You can see from this example how data-packing can increase a computer's effective memory capacity. Suppose you had a long document made up of pure ASCII characters, which can be stored in memory in the form of 4-bit numbers. Packing this text would cut in half the amount of memory the text occupied, since two characters could be stored in each 8-bit register of your computer's memory.

Unpacking Data

It wouldn't do any good to pack data, of course, if it couldn't be unpacked later. Data packed using ASL can be unpacked using the complimentary instruction LSR (logical shift right), together with the logical operator AND. Program 9-5 is a routine that shows how data can be unpacked using the LSR instruction.

Program 9-5
UNPACKING DATA USING THE LSR INSTRUCTION

```
10 *
20 * UNPACKIT
30 *
40 PKDBYT EQU $FB
50 LOWBYT EQU $FC
60 HIBYT EQU $FD
70 *
80  ORG $8000
90 *
100  LDA #255 ;OR ANY OTHER 8-BIT VALUE
110  STA PKDBYT
120  LDA #0 ;CLEAR LOWBYT AND HIBYT
130  STA LOWBYT
140  STA HIBYT
150 *
160  LDA PKDBYT
170  AND #$0F ;BINARY 0000 1111
180  STA LOWBYT
```

```
190   LDA PKDBYT
200   LSR A
210   LSR A
220   LSR A
230   LSR A
240   STA HIBYT
250   RTS
```

The routine illustrated in Program 9-5 works much like Program 9-4—but in reverse. In Program 9-5, the accumulator is loaded with an 8-bit byte into which two 4-bit values have been packed. The upper four bits of this packed byte are then zeroed out using the logical operator AND. Then the lower nybble of the byte is stored in a memory register called LOWBYT.

Next, the accumulator is loaded for a second time with the packed byte. This time the byte is shifted four places to the right using the instruction LSR. This maneuver results in a 4-bit value that is finally stored in a memory register called HIBYT. The packed value in PKDBYT has thus been split, or unpacked, into two 4-bit values—one stored in LOBYT and the other in HIBYT. Each of those 4-bit numbers (which may represent an ASCII character or any other 4-bit value) can now be processed as a separate entity.

The BIT Operator

The BIT operator is an instruction that's a little more complicated than AND, ORA, or EOR.

The BIT instruction is used to determine whether the value stored in a memory register matches a value stored in the accumulator. The instruction can be used only with absolute or zero-page addressing Here are two examples of correct formats for the BIT instruction.

```
BIT $02A7

BIT $FB
```

When the BIT instruction is used in either of these formats, a logical AND operation is performed on the byte being tested. The *opposite* of the result of this operation is then stored in the zero flag of the processor status register. In other words, if any *set* bits in the accumulator happen to match any set bits that are stored *in the same positions* in the value being tested, the Z flag will be cleared. If there are no set bits that match, the Z flag will be set.

Program 9-6 illustrates how the BIT instruction can be used.

Program 9-6
USING THE BIT INSTRUCTION

```
1   LDA #01
2   BIT $02A7
3   BNE MATCH
4   JMP NOGOOD
5   MATCH RTS
```

A check is made to determine whether BIT is set in the value stored in memory register $02A7. If the bit is set, the flag of the P register will be cleared and the program will branch to the line labeled MATCH. If there is no match, the Z flag will be set, and the program will jump to whatever routine has been labeled NOGOOD.

The BIT mnemonic also performs a couple of other functions. When the BIT instruction is used, bits 6 and 7 of the value being tested are always deposited directly into bits 6 and 7 of the P register. That information is very useful because bit 6 and bit 7 are very important flags in the 6502B/65C02 chip's processor status register. Bit 6 is the P register's overflow (V) flag, and bit 7 is its negative (N) flag. Therefore, the BIT instruction can also be used as a quick method of checking either bit 6 or bit 7 of any 8-bit value. If bit 6 of the value being tested is set, the P register's V flag will also be set, and a BVC or BVS instruction can then be used to determine what will happen next in the program. If bit 7 of the tested value is set, then the P register's N flag will be set, and a BPL or BMI instruction can be used to determine the outcome of the routine.

It's important to note that after all of these actions take place, the value in the accumulator and the memory location being tested always remain unchanged. Therefore, if you ever want to perform a logical AND operation without disturbing the value of the accumulator or the memory register you want to check, the BIT mnemonic may be the best instruction to use.

10

Assembly-Language Math

In this chapter, you'll learn how your Apple adds, subtracts, multiplies, and divides. The Apple IIc or Apple IIe can handle many kinds of numbers—including binary, decimal, hexadecimal, signed, and unsigned numbers, as well as binary-coded decimal numbers and floating-point decimal numbers. In this chapter, we're going to look at each of these types of numbers.

To understand how your computer works with numbers, it is essential to have a fairly good understanding of the busiest flag in the 6502B/65C02 microprocessor chip: the carry flag of the 6502B/65C02's processor status register, discussed briefly in Chapter 3.

A Close Look at the Carry Bit

The best way to get a close-up view of how the carry bit works is to examine it through an "electronic microscope"—that is, at the bit level.

HEXADECIMAL	BINARY
$04	0100
+ $01	+ 0001
$05	0101
$08	1000
+ $03	+ 0011
$0B	1011

Figure 10-1. Adding without a carry in the hex and binary systems

Figure 10-1 compares two 4-bit addition problems, one carried out using hexadecimal numbers and the other done with binary numbers. You can see that neither addition operation results in a carry; no carry is generated in either binary or hexadecimal notation.

Figure 10-2 illustrates two more addition problems, using larger (8-bit) numbers. The first of these two problems doesn't generate a carry, but the second one does. Note that the sum in the second problem is a 9-bit number—1 1000 1100 in binary, or 18C in hexadecimal notation.

HEXADECIMAL	BINARY
$8E	1000 1110
+ $23	+ 0010 0011
$B1	1011 0001
$8D	1000 1101
$FF	1111 1111
$018C	(1) 1000 1100

Figure 10-2. Two more addition problems in hex and binary

Program 10-1 is an assembly-language program that will perform the second addition problem in Figure 10-2.

Program 10-1
ADDNCARRY
8-Bit Addition With a Carry

```
 1 *
 2 * ADDNCARRY
 3 *
 4  ORG$8000
 5 *
 6  CLD
 7  CLC
 8  LDA #$8D
 9  ADC #$FF
10  STA $FB
11  RTS
```

Type Program 10-1, assemble it, and then run it using your Apple's machine-language monitor. Then use your monitor to take a look at the contents of memory location $FB: just type the address FB, and you will see a line something like this:

```
00FB-  8C
```

That line will show you that memory address $FB now holds the number $8C. That isn't the sum of the numbers $8D and $FF, but it's close. In hexadecimal arithmetic, the sum of $8D and $FF is $18C—exactly the sum we got, plus a carry. But where is the carry? The missing carry must be tucked away in the carry bit of your computer's processor status register.

Looking for a Bit in a Haystack

Looking for a carry bit inside an Apple may seem like looking for a needle in a haystack, but finding a carry bit really isn't too hard once you know where to look. One way to locate the carry that's missing from the ADDNCARRY program listed above, for example, is by the insertion of a few additional lines into the ADDNCARRY program. Program 10-2 is an expanded version of the program, with those extra lines inserted.

Program 10-2
ADDNCARRY2
Addition With a Carry (Improved Version)

```
 1 *
 2 * ADDNCARRY2
 3 *
 4   ORG $8000
 5 *
 6   CLD
 7   CLC
 8   LDA #$8D
 9   ADC #$FF
10   STA $FB
11   LDA #0
12   ROL A
13   STA $FC
14   RTS
```

In the lines that are added to the ADDNCARRY program in Program 10-2, the accumulator is cleared and the bit-shifting operator ROL is then used to rotate the P register's carry bit into the accumulator. Next the contents of the accumulator are deposited into memory register $FC using an ordinary STA instruction. If this routine works, it means that we've found our missing carry bit.

To see whether the program works, you should now type it, assemble it, and run it. Then you can peek into memory addresses $FB and $FC using your machine-language monitor to see whether the calculation in the ADDNCARRY program resulted in a carry. Here, in Apple II monitor format, is what those two registers should contain:

```
00FB- 8C 01
```

When the line shown appears on your monitor screen, it tells you that memory address $FB once again holds the number $8C (the result of our ADDNCARRY calculation, without its carry) and that the carry resulting from the calculation now resides in memory register $00FC.

A 16-Bit Addition Program

Program 10-3, entitled ADD16, will add two 16-bit numbers. The same principles used in this program can also be used to write programs that will add numbers having 24 bits, 32 bits, and more.

Program 10-3
ADD16
A 16-Bit Addition Program

```
 1 *
 2 *ADD16
 3 *
 4 *THIS PROGRAM ADDS A 16-BIT NUMBER IN $FB AND $FC
 5 *TO A 16-BIT NUMBER IN $FD AND $FE
 6 *AND DEPOSITS THE RESULTS IN $0300 AND $0301
 7 *
 8  ORG $8000
 9 *
10  CLD
11  CLC
12  LDA $FB;REM LOW HALF OF 16-BIT NUMBER IN $FB AND $FC
13  ADC $FD;REM LOW HALF OF 16-BIT NUMBER IN $FD AND $FE
14  STA $0300 ;LOW BYTE OF SUM
15  LDA $FC ;REM HIGH HALF OF 16-BIT NUMBER IN $FB AND $FC
16  ADC $FE ;REM HIGH HALF OF 16-BIT NUMBER IN $FD AND $FE
17  STA $0301 ;HIGH BYTE OF SUM
18  RTS
```

When you look at this program, remember that your Apple computer stores 16-bit numbers with the low-order byte first and the high-order byte second—the reverse of what you might expect. Once you understand this characteristic of all 6502/6502B/65C02-based computers, 16-bit binary addition isn't hard to understand.

In this program, we first clear the carry flag of the P register. Next we add the low byte of a 16-bit number in $FB and $FC to the low byte of a 16-bit number in $FD and $FE.

The result of this half of our calculation is then placed in memory address $0300. If there is a carry, the P register's carry bit will be set automatically.

In the second half of the program, the high byte of the number in $FB and $FC is added to the high byte of the number in $FD and $FE. If the P register's carry bit has been set as a result of the preceding addition operation, a carry will also be added to the high bytes of the two numbers. If the carry bit is clear, there will be no carry.

When this half of our calculation has been completed, its result is deposited into memory address $0301. Finally, the results of our completed addition problem are stored (low byte first) in memory addresses $0300 and $0301.

16-Bit Subtraction

Program 10-4 illustrates a 16-bit subtraction program.

Program 10-4
SUB16
A 16-Bit Subtraction Program

```
 1 *
 2 *SUB16
 3 *
 4 *THIS PROGRAM SUBTRACTS A 16-BIT NUMBER IN $FB AND $FC
 5 *FROM A 16-BIT NUMBER IN $FD AND $FE
 6 *AND DEPOSITS THE RESULTS IN $0300 AND $0301
 7 *
 8  ORG $8000
 9 *
10  CLD
11  SEC ;REM SET CARRY
12  LDA $FD;REM LOW HALF OF 16-BIT NUMBER IN $FD AND $FE
13  SBC $FB;REM LOW HALF OF 16-BIT NUMBER IN $FB AND $FC
14  STA $0300 ;LOW BYTE OF THE ANSWER
15  LDA $FE ;REM HIGH HALF OF 16-BIT NUMBER IN $FD AND $FE
16  SBC $FC ;REM HIGH HALF OF 16-BIT NUMBER IN $FB AND $FC
17  STA $0301 ;HIGH BYTE OF THE ANSWER
18  RTS
```

Since subtraction is the exact opposite of addition, the carry flag is set, rather than cleared, before a subtraction operation is performed in 6502B/65C02 binary arithmetic. In subtraction, the carry flag is treated as a borrow, not a carry, and it must therefore be set—rather than cleared—so that if a borrow is necessary, there will be a value to borrow from.

After the carry bit is set, a 6502B/65C02 subtraction problem is quite straightforward. In our sample problem, the 16-bit number in $FB and $FC is subtracted, low byte first, from the 16-bit number in $FD and $FE. The result of the problem—including, if necessary, a borrow from the high byte—is then stored in memory addresses $0300 and $0301, low byte first.

Binary Multiplication

Binary numbers are multiplied the same way as decimal numbers. Figure 10-3 is an example of binary multiplication. Unfortunately, there are no 6502B/65C02 assembly-language instructions for multiplication or division. To multiply a pair of numbers using 6502B/65C02

```
   0110      ($06)
 × 0101      ($05)
-------
   0110
  0000
 0110
0000
-------
0011110      ($1E)
```

Figure 10-3. An example of binary multiplication

assembly language, you have to perform a series of addition operations. To divide numbers, you have to perform subtraction sequences.

If you look closely at the multiplication problem in Figure 10-3, you will see that it isn't difficult to split a multiplication problem into a series of addition problems. In the example, the binary number 0110 is first multiplied by 1. Then the result of this operation—also 0110, of course—is written down.

Next, 0110 is multiplied by 0. The result of that operation—a string of 0's—is also shifted one space to the left and written down. Then 0110 is multiplied by 1 again and the result is again shifted left and written down. Finally, another multiplication by 0 results in another string of 0's, which are also shifted left and noted.

Finally, all the partial products of our problem are added up, just as they would be in a conventional multiplication problem. The result of this addition, as you can see, is the final product $1E.

This multiplication technique works well, but it's really an arbitrary method. Why, for example, did we shift each partial product in this problem to the left before writing it down? We could have accomplished the same result by shifting the partial product above it *to the right* before adding.

In 6502B/65C02 multiplication, that's exactly what's often done; instead of shifting each partial product to the left before storing it in memory, many 6502B/65C02 multiplication algorithms shift the preceding partial product to the right before adding it to the new one.

Program 10-5 is a program that shows you this method.

Program 10-5
MULT16
A 16-Bit Multiplication Program

```
 1 *
 2 *MULT16
 3 *
 4 MPR EQU $FD ;MULTIPLIER
 5 MPD1 EQU $FE ;MULTIPLICAND
 6 MPD2 EQU $0300 ;NEW  MULTIPLICAND AFTER 8 SHIFTS
 7 PRODL EQU $0301 ;LOW BYTE OF PRODUCT
 8 PRODH EQU $0302 ;HIGH BYTE OF PRODUCT
 9 *
10  ORG $8000
11 *
12 *THESE ARE THE NUMBERS WE WILL MULTIPLY
13 *
14  LDA #250
15 STA MPR
16  LDA #2
17  STA MPD1
18 *
19 MULT CLD
20  CLC
21  LDA #0 ;CLEAR ACCUMULATOR
22  STA MPD2 ;CLEAR ADDRESS FOR SHIFTED MULTIPLICAND
23  STA PRODH ;CLEAR HIGH BYTE OF PRODUCT ADDRESS
24  STA PRODL ;CLEAR LOW BYTE OF PRODUCT ADDRESS
25  LDX #8 ;WE WILL USE THE X REGISTER AS A COUNTER
26 LOOP LSR MPR ;SHIFT MULTIPLIER RIGHT; LSB DROPS INTO
     CARRY BIT
27  BCC NOADD ;TEST CARRY BIT; IF ZERO, BRANCH TO NOADD
28  LDA PRODH
29  CLC
30  ADC MPD1 ;ADD HIGH BYTE OF PRODUCT TO MULTIPLICAND
31  STA PRODH ;RESULT IS NEW HIGH BYTE OF PRODUCT
32  LDA PRODL ;LOAD ACCUMULATOR WITH LOW BYTE OF PRODUCT
33  ADC MPD2 ;ADD HIGH PART OF MULTIPLICAND
34  STA PRODL ;RESULT IS NEW LOW BYTE OF PRODUCT
35 NOADD ASL MPD1 ;SHIFT MULTIPLICAND LEFT; BIT 7 DROPS
     INTO CARRY
36  ROL MPD2 ;ROTATE CARRY BIT INTO BIT 7 OF MPD1
37  DEX ;DECREMENT CONTENTS OF X REGISTER
38  BNE LOOP ;IF RESULT ISN'T ZERO, JUMP BACK TO LOOP
39  RTS
```

As you can see, 8-bit binary multiplication isn't exactly a snap. There's a lot of left and right bit-shifting involved, and it's hard to keep track of. In Program 10-5, the most difficult manipulation to follow is probably the one involving the multiplicand (MPD1 and MPD2). The multiplicand is only an 8-bit value, but it's treated as a 16-bit value because it keeps getting shifted to the left; and while it is moving, it takes a 16-bit address (actually two 8-bit addresses) to hold it.

To see for yourself how the program works, type it on your keyboard and assemble it. Then use the G command of your monitor to execute it. While you're still in the monitor mode, you can look at the contents of memory addresses $0301 and $0302. These two registers should now hold the number $01F4 (low byte first). That's the hex equivalent of the decimal number 500, which is, of course, the product of the decimal numbers 2 and 250—the problem that our program was supposed to multiply.

An Improved Multiplication Program

Although Program 10-5 works well, it isn't the only 16-bit multiplication program available; in fact, it isn't even a very good one. There are many algorithms for binary multiplication, and some of them are shorter and more efficient than the one we just executed. Program 10-6, for example, is considerably shorter and therefore is both more memory-efficient and faster-running. One of the best features of this improved multiplication program is that it uses the 6502B/65C02's accumulator, rather than a memory address, for temporary storage of the problem's results.

Program 10-6
MULT16B
An Improved 16-Bit Multiplication Program

```
 1 *
 2 * MULT16B
 3 * (AN IMPROVED 16-BIT MULTIPLICATION PROGRAM)
 4 *
 5 PRODL EQU $FD
 6 PRODH EQU $FE
 7 MPR EQU $0300
 8 MPD EQU $0301
 9 *
10  ORG $8000
11 *
12 VALUES LDA #10
13  STA MPR
14  LDA #10
15  STA MPD
16 *
17  LDA #0
18  STA PRODH
19  LDX #8
20 LOOP LSR MPR
21  BCC NOADD
22  CLC
23  ADC MPD
```

```
24 NOADD ROR A
25   ROR PRODH
26   DEX
27   BNE LOOP
28   STA PRODL
29   RTS
```

You may want to test out the improved multiplication capabilities of Program 10-6 the same way that we tested Program 10-5: execute it using your machine-language monitor, and then use your monitor to look at the results.

You can experiment with these two multiplication problems as much as you like, trying out different values and seeing how those values are processed in each program.

However, the best way to become intimately familiar with how binary multiplication works is to do a few problems by hand, using pencil and paper. If you work enough binary multiplication problems on paper, you'll soon understand the principles of 6502B/65C02 multiplication.

Multiprecision Binary Division

Earlier in this chapter, we demonstrated that subtraction is reverse addition. Similarly, division is nothing but reverse multiplication. We also know that the 6502B/65C02 chip, which has no specific instructions for multiplying numbers, also has no instructions for dividing numbers.

Still, it is possible to perform division—even multiple-precision long division—using instructions that are available to the 6502B/65C02 microprocessor. The 6502B/65C02 chip can multiply numbers if the multiplication problems presented to it are broken down into sequences of addition problems. In the same way, the 6502B/65C02 chip can divide numbers, as long as the division problems presented to it are broken down into sequences of subtraction problems.

Program 10-7, for example, is a routine designed to divide one number by another number by breaking the division process down into a series of subtraction routines. During the execution of Program 10-7, the high part of the dividend will be stored in the accumulator and the low part of the dividend will be stored in a variable called DVDL.

The program contains a lot of shifting, rotating, subtracting, and decrementing of the X register. When the main body of the program ends, the quotient will be stored in a variable labeled QUOT, and the quotient's remainder will be in the accumulator. Then, in line 38, the

remainder will be moved out of the accumulator and into a variable called RMDR. An RTS instruction will end the program.

Program 10-7
DIV8.16
A Binary Long-Division Program

```
 1 *
 2 * DIV8.16
 3 *
 4  ORG $8000
 5 *
 6 DVDH EQU $FD ;LOW PART OF DIVIDEND
 7 DVDL EQU $FE ;HIGH PART OF DIVIDEND
 8 QUOT EQU $0300 ;QUOTIENT
 9 DIVS EQU $0301 ;DIVISOR
10 RMDR EQU $0302 ;REMAINDER
11 *
12  LDA #$1C ;JUST A SAMPLE VALUE
13  STA DVDL
14  LDA #$02 ;THE DIVIDEND IS NOW $021C
15  STA DVDH
16  LDA #$05 ;ANOTHER SAMPLE VALUE
17  STA DIVS ;WE'RE DIVIDING BY 5
18 *
19  LDA DVDH ;ACCUMULATOR WILL HOLD DVDH
20  LDX #08 ;FOR AN 8-BIT DIVISOR
21  SEC
22  SBC DIVS
23 DLOOP PHP ;SAVE P REGISTER (ROL & ASL AFFECT IT)
24  ROL QUOT
25  ASL DVDL
26  ROL A
27  PLP ;RESTORE P REGISTER
28  BCC ADDIT
29  SBC DIVS
30  JMP NEXT
31 ADDIT ADC DIVS
32 NEXT DEX
33  BNE DLOOP
34  BCS FINI
35  ADC DIVS
36  CLC
37 FINI ROL QUOT
38  STA RMDR
39  RTS
```

Running the Program

Program 10-7 can be used to divide any unsigned 16-bit number by any unsigned 8-bit number. As written, it divides the hexadecimal number $021C (540 in decimal notation) by 5. The quotient is stored in memory

register $0300, and the remainder, if any, is stored in memory register $0302.

Type the program, assemble it, and run it, and then use your monitor to inspect the contents of memory addresses $0300 and $0302. Address $0300 should now hold the hexadecimal number $6C (108 in decimal notation), and there should be a 0 in address $0302, since the quotient of 540 divided by 5 is 108, with no remainder.

Not the Ultimate Division Program

As you can see, it's even more difficult to write a division routine for an Apple than it is to write an Apple multiplication program. In fact, writing just about any kind of multiple-precision math program for an 8-bit computer is usually more trouble than it's worth. When you have to write a program in which just a few calculations have to be made, you can sometimes use short, simple routines such as the ones presented in this chapter.

However, assembly language is usually not the best language to use for writing long, complex programs that contain a lot of multiple-precision math. If you ever have to write such a program, you might find it worthwhile to write part of the program in assembly language and the other part—the part with all the math—in BASIC. That way, you can take advantage of the excellent floating-point math package that's built into the BASIC interpreter in your Apple. If you can't do that, it might still be better to write the program in BASIC, Pascal, COBOL, Logo, or almost any other high-level programming language than in assembly language.

If, despite this warning, you still want to write complex math routines in 6502B/65C02 assembly language, there are a few books that may provide you with some help. One text that contains a number of fairly complex math routines you can type is *6502 Assembly Language Subroutines*, written by Lance A. Leventhal and Winthrop Saville and published by Osborne/McGraw-Hill. There are also quite a few type-and-run math routines in some of the other manuals and texts listed in this book's bibliography.

Signed Numbers

To represent a signed number in binary arithmetic, all you have to do is let the far-left bit (bit 7) represent a positive or negative sign. In signed binary arithmetic, if bit 7 of a number is 0, the number is positive; if bit 7 is a 1, the number is negative.

Obviously, if you use one bit of an 8-bit number to represent its sign, you no longer have an 8-bit number. What you then have is a 7-bit number—or, if you want to express it another way, you have a signed number that can represent values from −128 to +127 instead of from 0 to 255.

Signed Binary Addition

It takes more than the redesignation of a bit to turn unsigned binary arithmetic operations into signed binary arithmetic operations. Consider, for example, what we would get if we tried to add the numbers +5 and −4 by doing nothing more than using bit 7 as a sign.

```
  0000 0101 (+5)
+ 1000 0100 (−4)
-----------------
  1000 1001 (−9)
```

The answer is wrong. The answer should be +1.

The reason we got the wrong answer is that we tried to solve the problem without using a concept that is fundamental to the use of signed binary arithmetic: the concept of *complements.*

Complements are used in signed binary arithmetic because negative numbers are complements of positive numbers, and complements of numbers are very easy to calculate in binary arithmetic. In binary math, the complement of a 0 is a 1, and the complement of a 1 is a 0.

One's and Two's Complement Addition

It is reasonable to assume that the negative complement of a positive binary number could be arrived at by complementing each 0 in the number to a 1 and each 1 to a 0 (except for bit 7, of course, which must be used to represent the number's sign). This technique of calculating the complement of a number by flipping its bits from 0 to 1 and from 1 to 0 has a name in assembly-language circles. It's called *one's complement.*

To see if the one's complement technique works, let's try using it to add two signed numbers, say +8 and −5.

```
  0000 1000 (+8)
+ 1111 1010 (−5)    (one's complement)
-----------------
  0000 0010 (+2)    (plus carry)
```

That's wrong, too! The answer should be +3. That takes us back to the drawing board. One's complement arithmetic doesn't work.

Fortunately, there's another technique that does work. It's called

two's complement. To use this technique, first calculate the one's complement of a positive number. Then simply add one. That will give you the two's complement—the true complement—of the number. Then you can use the conventional rules of binary math on signed numbers.

Here are two examples of two's complement addition.

```
  0000 0101 (+5)
+ 1111 1000 (−8)   (two's complement)
-----------------
  1111 1101 (−3)

  1111 1011 (−5)   (two's complement)
+ 0000 1000 (+8)
-----------------
  0000 0011 (+3)   (plus carry)
```

A Few Examples

It isn't easy to explain why the two's complement method works, but when you've worked with signed binary numbers for a while, you begin to get a feel for them. It helps to remember that the highest bit of a binary number is always interpreted as a sign in two's complement notation, so a binary number with the highest bit set is always interpreted as a negative number. Therefore, the hexadecimal number $7F, which equates to the decimal number 127, is the highest positive number that can be expressed in 8-bit two's complement notation.

If you increment the hex number $7F, you'll see why this is true. In binary notation, $7F is written %0111 1111 (a binary number in which the high bit is not set). If you increment $7F, though, you'll get $80, or %1000 0000, a number that has its high bit set and will therefore be interpreted in 8-bit two's complement notation as −128, not +128. Thus, the largest positive number that can be expressed in 8-bit two's complement notation is 127.

Now let's take a look at some negative binary numbers. In two's complement arithmetic, negative numbers start at −1 and work backward, just as negative numbers do in ordinary arithmetic. In conventional arithmetic, though, there's no such number as −0, so when you decrement a 0, what you get is not minus 0, but −1. If you decrement −1 you get −2, which may look like a larger number but is really a smaller one. (If, for example, you decrement −2, you will get −3.)

Two's complement arithmetic works in a similar fashion. If you decrement 0 using 8-bit two's complement arithmetic, you'll get $FF, which equates to −1 in decimal notation. Decrement $FF in two's com-

plement, and you get $FE, the 8-bit signed-binary equivalent of −2. The decimal number −3 is written $FD in 8-bit two's complement notation, and the decimal number −4 is written $FC. Keep working backward, and you'll eventually discover that the smallest negative number that can be expressed in 8-bit two's complement notation is the hexadecimal number $80, which equates to −128 in conventional decimal numbers.

If you ever start writing programs that make use of signed binary numbers, you'll actually need instructions much more detailed than the ones provided here. In this chapter, my intention is merely to introduce you to some of the techniques that are used in programs containing signed binary numbers.

Using the Overflow Flag

In signed binary arithmetic, when you carry out calculations using signed numbers, the overflow (V) flag of the processor status register—not the carry flag of the processor status register—is used to carry numbers from one byte to another. The reason is that the carry flag of the P register is set when there's an overflow from bit 7 of a binary number; but when the number is a signed number, bit 7 is the *sign* bit, not part of the number. Therefore, the carry flag cannot be used to detect a carry in an operation that involves signed numbers.

You can, however, use the overflow bit of the processor status register. The overflow bit is set when there is an overflow from bit 6, rather than bit 7. Thus it can be used as a carry bit in arithmetic operations on signed numbers.

As you may recall from high school algebra, the rules of adding, subtracting, multiplying, and dividing signed numbers are rather complex; they vary according to the signs of the numbers that are involved in the calculations and according to what kinds of calculations are being performed. It should come as no surprise, then, that the rules for using the overflow flag in calculations involving signed binary numbers are also a little complicated. You can find them in textbooks on advanced assembly-language programming, but they are well beyond the scope of this chapter.

BCD (Binary-Coded Decimal) Numbers

In BCD notation, the digits 0 through 9 are expressed just as they are in conventional binary notation, but the hexadecimal digits $A through $F (%1010 through %1111 in binary) are not used. Long numbers, therefore,

must be represented differently in BCD notation than they are in conventional binary notation.

The decimal number 1258, for example, would be written in BCD notation as

1	2	5	8
0000 0001	0000 0002	0000 0101	0000 1000

In conventional binary notation, the same number would be written as

$0	$4	$E	$A
0000	0100	1110	1010

which equates to $04EA, or the hexadecimal equivalent of 1258.

BCD notation is often used in bookkeeping and accounting programs because BCD arithmetic, unlike straight binary arithmetic, always yields results that are 100% accurate. BCD numbers are also sometimes used when it is desirable to display them instantly, digit by digit, as they are being used (for example, when numbers are being used for onscreen scorekeeping in a game program).

The main disadvantage of BCD numbers is that they tend to be difficult to work with. When you use BCD numbers, you must be extremely careful with signs, decimal points, and carry operations. You must also decide whether you want to use an 8-bit byte for each digit (which wastes memory, since it really only takes four bits to encode a BCD digit) or whether to pack two digits into each byte, which saves memory but consumes processing time.

Floating-Point Numbers

Floating-point numbers, as you may know, are numbers that enable computers and calculators to perform mathematical calculations on decimal values and fractions. Most calculators use floating-point numbers to perform mathematical calculations, and so does the BASIC interpreter that's built into your Apple. In the Apple's floating-point package, numbers are broken into three parts—an exponent, a mantissa, and a sign. These parts are stored in a block of memory called a floating-point accumulator, which resides in memory registers $61 to $66. There's another floating-point accumulator in memory registers $69 through $6E.

Unfortunately for assembly-language programmers, it's extremely difficult to understand how floating-point routines work, and it's even

more difficult to write them. It's nice to know, then, that there's a very good floating-point package built right into your Apple. To use the Apple floating-point package in an assembly-language program, all you have to do is write the program partly in assembly language and partly in BASIC and then intermix the BASIC and assembly-language sections of your program using the USR(X) function that was explained in Chapter 5. When you write a program in this fashion, you can use assembly language for the portions of the program that require high speed or high performance—and for portions of the program that require high-precision math, you can use BASIC to access your computer's built-in floating-point package. (You can also access the built-in floating-point routines from assembly-language programs, provided you know how to convert a floating-point number to a binary number, and vice versa, but that process is a bit beyond the scope of this chapter.)

Since the floating-point package exists and is easy to use, you may never need to know most of the programming techniques described in this chapter. An understanding of how they work, however, will definitely make you a better Apple assembly-language programmer.

Furthermore, you have to know at least the fundamentals of 6502B/65C02 arithmetic if you want to become a good Apple assembly-language programmer. After all, mathematical processing, in one form or another, really is what computer programming is all about. Since your Apple adds, subtracts, compares, and bit-shifts its way through every program it processes, it would be difficult to write Apple programs for any length of time without knowing at least something about binary addition, subtraction, multiplication, and division. So even though you may never have to write an assembly-language routine that will perform long division on signed numbers, accurate to 17 decimal places, chances are pretty good that you'll eventually have to use some arithmetic operations in at least some of the programs that you write. So before you move on to the next chapter, make sure that you understand this one fairly well. You'll be glad you did.

11

Memory Magic

The engineers who designed the Apple IIc and the Apple IIe accomplished quite a feat. They crammed more than 128K of memory into a pair of 64K machines. The secret behind this remarkable operation can be summed up in one hyphenated word: bank-switching.

Bank-switching, you will recall from Chapter 3, is based on the concept that two blocks of memory can share the same address space, as long as they do not try to occupy that space at the same time. This is the same concept families use when they share a vacation condominium but use it at different times.When bank-switching capabilities are built into a computer, various blocks of RAM and ROM are assigned identical addresses, and special switching facilities are then provided so that these blocks of memory can be switched into and out of the address space they share. Program designers can then move specific blocks of memory in and out of the address space that is available, in accordance with the changing needs of their programs.

In Apple computers, bank-switching is usually accomplished with the aid of special electronic circuits called soft switches. A soft switch, as its name implies, is a microcomputer circuit that can be used just

like a switch. When a computer is designed to use bank-switching for memory management, soft switches are built into the machine and can be used in programs to determine which blocks of bank-switched memory will occupy specific addresses. Soft switches can also be used to protect certain blocks of memory by making it possible to read those blocks of memory but impossible to write to them.

Memory: Some Basic Concepts

In order to understand how bank-switching works, it helps to have in mind a few basic concepts of memory management. Here are some of the principles of microcomputer memory mapping.

The Difference a "K" Makes

The letter K, an abbreviation of the word *kilobyte*, stands for the decimal number 1024. The decimal number 1024 equates to the hexadecimal number $400. A kilobyte derives its name from the fact that it is very close to the decimal value 1000, which has long been abbreviated *kilo* (and sometimes "K"). A 64K computer, then, is one that can address 65,536 (1024 × 64) bytes of memory. A 128K computer has twice that memory capacity: 131,072 (1024 × 128) bytes of memory.

The "Page" Concept

In the world of 8-bit microcomputers, a block of 256 memory addresses (usually numbered from 0 to 255) is called a *page*. A page is a very convenient unit of measurement to use when you are dealing with microcomputers, since the decimal numbers 0 to 255 equate to the hexadecimal numbers $00 to $FF and can therefore be expressed as the first two digits of a four-digit hexadecimal address. In an Apple IIc or IIe computer, Page Zero consists of memory addresses $00 through $FF, Page One includes memory addresses $0100 through $01FF, and so on. The highest memory page in an Apple II-series computer is Page $FF, which includes memory addresses $FF00 through $FFFF.

In Apple graphics (and text), the word page can also refer to a segment of memory that is used for a screen display. This is a completely different meaning of the word page from the one we're considering now. The context in which the word is used will usually make its meaning clear.

Bank-Switching

Figure 11-1 is a memory map that shows how bank-switching works in the Apple IIc. An Apple IIe equipped with an expanded 80-column text card uses exactly the same memory map. In Figure 11-1, there are two

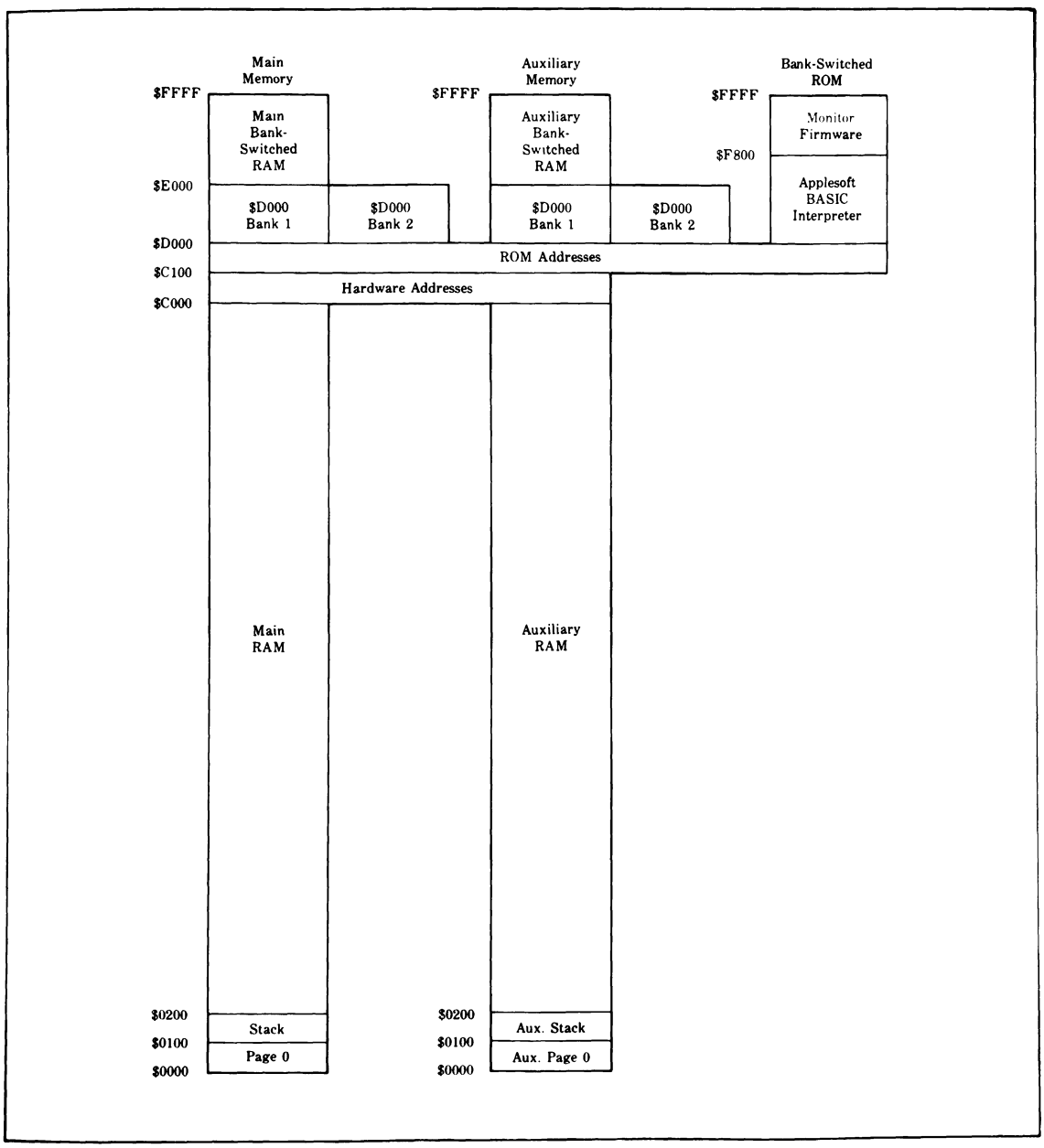

Figure 11-1. Bank-switching the Apple IIc and the fully expanded Apple IIe

long ribbons, one labeled "Main RAM" and the other labeled "Auxiliary RAM." These two ribbons represent the 128 kilobytes of memory that are built into the Apple IIc and are also available in a fully expanded Apple IIe.

Examine the memory addresses that are printed alongside the "Main Memory" and "Auxiliary Memory" columns in Figure 11-1, and you will see that both columns use the same series of addresses: the 64K of address space that extends from $0000 to $FFFF (or from 0 to 65,535 in decimal notation). To the right of these two columns there is a shorter column, labeled "Bank-Switched ROM," that extends from memory address $C100 through memory address $FFFF.

With the help of the soft switches built into the Apple IIc and the Apple IIe, it is possible to write assembly-language programs that use both the main and auxiliary banks that are available in an Apple IIc or an expanded Apple IIe. With the help of these soft switches, almost all of the 128K of memory space on the Apple IIc/IIe memory map can be used as RAM. Alternatively, in programs that make use of Applesoft BASIC or the built-in Apple IIc/IIe monitor, ROM can be switched into the address space ranging from $D000 to $FFFF, and both Applesoft BASIC and the built-in Apple monitor can then be used in programs.

Main and Auxiliary RAM

The 48K block of memory that extends from $0000 to $BFFF can be used as either main RAM or auxiliary RAM. Furthermore, it is possible to write assembly-language programs that can read from one of these data banks and write to the other. Therefore, a program that is stored in one memory space can read and write data that is stored in the other.

There are two soft switches that can be used to switch back and forth between the main and auxiliary memory banks in assembly-language programs. One of these switches, labeled RAMRD, is used to select main or auxiliary memory for reading. The other switch, labeled RAMWRT, selects main or auxiliary memory for writing.

The RAMRD switch occupies three memory registers: $C002, $C003, and $C013. The RAMWRT switch also occupies three memory registers: $C004, $C005, and $C014.

Storing any value in the RAMRD soft switch at $C002 turns the RAMRD switch *off* and selects main memory for *reading*. Storing any value in the RAMRD soft switch at $C003 turns the switch *on* and selects auxiliary memory for *reading*.

Placing any value in the RAMWRT soft switch at $C004 turns the RAMWRT switch *off* and selects main memory for *writing.* Placing any value in the RAMWRT soft switch at $C005 turns the switch *on* and selects auxiliary memory for *writing.*

By reading the values of memory registers $C013 and $C014, a programmer can check on the status of either the RAMRD switch or the RAMWRT switch. If register $C013 has its high bit set, then the RAMRD switch is on and auxiliary memory has been selected for reading. If register $C014 has its high bit set, then the RAMWRT switch is on and auxiliary memory has been selected for writing.

The functions of the RAMRD and RAMWRT switches—along with the functions of several other soft switches—are listed in Table 11-3.

Main and Auxiliary Bank-Switched Memory

Another block of memory that can be controlled by bank-switching is the segment that extends from $D000 to $FFFF. By setting and clearing soft switches in various combinations, a programmer can use this memory bank as main memory, auxiliary memory, or ROM. Furthermore, the block of memory that extends from $D000 to $DFFF can be subdivided into a pair of memory blocks. These blocks are labeled "$D000 Bank 1" and "$D000 Bank 2" in Figure 11-1. The $D000-to-$DFFF memory block can thus be used in five different ways: as Main Memory Bank 1, Main Memory Bank 2, Auxiliary Memory Bank 1, Auxiliary Memory Bank 2, and ROM.

The entire block of memory that extends from $D000 to $FFFF can be controlled with a series of soft switches located at memory addresses $C080 to $C08F. With the help of these switches, it is possible to write assembly-language programs that will read from the ROM that extends from $D000 to $FFFF while writing to the RAM (either the main RAM or the auxiliary RAM) that resides in the same block of memory. The same series of soft switches also controls whether $D000 Bank 1 or $D000 Bank 2 is selected for reading and/or writing.

Table 11-1 illustrates how the soft switches at $C080 to $C08F are used. To use the soft switches listed in Table 11-1, it is not necessary to write any values into them; as strange as it may seem, a programmer has only to *read* the switch to turn it on. When the switch is read, nothing has to be done with the value that is obtained, since the very act of reading the switch sets it.

Table 11-1. Bank-Select Soft Switches

Address of Switch	Double-Read Operation?	Function
$C080 Bank 2.	N	Read from RAM; no writing; use $D000
$C081 Bank 2.	Y	Read from ROM; write to RAM; use $D000
$C082 Bank 2.	N	Read from ROM; no writing; use $D000
$C083 Bank 2.	Y	Read from and write to RAM; use $D000
$C088 Bank 1.	N	Read from RAM; no writing; use $D000
$C089 Bank 1.	Y	Read from ROM; write to RAM; use $D000
$C08A Bank 1.	N	Read from ROM; no writing; use $D000
$C08B Bank 1.	Y	Read from and write to RAM; use $D000

Here is an example of how the switch at $C080 can be set using a read operation:

```
LDA $C080
```

The mnemonic LDA is, of course, a "read" instruction. Under ordinary conditions, the instruction LDA has no effect at all on the contents of the memory register that follows it. In the Apple IIc/IIe, however, memory address $C080 is one of the soft switches that *are* affected by "read" instructions such as LDA. Thus, when the instruction

```
LDA $C080
```

is encountered in an Apple IIc/IIe assembly-language program, the soft switch at memory address $C080 will be set, and the instructions listed for $C080 in Table 11-1 will be followed.

Double-Read Operations

When a switch is so sensitive that it can be set using what is ordinarily a read-only command, unfortunate—even disastrous—accidents can occur. So the engineers who designed the Apple IIc and the Apple IIe

built extra protection into some of the more critical soft switches at memory locations $C080 through $C08F. To set these switches, it is necessary to read them twice—in other words, to carry out two consecutive "read" operations. For example, to set the switch at memory location $C083, you have to carry out a pair of operations:

```
LDA $C083
LDA $C083
```

Other Soft Switches

In addition to the soft switches listed in Table 11-1, two other switches are sometimes used in Apple IIc/IIe bank-switching operations.

- A soft switch labeled RDBNK2, and located at memory address $C011, can be read to determine whether $D000 Bank 1 or $D000 Bank 2 is in use. If bit 7 of RDBNK2 is set, Bank 2 is being used. If bit 7 of RDBNK2 is clear, Bank 1 is being used.
- A soft switch labeled RDLCRAM, located at memory address $C012, can be read to determine whether RAM or ROM is being read. If bit 7 of RDLCRAM is set, RAM is being read. If bit 7 of RDLCRAM is clear, ROM is being read.

Two Stacks and Two Zero Pages

If you look at the bottom of Figure 11-1, in memory locations $0000 to $01FF, you will see that there are two stacks and two Zero Pages—one pair residing in main memory and the other situated in auxiliary memory. When the soft switches RAMRD and RAMWRT are used to determine whether main or auxiliary memory is to be used in a program, the stack and the Zero Page that reside in the appropriate memory bank are automatically selected. However, if you don't want to accept this automatic selection process, there is a soft switch that you can use to make a manual selection of the stack and Zero Page that you want to use in a program. This soft switch, labeled ALTZP, is made up of three memory registers: $C008, $C009, and $C016.

Storing any value in $C008 turns the ALTZP switch off and selects the main-memory stack and Zero Page for both reading and writing. Storing any value in $C009 turns the switch on and selects the auxiliary-memory stack and Zero Page for both reading and writing.

The soft switch at memory register $C016 can be used to check the status of the ALTZP switch. If the high bit of $C016 is set, then ALTZP is on and the alternate-memory stack and Zero Page are selected. If the high bit of $C016 is clear, then ALTZP is off and the main-memory

stack and Zero Page are selected.

If you use the ALTZP soft switch, you have to be careful. When the ALTZP switch is used to change the location of Page Zero, it also changes the location of your computer's hardware stack—and that change, if sloppily carried out, can wreak havoc on the program you are running. Another potential problem is that the ALTZP switch can be overridden by the RAMRD and the RAMWRT switches, since these two switches automatically select which stack and Zero Page will be used in a program. Still another possible source of trouble is your Apple's interrupt handler, which can turn off the ALTZP switch without warning.

Before you can use the ALTZP switch safely, then, you need a thorough understanding of interrupts, stack operations, the RAMRD and RAMWRT switches, and a few other techniques that are touched upon only briefly in this book. If you're a beginning-level assembly-language programmer, you should consider not using the ALTZP switch very often right now.

Non-Switchable Memory

The blocks of memory that extend from $C000 to $CFFF are reserved for use by the Apple IIc/IIe operating system and are not subject to control by bank-switching. The contents of these blocks of memory will be covered in greater detail in the following section.

A More Detailed Memory Map

The first section of this chapter was a very brief introduction to Apple IIc/Apple IIe memory management. Now we will explore the same topic in a little more detail by using the map that appears in Figure 11-2.

In exploring the memory map in Figure 11-2, we'll start at the bottom—or, in computer jargon, in *low memory*—and work our way up to the top of the Apple IIc/Apple IIe memory map. Along the way, we'll pause at a number of locations and take a close look at the contents of some of the more important segments of your Apple's memory.

Addresses $00 to $FF (Page Zero)

As pointed out earlier in this chapter, memory addresses $00 to $FF are known collectively as the Zero Page, or Page Zero. When Page Zero is used in a computer program, it can speed up the operation of the program. When a machine-language operand can be expressed as a Zero-

Page address, only one byte of memory is required, rather than the two bytes of memory required by a non-Zero-Page address. The program can thus be written in fewer bytes and will therefore run faster.

More important, there are some addressing modes—specifically, indirect addressing modes—that *require* the use of Zero-Page addresses.

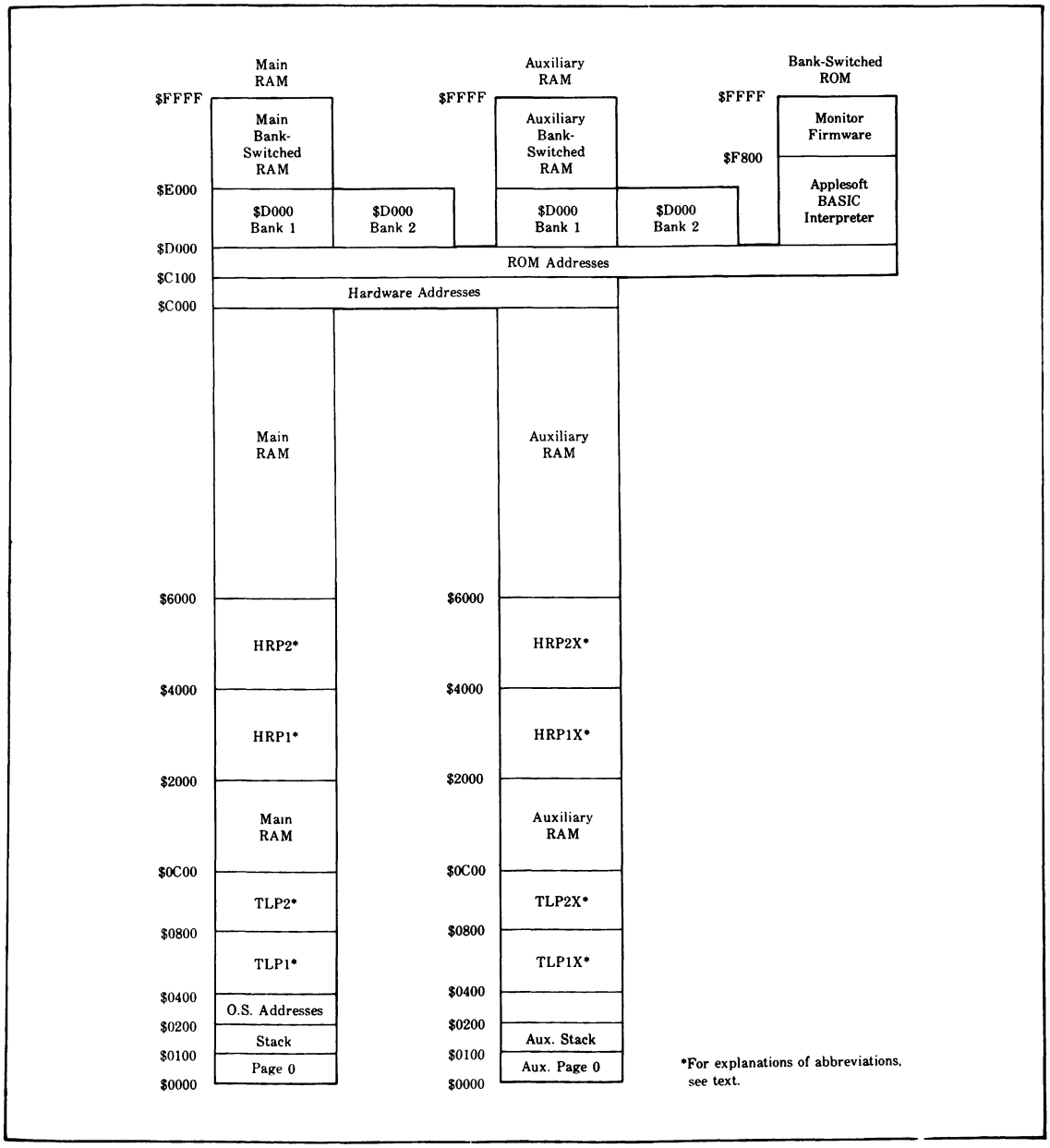

Figure 11-2. Memory map of the Apple IIc and the fully expanded Apple IIe

Since indirect addressing modes are very powerful, most assembly-language programs make extensive use of Page Zero.

Unfortunately, it is not always easy to find free space on Page Zero for user-written programs. That's because Page Zero is so useful that the engineers who designed your computer have claimed most of it for themselves.

- Applesoft BASIC uses Zero-Page registers $00 through $05, $0A through $18, and most of the memory registers from $50 through $F8.
- The Apple IIc/IIe machine-language monitor uses Zero-Page registers $00 through $19 and $1E through $25.
- The ProDOS disk operating system uses Zero-Page registers $0A through $1E. However, the ProDOS system saves the information in registers $10 through $1E before it uses them, and it restores the contents of those registers after they are used. In effect, then, the only registers that ProDOS claims for its exclusive use are $0A through $0F.

Since most of Page Zero is used by BASIC, the Apple monitor, and ProDOS, there are only a few small blocks of Page-Zero space that are completely free for use in user-written programs written under ProDOS. Table 11-2 is a list of all such Zero-Page locations.

If you use only the Zero-Page locations in Table 11-2 in your assembly-language programs, you won't interfere with the operation of Applesoft BASIC, ProDOS, or your computer's machine-language monitor. However, if you ever need to use more Zero-Page locations than the ones listed in Table 11-2, you have several options:

- Make sure that your program doesn't require the use of Applesoft BASIC. If you write a program that is completely independent of BASIC, it can make free use of addresses $56 through $FF—and that's most of Page Zero.
- Disable interrupts with an SEI instruction, and then save the contents of part of Page Zero in another segment of memory. You can then use the part of Page Zero that has been saved. After you have finished using that part of Page Zero, you can restore its original contents with another block move. You can restore interrupts with a CLI instruction to resume normal O.S. operations.
- With the help of the RAMRD and RAMWRT soft switches (and other soft switches that will be mentioned later in this book), you can write your program so that it resides in auxiliary memory.

Table 11-2. Zero-Page Locations Available to User-Written Programs

$06—$09
$19—$1F
$CE—$CF
$D6—$D7
$E3
$EB—$EF
$F9—$FF

Then you won't have to worry about any of the main-memory Zero-Page registers that are claimed by Applesoft, ProDOS, or your computer's machine-language monitor. Alternatively, you can use the ALTZP soft switch to select the Zero Page that is located in auxiliary memory. If you choose that procedure, please heed the warnings about the ALTZP switch provided earlier in this chapter.

Addresses $0100 to $01FF (Page 1: the Stack)

The stack is located in Page $01 of your computer's memory —that is, in memory addresses $0100 through $01FF. (A detailed explanation of the hardware stack was presented in Chapter 5.)

In the sections of this chapter that dealt with soft and Zero-Page addresses, we learned that there are actually two stacks in an Apple IIc and expanded Apple IIe—one residing in main memory and the other situated in auxiliary memory. Three soft switches—RAMRD, RAMWRT and ALTZP—can be used to switch back and forth between these two stacks during assembly-language programs. However, great caution should be exercised when these switches are used to change the location of the stack, since moving the stack without taking the proper precautions can crash an assembly-language program.

Addresses $0200 to $02FF (Page 2: the Input Buffer)

Both Applesoft BASIC and the Apple IIc/IIe monitor use memory addresses $0200 through $02FF as a keyboard-input buffer. This buffer is used by both Applesoft BASIC and the GETLN routine that is built into the Apple IIc and the Apple IIe. Since this buffer occupies a full page of

memory—specifically, Page 2—it is 256 bytes long. In assembly-language programs that do not require the use of input strings, this segment of memory can be used for other purposes. If input strings are required but will never reach a length of 256 bytes, then the upper part of Page 2 can be used in assembly-language programs. In auxiliary memory, the $0200-to-$02FF memory block can be used as free RAM.

Addresses $0300 to $03FF (Page 3: Vectors and Link Addresses)

The ProDOS disk operating system and the Apple IIc/IIe monitor use the address space from $03F0 through $03FF for certain link addresses and vectors. The addresses from $0300 to $03EF are generally available to user-written assembly-language programs. In auxiliary memory, the $0300-to-$03FF memory block can be used as free RAM.

Addresses $0400 to $07FF: Text and Low-Resolution Page 1

The segment of memory that extends from $0400 to $07FF is the primary *screen display area* in the Apple IIc and the Apple IIe. When your Apple is in 40-column text mode, the memory registers that extend from $0400 to $07FF are used to hold the text characters that appear on your monitor screen. When the $0400-to-$7FFF memory block is not needed to generate screen displays, it can be used as ordinary free RAM in assembly-language programs.

Since the $0400-to-$07FF memory bank is the main screen display area in the Apple IIc/IIe, it is sometimes known as *Text and Low-Resolution Page 1*, or *TLP1*. (The word page, in this case, refers to a display screen.)

To create an 80-column text display, the Apple IIc/IIe ordinarily uses two screen display areas, usually TLP1 and TLP1X. Alternatively, display areas *TLP2* (*Text and Low-Resolution Page 2*) and TLP2X may be used. When the Apple IIc/IIe is in 80-column mode, it creates an 80-column screen display by interleaving whatever main-memory screen buffer is being used with the corresponding screen buffer in auxiliary memory. When this technique is used, every other character on the screen comes from the main-memory screen buffer, and the characters in between come from the auxiliary-memory screen buffer. A more detailed explanation of how this process works will be provided in later chapters that deal specifically with Apple graphics and video screen displays.

Addresses $0800 to $0BFF: Text and Low-Resolution Page 2

As we have seen, Text and Low-Resolution Page 2 (TLP2) is an alternate screen buffer that can be used to generate a 40-column text display. This screen buffer is sometimes used together with TLP1 so that programs can "flip" back and forth instantly between different screen displays. If the $0800-to-$0BFF memory block is not needed in a program, it can be used as free RAM.

In auxiliary memory, the screen display area that corresponds to TLP2 is known as TLP2X. TLP2X, like TPL2, can be used as free RAM when it is not needed for screen displays.

In addition to the text and low-resolution screen buffers described above, four high-resolution screen buffers are also available in the Apple IIc and the expanded Apple IIe. Locations and descriptions of these high-resolution screen-memory areas will be listed later in this section.

In the Apple IIc and the Apple IIe, soft switches are used to determine which area or areas of memory will be used to generate screen displays. This set of soft switches is described in Table 11-3.

Table 11-3. How Soft Switches Are Used to Select Buffers for Screen Displays

Name of Switch	Location of Switch	How Switch Is Used
RAMRD	$C002	Writing any value to $C002 turns RAMRD off and selects main memory for reading.
RAMRD	$C003	Writing any value to $C003 turns RAMRD on and selects auxiliary memory for reading.
RAMWRT	$C004	Writing any value to $C004 turns RAMWRT off and selects main memory for writing.
RAMWRT	$C005	Writing any value to $C005 turns RAMWRT on and selects auxiliary memory for writing.
HIRES	$C056	Writing any value to $C056 turns HIRES off. When HIRES is off, a text and low-resolution page (TLP) is displayed, and PAGE2 switches between TLP1 and TLP2.
HIRES	$C057	Writing any value to $C057 turns HIRES on. When HIRES is on, a high-resolution page is displayed and PAGE2 switches between HRP1 and HRP2.
80STORE	$C000	Writing any value to $C000 turns 80STORE off. When 80STORE is off, RAMRD and RAMWRT will determine whether the display space in main or auxiliary memory will be used for reading and writing. PAGE2 will select pages for display, but not for reading and writing.

Table 11-3. How Soft Switches Are Used to Select Buffers for Screen Displays *(continued)*

Name of Switch	Location of Switch	How Switch Is Used
80STORE	$C001	Writing any value to $C001 turns 80STORE on. When 80STORE is on, PAGE2 will switch between TLP1 and TLP1X (if HIRES is off) or between HRP1 and HRP1X (if HIRES is on). Also, RAMRD and RAMWRT will be overridden with respect to screen displays, and pages selected by HIRES and PAGE2 will be displayed.
PAGE2	$C054	Writing any value to $C054 turns PAGE2 off. When PAGE2 is off and HIRES is off, TLP1 will be selected. When PAGE2 is off and HIRES is on, HRP1 will be selected. If 80STORE is off, RAMRD and RAMWRT will determine whether the display space in main or auxiliary memory will be used for reading and writing, and PAGE2 will select pages for display, but not for reading and writing. If 80STORE is on, then RAMRD and RAMWRT will be overridden with respect to screen displays, and pages selected by HIRES and PAGE2 will be displayed.
PAGE2	$C055	Writing any value to $C055 turns PAGE2 on. When PAGE2 is on and HIRES is off, TLP2 will be selected. When PAGE2 is on and HIRES is on, HRP2 will be selected. If 80STORE is off, RAMRD and RAMWRT will determine whether the display space in main or auxiliary memory will be used for reading and writing, and PAGE2 will select pages for display, but not for reading and writing. If 80STORE is on, then RAMRD and RAMWRT will be overridden with respect to screen displays, and pages selected by HIRES and PAGE2 will be displayed.
RDRAMRD	$C013	Bit 7 of $C013 can be read to determine whether main (0) or auxiliary (1) memory is in use for reading.
RDRAMWRT	$C014	Bit 7 of $C014 can be read to determine whether main (0) or auxiliary (1) memory is in use for writing.
RDHIRES	$C01D	Bit 7 or $C01D can be read to determine whether HIRES is on (1) or off (0).
RD80STORE	$C018	Bit 7 or $C018 can be read to determine whether 80STORE is on (1) or off (0).
RDPAGE2	$C01C	Bit 7 or $C01C can be read to determine whether PAGE2 is on (1) or off (0).

Addresses $0C00 to $1FFF: Free RAM

The memory registers that extend from $0C00 to $1FFF are free RAM and can be used for any purpose in user-written programs. Since it is considered good programming practice to separate the memory areas that are used for programs from memory areas that are used for data, the $0C00-to-$1FFF (memory block) is often used for data tables in Apple IIc/IIe programs. The larger block of free RAM that extends from $6000 to $BFFF (or from $2000 or $4000 to $BFFF, if high-resolution screen graphics are not needed) can then be used for storing executable-code portions of machine-language programs.

In auxiliary memory, just as in main memory, the $0C00-to-$1FFF memory bank is available for use as free RAM.

Addresses $2000 to $5FFF: High-Resolution Pages 1 and 2

Memory block HRP1, which extends from $2000 to $3FFF, is the screen-memory buffer that is used most often to create high-resolution graphics displays. Many high-resolution graphics programs also make use of screen memory area HRP2, which extends from $4000 to $5FFF. When these blocks of memory are not needed to generate screen displays, they can be used as free RAM.

In auxiliary memory, the memory block that extends from $2000 to $3FFF is called HRP1X, and the memory block that extends from $4000 to $5FFF is called HRP2X. These memory banks can also be used as free RAM when they are not needed to generate screen displays.

The Apple IIc/IIe can generate double high-resolution color graphics by interleaving the high-resolution screen buffers in main memory with their corresponding buffers in auxiliary memory. More about double high-resolution graphics will be provided in the graphics chapters of this book.

Addresses $6000 to $BFFF: Free RAM

As mentioned earlier in this chapter, the memory bank that extends from $6000 to $BFFF in main memory is called main RAM, and the 48K bank that extends from $6000 to $BFFF in auxiliary memory is

called auxiliary RAM. These are the primary RAM banks in the Apple IIc and the Apple IIe. Both banks are available for use in user-written programs.

Addresses $C000 to $C0FF: Hardware Addresses

The memory registers that extend from $C000 to $C0FF are used for five different types of hardware functions:

- The *data input* can be read to determine whether a key on the keyboard has been pressed and what key that is.
- A set of *flag inputs* can be used to read hand controllers, the OPEN-APPLE and CLOSED-APPLE keys on the Apple IIc/IIe keyboard, and the button switch on the Apple mouse.
- A pair of *strobe outputs* can determine whether a key has been pressed and what key that is; they can also read game paddles.
- A *toggle switch* operates the small loudspeaker in the Apple IIc/IIe.
- A series of soft switches, some of which have been mentioned in this chapter. Other soft switches that reside in the $C000-to-$C0FF block of memory will be described in chapters 12 and 13.

Addresses $C100 to $CFFF: ROM Addresses

The block of memory that extends from $C100 to $CFFF is dedicated solely to ROM. The ROM addresses that reside in this block of memory include:

- Entry points for accessing Serial Port 1 and Serial Port 2 on the IIc.
- Entry points for accessing video output, enhanced video output, and miscellaneous I/O.
- In an Apple IIc or a mouse-equipped IIe, entry points for accessing Mouse firmware.
- Entry points for disk I/O.

RAM Addresses $D000 to $FFFF

When addresses $D000 to $FFFF are used as RAM, the block of memory that extends from $D000 to $DFFF can be used in four different ways: as *Main Memory $D000 Bank 1*, *Main Memory $D000 Bank 2*, *Auxiliary Memory $D000 Bank 1*, and *Auxiliary Memory Bank $D000 Bank 2*. Switching back and forth among these four memory

banks is controlled by means of soft switches, using the techniques described earlier in this chapter.

The block of memory that extends from $E000 to $FFFF can be used in two different ways: as *main bank-switched RAM* or as *auxiliary bank-switched RAM*. Switching back and forth between this pair of memory banks is also controlled by means of soft switches. The soft switches that are available for this purpose and the techniques for using them are listed in Table 11-1.

ROM Addresses $D000 to $F7FF: The BASIC Interpreter

An Applesoft BASIC interpreter is built into both the Apple IIc and the Apple IIe. This interpreter resides in the block of memory that extends from $D000 to $F800. It can only be used when ROM is switched into this block of memory. The soft switches listed in Table 11-1 can be used to place ROM in the $D000-to-$F800 block of memory.

ROM Addresses $F800 to $FFFF: The Monitor

When ROM is switched into the $F800-to-$FFFF memory block, that is where the Apple IIc/IIe monitor resides. Since the monitor is a handy utility to have around when assembly-language programs are being written and debugged, it's usually a good idea to avoid placing user-written code into this segment of memory.

Under ProDOS Assembly-Language Programming

With the introduction of the Apple IIc, a new operating system called ProDOS replaced Apple's old II-series disk operating system, DOS 3.3. Most professional software for the Apple IIc and the Apple IIe is now written under ProDOS, so it would probably make sense for you to write your assembly-language programs under ProDOS, too.

ProDOS is not just a disk operating system. It is a complete operating system that allows an assembly-language programmer to manage many of the resources provided by the Apple IIc and the Apple IIe. ProDOS functions primarily as a disk operating system, but it also handles interrupts. In addition, ProDOS has a built-in memory-management tool called a *Machine Language Interface*, or *MLI*. This Machine Language Interface is the portion of the operating system that

receives, validates, and issues operating-system commands.

In Apple IIc/IIe assembly language, MLI commands are used in much the same way that DOS commands are used in BASIC. In Apple IIc/IIe assembly-language programs, MLI calls can be used to open files, close files, create files, delete files, and perform many other disk-related functions. (In fact, virtually all ProDOS commands available in BASIC—and some that are not—are available through MLI.)

A complete discussion of ProDOS and the ProDOS Machine Language Interface is beyond the scope of this chapter. If you want to learn more about ProDOS, there are several books on the subject that you should read, including the *ProDOS User's Manual, BASIC Programming With ProDOS,* and of course, the *ProDOS Technical Reference Manual.* The main purpose of this section is to tell you where ProDOS resides when it's loaded into your computer's memory so you won't overwrite ProDOS when writing programs to be run on a ProDOS-equipped Apple IIc or Apple IIe.

A ProDOS Memory Map

When ProDOS is loaded into the memory of an Apple IIc or an Apple IIe, it takes up varying amounts of RAM, depending upon the configuration of the computer. If ProDOS is used with an Apple IIc, or with an Apple IIe equipped with an expanded 80-column card, some of the space consumed by ProDOS is situated in main memory and some is located in auxiliary memory. ProDOS is usually loaded into memory along with a file called BASIC.SYSTEM, which enables ProDOS to process BASIC commands. When ProDOS and BASIC.SYSTEM are loaded simultaneously, additional memory space is consumed, since the BASIC.SYSTEM file on a ProDOS startup disk includes a ProDOS command interpreter.

Figure 11-3 is a map that shows what blocks of memory are consumed by the ProDOS and BASIC.SYSTEM files on a ProDOS startup disk when ProDOS is loaded into memory. Assembly-language programs that are designed to be run under ProDOS control should not encroach upon the shaded areas in Figure 11-3.

Memory Requirements of Assemblers

When you write an assembly-language program, you should be aware of where your assembler-editor system resides in your computer's memory.

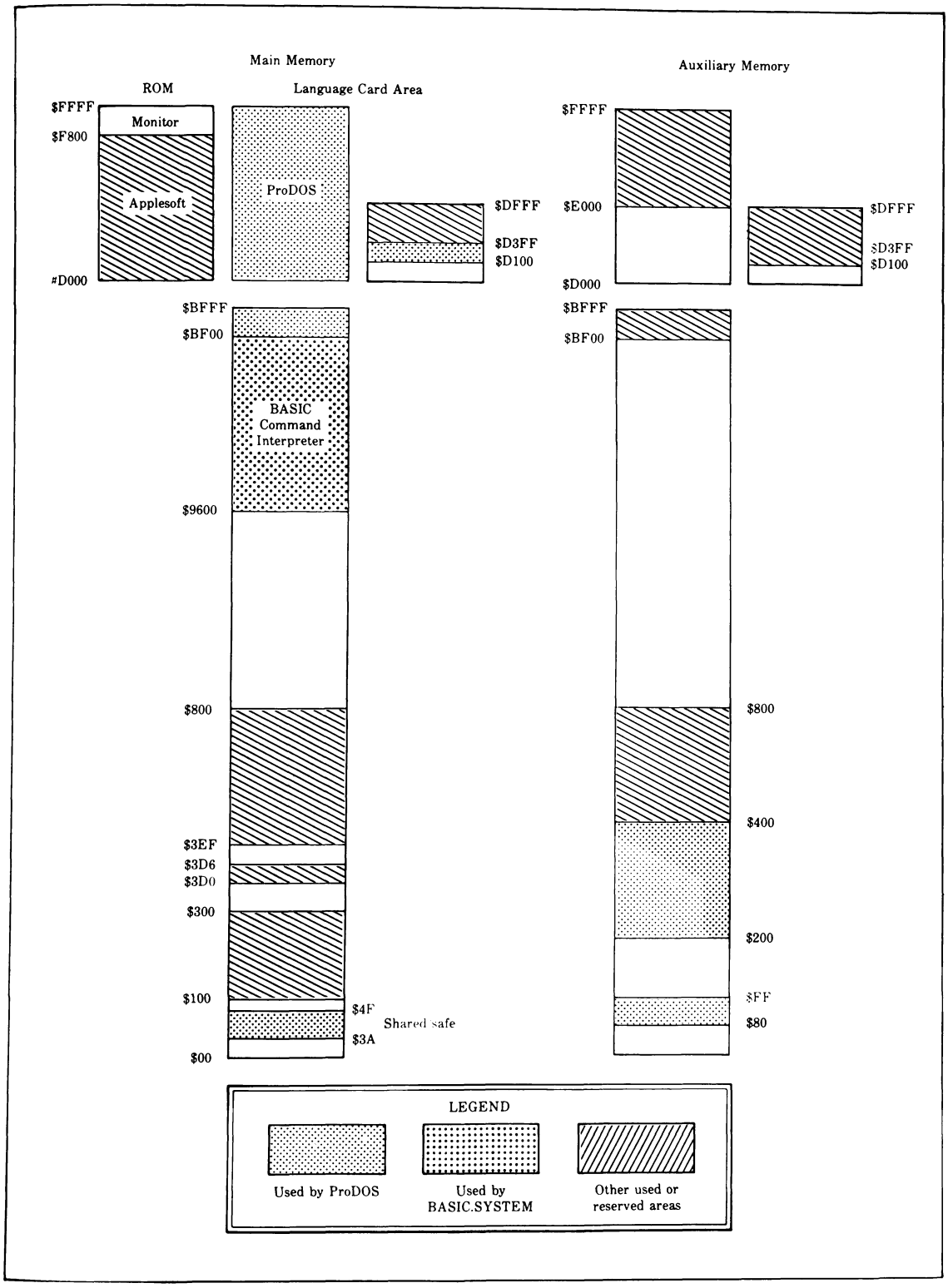

Figure 11-3. ProDOS and BASIC memory map

Then you can avoid the danger of writing a program that damages your assembler while it's editing or assembling your program.

Merlin's Memory

The Merlin Pro assembler is a complex piece of software, and it consumes quite a bit of RAM, in both main and auxiliary memory. But Merlin provides a linking feature that can be used to link long programs together by assembling them directly on a disk. Details on how to use this linker can be found in the Merlin instruction manual.

Mapping the Apple ProDOS Assembler

Figure 11-4 is a memory map of the Apple ProDOS assembler-editor. The Apple package consumes less memory space than Merlin does, and all the space it occupies is in main memory. Thus the Apple ProDOS assembler leaves all 64K of auxiliary memory free for use by user-written programs.

One noteworthy feature of the Apple ProDOS package is that its editor and assembler do not reside in memory at the same time. Therefore, when you write a source-code program using the Apple assembler, you have to save it on a disk before you can assemble it. That process takes time, but it saves memory.

Memory Requirements Of the ORCA/M Assembler

When the ORCA/M assembler is used to edit and assemble a program, it performs even more disk-swapping than does the Apple assembler-editor system. Because of all this disk-swapping—and because of the elegant way in which ORCA/M is designed—it is possible for the user of an ORCA/M system to write very long assembly-language programs and place them almost anywhere in memory. According to The Byte Works of Albuquerque, New Mexico, which manufactures ORCA/M, the following list presents virtually all of the memory constraints on the user of an ORCA/M assembler:

- The ORCA/M assembler-editor runs under the control of a program that is called ORCA.HOST. This program, which occupies addresses $0800 through $1FFF, must remain in memory at all times when ORCA/M is running. Therefore, it must never be overwritten by a user-written program.
- On Page Zero, memory locations $00 through $7F are available for use in user-written programs, but addresses $80 through $FF should be avoided, since they are used by the ORCA/M system.

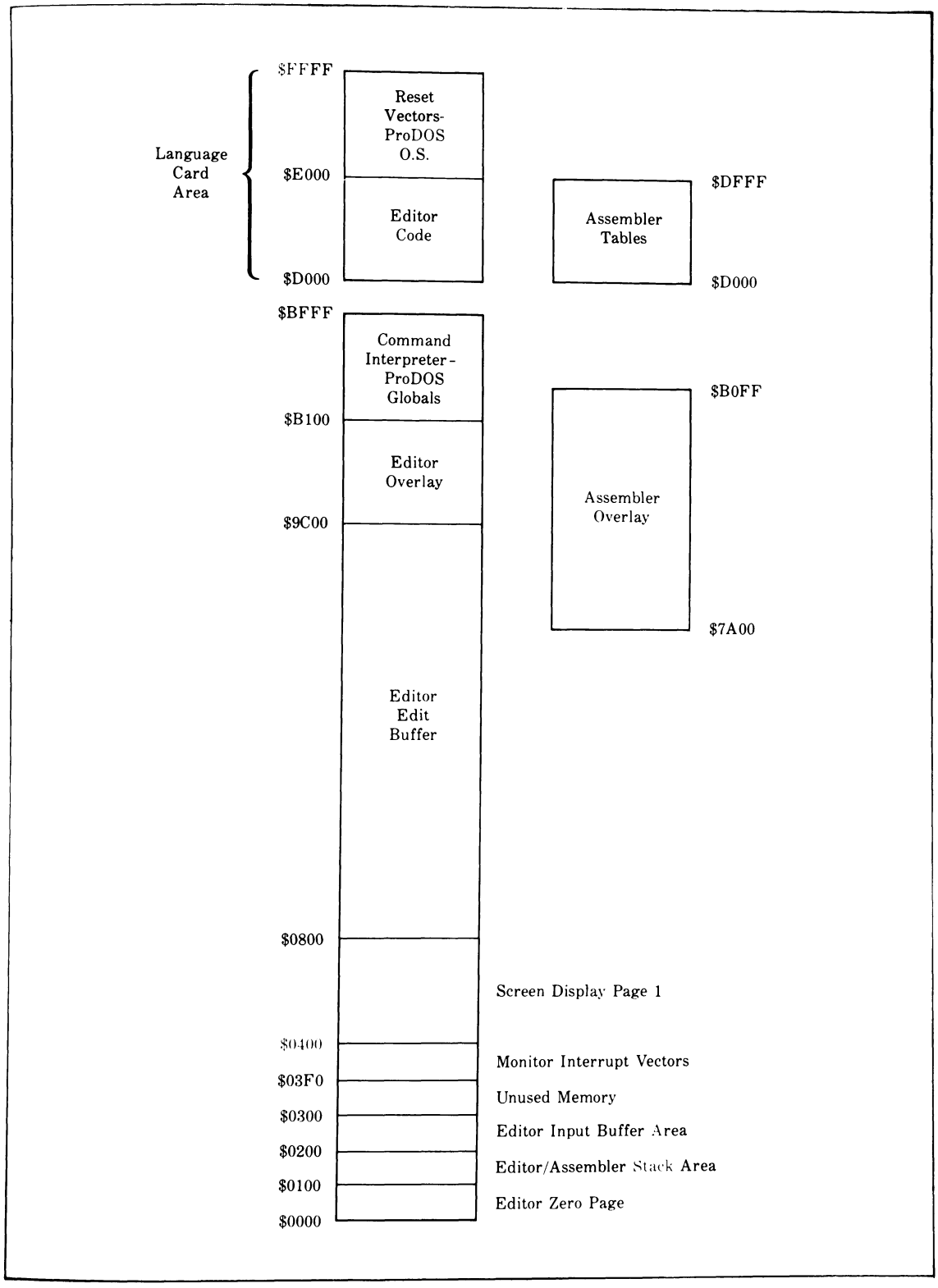

Figure 11-4. Apple ProDOS assembler memory map

- The ProDOS line buffer uses memory registers $200 through $2FF, and ORCA/M uses the ProDOS line buffer when the assembler is in command mode. Until a program is debugged and ready to run, it is a good idea to avoid using these memory addresses.
- Non-standard clock cards sometimes use memory registers $300 to $3FF, so assembly-language programs that make use of these registers could interfere with the operation of such cards. (This restriction applies to programs written using *any* assembler, not just ORCA/M.)

Conclusion

Once you're familiar with the memory requirements (and the memory limitations) of your computer, operating system, and assembler-editor system, you have all the information you need to use memory safely and efficiently in assembly-language programs.

12

Fundamentals Of Apple IIc/IIe Graphics

Unless you spend most of your free time with other programmers, you've probably noticed that most people aren't too impressed with elegant algorithms or long listings of assembly-language code. Fortunately, though, if you know how to write graphics programs—especially the kind of fast-action, razzle-dazzle programs that can be created only in assembly language—then you really can amaze your friends with your knowledge of assembly language. If that prospect intrigues you, don't stop now, because the rest of this book is about assembly-language graphics programs.

In this chapter, you'll learn how the Apple IIc and the Apple IIe generate their text and graphics displays. In Chapter 13, you'll have a chance to type and run programs that demonstrate your computer's low-resolution graphics mode; you'll also see how game paddles, joysticks, and the

Apple mouse can be used in assembly-language programs. Chapter 14 will show you how to customize a character set, how to display standard-size and headline-size characters on a high-resolution screen, and how to write programs using double high-resolution graphics.

Text and Graphics Modes

If you own an Apple IIc, or a late-model Apple IIe with an expanded memory 80-column card installed, your computer can be used in six primary text and graphics modes:

- Forty-column text
- Eighty-column text
- Low-resolution graphics
- Double low-resolution graphics
- High-resolution graphics
- Double high-resolution graphics.

All six of these display modes will be discussed at least briefly in this chapter. However, we will focus most of our attention on the two most commonly used display modes in Apple IIc/IIe graphics programming: the standard low-resolution graphics mode and the standard high-resolution graphics mode.

40-Column and 80-Column Text Modes

A standard Apple IIe, without an 80-column text card or an AppleColor Adaptor card installed, has only one text mode: a 40-column mode that generates a 40-column by 24-line text display. An Apple IIc, or an Apple IIe equipped with an 80-column card, can generate either a 40-column by 24-line text display or an 80-column by 24-line text display.

As you may recall from Chapter 11, the Apple IIc and the Apple IIe are equipped with soft switches that can be used to determine whether a 40-column or an 80-column display will be used in an assembly language program. Soft switches can also be used to switch back and forth between text and graphics displays and to place a four-line, 40-column or 80-column "text window" at the bottom of a low-resolution or high-resolution graphics screen.

Memory Mapping

Your Apple IIc or Apple IIe uses a technique called *memory mapping* to create its text and graphics displays. This means that, by storing specific values in a certain block of your computer's memory, you can control its screen display. When your Apple is in one of its text modes, for example, you can store ASCII (American Standard Code for Information Interchange) codes (modified for Apple computers) in a specific block of RAM. Once you have done that, your computer's operating system will convert each code number that you have used into a letter, number, or special character. Then it will display each character in a screen position that is determined by the specific byte in RAM in which the code number for the character has been stored.

Ordinarily, the Apple ASCII codes that generate a 40-column text display are stored in the area of RAM called Text and Low-Resolution Display Page 1, extending from memory address $400 to memory address $7FF. Alternatively, these codes can be stored in the block of RAM known as Text and Low-Resolution Display Page 2, which ranges from $800 to $BFF.

If you own an Apple IIc, or an Apple IIe equipped with an expanded 80-column card, there are two other areas of RAM that can be used as screen memory for a 40-column text display. These blocks of RAM, called Text and Low-Resolution Display Pages 1X and 2X, extend from $400 to $7FF and from $800 to $BFF in auxiliary memory.

In both the Apple IIc and the Apple IIe, soft switches can be used to determine what areas of main and auxiliary memory will be used for screen memory. The functions and memory addresses of most of these soft switches were listed in Chapter 11. Memory maps showing the segments of RAM that can be used for storing text and graphics information were also presented in Chapter 11.

Figure 12-1 is a memory map of Display Page 1, the area of memory most often used for an Apple IIc/IIe 40-column text display.

Creating an 80-Column Text Display

An Apple IIc, or an Apple IIe equipped with an 80-column text or color card, is also capable of generating an 80-column by 24-line text display. When an Apple IIc or IIe is in 80-column text mode, the ASCII codes for characters in even-numbered screen columns (with the first column

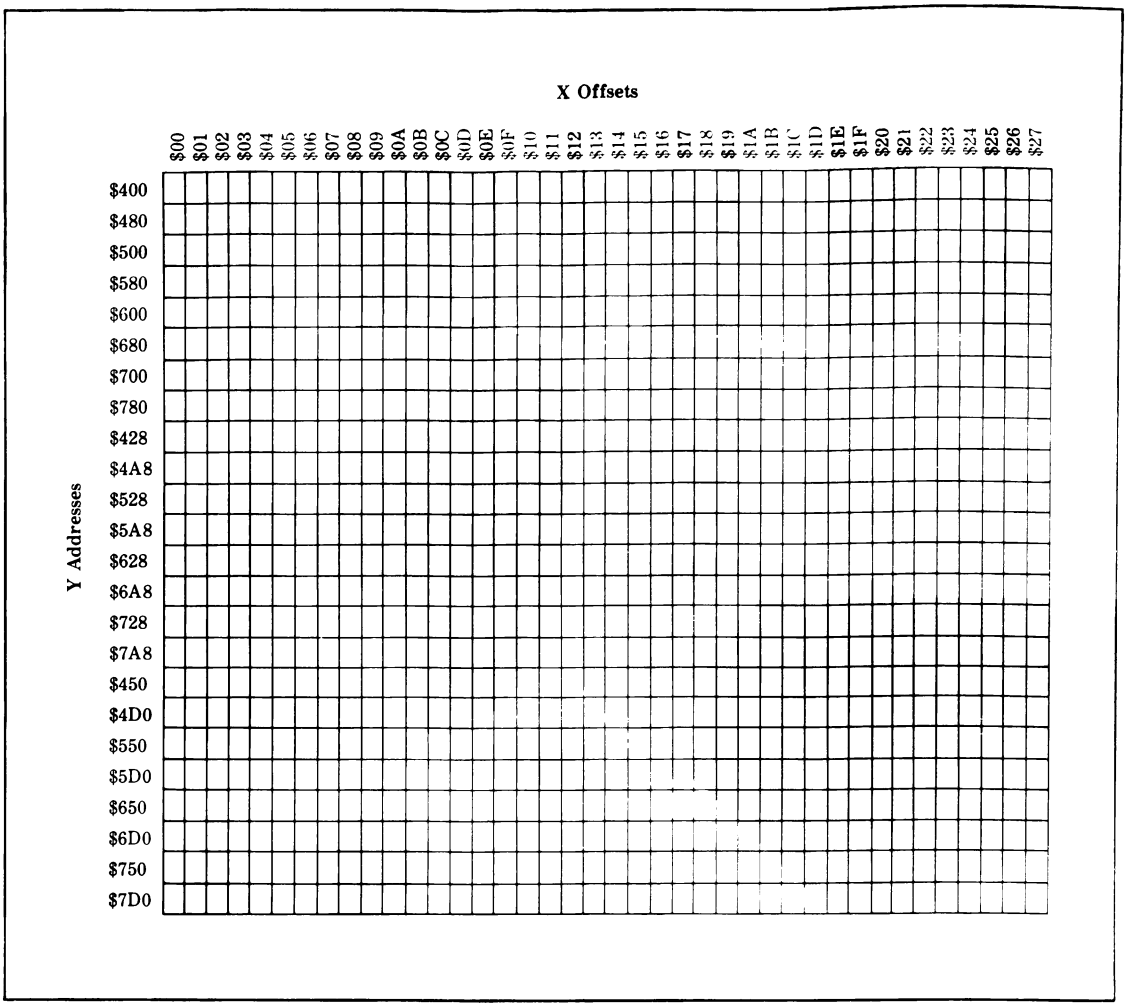

Figure 12-1. 40-column text screen display map

on the screen numbered Column 0) are stored on Display Page 1 in main memory. The codes for the characters in odd-numbered screen columns (with the second column on the screen numbered Column 0) are stored on Display Page 1X in auxiliary memory. When the characters from these two blocks of memory are displayed on an 80-column screen, they are automatically interleaved as illustrated in Figure 12-2.

Fortunately, you may never have to worry about this interleaving process when you write text programs for your Apple IIc or IIe. This process will ordinarily be taken care of by your computer's operating system.

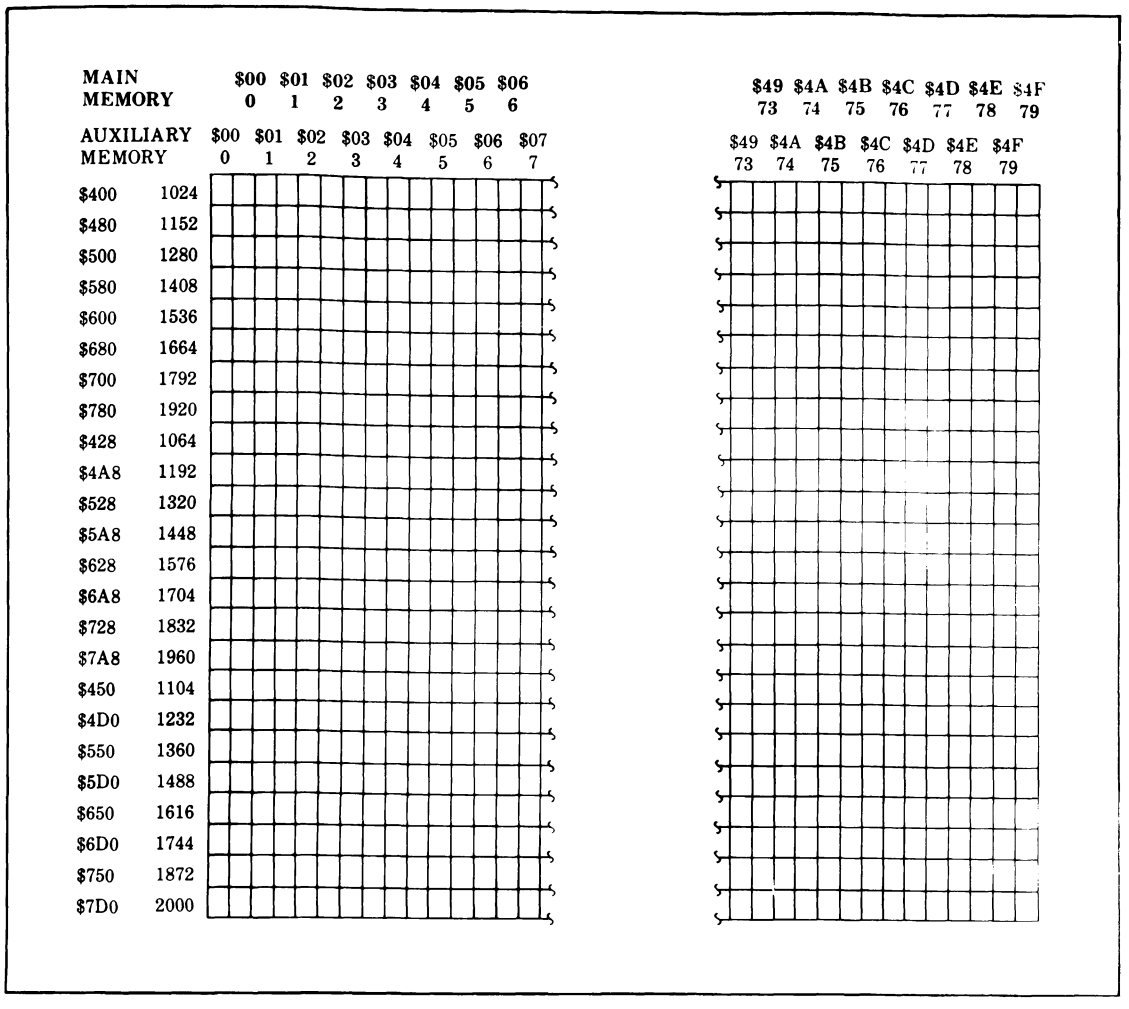

Figure 12-2. 80-column text screen display map

Low-Resolution Graphics

The memory-mapping system that is used for low-resolution graphics is very similar to the system used for a 40-column text display. An Apple IIc/IIe low-resolution graphics screen is made up of a grid of colored rectangles called *picture elements*, or *pixels*. This matrix of colored blocks measures 40 pixels wide by 48 pixels deep—in other words, 40 columns by 48 rows.

To put your computer into low-resolution graphics mode, you simply set a soft switch situated at memory address $C056, as explained in Chapter 11.

Once your computer is in low-resolution graphics mode, you can create a low-resolution display by storing data in Display Pages 1, 2, 1X, and 2X (the same areas of memory that store the Apple ASCII codes used for 40-column text displays). When your computer is in low-resolution graphics mode, however, it will not interpret the data stored in these areas as codes for text characters. Instead, it will interpret them as colors and will display those colors on the screen.

In low-resolution graphics mode, the Apple IIc/IIe can display 16 colors, including black and white. These colors, along with the codes that are used to generate them in low-resolution graphics, are listed in Table 12-1.

Figure 12-3 shows how Display Page 1 is laid out when it is used to display a low-resolution graphics screen. Each byte stored in screen memory is used to generate two colored pixels, one sitting on top of the other. Thus there are twice as many elements on a low-resolution graphics screen as there are on a 40-column text screen; a low-resolution graphics screen measures 40 columns by 48 rows, while a 40-column text screen measures 40 columns by 24 rows.

High-Resolution Graphics

In high-resolution (or high-res) graphics mode, your computer can display a grid of screen dots measuring 280 dots wide by 192 dots deep in monochrome; the effective resolution is 140 dots wide by 192 dots deep when colors are used. In color high-resolution graphics, the color of

Table 12-1. Color Codes Used in Low-Resolution Graphics

HEX	COLOR	HEX	COLOR
$0	Black	$8	Brown
$1	Magenta	$9	Orange
$2	Dark Blue	$A	Gray 2
$3	Purple	$B	Pink
$4	Dark Green	$C	Light Green
$5	Gray 1	$D	Yellow
$6	Medium Blue	$E	Aquamarine
$7	Light Blue	$F	White

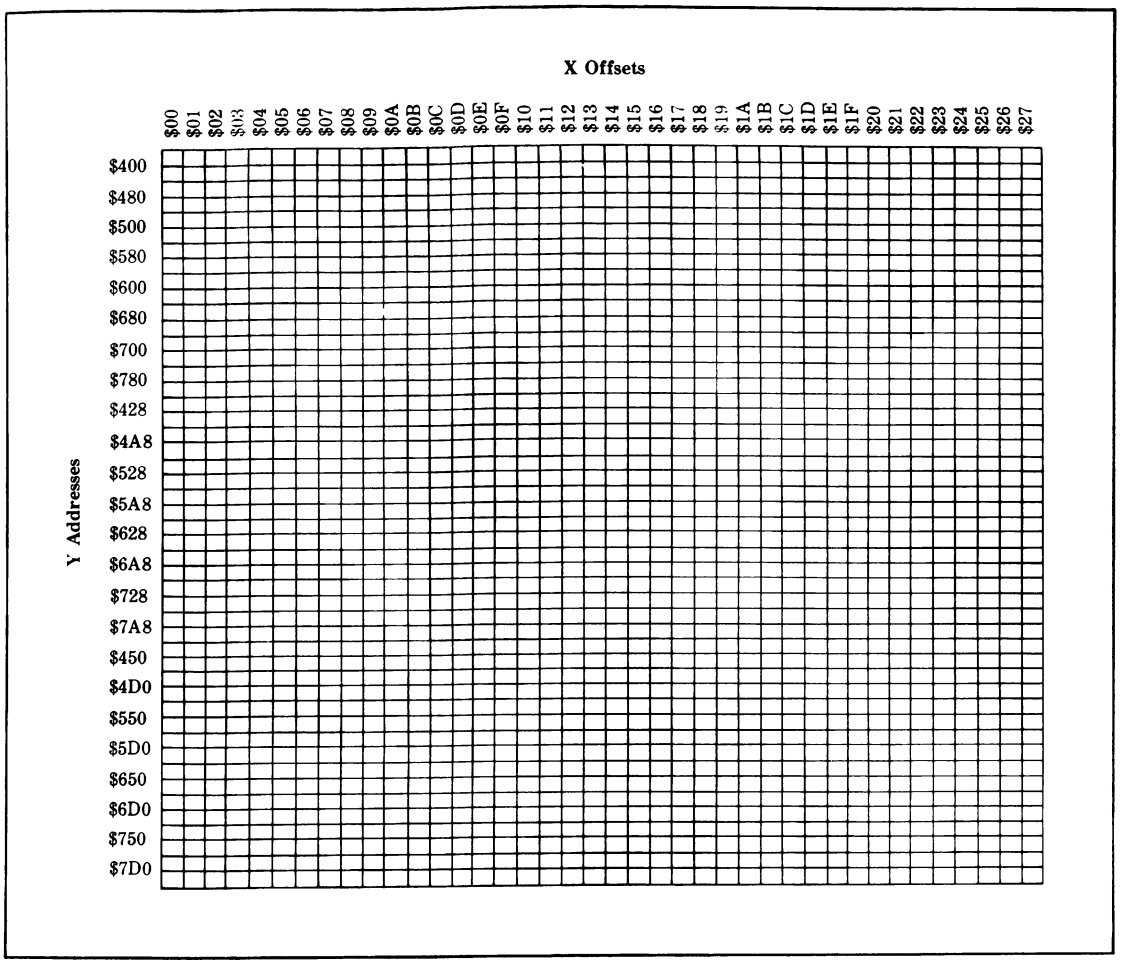

Figure 12-3. Low-resolution graphics screen map

each pixel on the screen can be individually controlled. When you know how to use high-resolution graphics, you can create pictures and characters that are fairly detailed and quite attractive.

There are some limitations, however, on the ways in which colors can be used in Apple IIc/IIe high-resolution graphics. Only six true colors are available in high-res graphics: black, white, violet, green, orange, and blue. By using alternating rows of colors, you can blend some of these colors to create other colors, but unless you're a real expert at this kind of color blending, it's seldom worth the effort that it requires.

There are also limitations on the way that the six colors available in high-resolution graphics can be combined. These limitations will be

explained in detail in Chapter 14, which deals solely with high-resolution graphics.

When your computer is in high-resolution graphics mode, the area of RAM that is used most often for screen memory ranges from memory address $2000 to $3FFF—a total of 8192 bytes of memory. This block of memory is sometimes referred to as High-Resolution Page 1. Another 8192-byte block of memory, ranging from $4000 to $5FFF, is also used quite often. This block of RAM is sometimes referred to as High-Resolution Page 2.

If you own an Apple IIc or an Apple IIe equipped with an expanded 80-column card, two more blocks of high-resolution screen memory are available. These two segments of RAM are High-Resolution Display Page 1X, which extends from $2000 to $3FFF in auxiliary memory, and High-Resolution Display Page 2X, which ranges from $4000 to $5FFF in auxiliary memory.

Your Apple uses what is often referred to as a *bit-mapped* screen display when it is in high-resolution graphics mode. In a bit-mapped screen display—the type of display most often used in commercial graphics programs—each dot on the screen corresponds to a bit in the computer's memory. When a bit is set, a dot on the screen generally lights up; when the bit is cleared, the dot goes dark. Every dot on the screen can thus be controlled individually by a bit-mapped graphics program.

In Apple IIc/IIe high-resolution graphics, each byte in screen memory controls seven dots on the screen. Since there are eight bits in a byte, one bit in each byte is therefore left free for use as a status bit. By setting or clearing this status bit, a programmer can control the colors generated by the other bits in the same byte when they are displayed on the screen.

More details on how this process works will be provided in Chapter 14. Meanwhile, we'll take a brief look at double low-resolution graphics and double high-resolution graphics, two display modes that were not available to Apple programmers until the introduction of the Apple IIc and the Apple IIe.

Double Low-Resolution Graphics

Double low-resolution graphics, as its name implies, is a screen display mode with twice the resolution of ordinary low-resolution graphics. With double low-resolution graphics, you can create a 16-color screen display that is 80 pixels wide and 48 pixels high.

If you own an Apple IIc, or an Apple IIe with an 80-column card and a Revision-B (or later) motherboard, you can write and run double low-resolution graphics programs.

To find out what kind of circuit board your IIe has, open it up and look at the markings that are printed on the main circuit board just behind its expansion slots. On a Revision-B computer, you'll see the letter B after the part number. (The part number for my IIe is 820-0064-B.) On a Revision-A machine, the letter after the part number will be an A.

To use double low-resolution (or double high-resolution) graphics with an Apple IIe, you also need a properly configured 80-column text or text/color card. If you have a standard, non-color 80-column card, you also have to connect Pins 50 and 55 on your card. If your non-color card is an unexpanded model (without an extra 64K of memory), you need to connect those pins with a soldering iron. (Unless you're a real hardware expert, you should seek the assistance of a qualified technician.)

Having an expanded non-color card, however, makes the job of connecting Pins 50 and 55 much easier. All you have to do is connect a jumper cable that came with your card. You won't have to do any hardware work at all, though, if you have an Apple IIc or an Apple IIe with a Text/AppleColor card installed, because your computer is ready for double low-res and double high-res graphics display.

Once you're sure that your computer can handle double low-res programming, you can put it into double low-res mode by storing any value at all in the soft switches situated at memory addresses $C050, $C056, $C052, $C00D, $C05E, and $C05F. As you can determine for yourself by consulting the soft-switch tables in Chapter 11, setting these four soft switches will select graphics rather than text, low-resolution rather than high-resolution graphics, and full-screen graphics (without a text window). Setting these switches will also turn on your computer's 80-column firmware. Next you have to turn on still another soft switch, called AN3, which is situated at memory locations $C05E and $C05F. To enable double low-resolution or double high-resolution graphics, you have to turn AN3 *off* by either reading or writing to memory address $C05E. To disable double low-resolution or double high-resolution graphics, you have to turn AN3 *on* by either reading or writing to memory address $C05F.

To put a properly equipped Apple into double low-resolution mode, the following sequence of instructions could be used:

```
STA $C050 ;TURN OFF TEXT MODE
STA $C056 ;TURN OFF HI-RES MODE
STA $C052 ;TURN OFF MIXED MODE (OPTIONAL)
STA $C00D ;TURN ON 80-COLUMN DISPLAY
STA $C05E ;ENABLE DOUBLE LOW-RES GRAPHICS
```

When an Apple IIc or IIe is in double low-resolution graphics mode, it displays colors in much the same way that it displays 80-column text when it's in 80-column text mode. When double low-res graphics are enabled, the data stored in Low-Resolution Graphics Page 1 (the block of memory that extends from $400 to $7FF in main memory) will be displayed as blocks of color in even-numbered columns (beginning with Column 0) on your computer screen. These colors will be interleaved with colors generated by data stored on Low-Resolution Graphics Page 1X (the block of memory that extends from $400 to $700 in auxiliary memory). The colors generated by Page 1X will be displayed in the odd-numbered columns (beginning with Column 1) on your computer screen.

Auxiliary-Memory Color Codes

If you ever decide you want to use double low-resolution graphics, you may encounter one small problem: because of main-memory and auxiliary-memory timing differences, the color codes that are used for low-resolution data stored in auxiliary memory are different from those used for data stored in main memory. Table 12-2 lists the color codes used for double low-resolution graphics data stored in auxiliary memory.

Double High-Resolution Graphics

Double high-resolution graphics is a mode that offers either twice the resolution or twice the number of colors of ordinary high-resolution

Table 12-2. Color Codes Used for Double Low-Resolution Graphics Data Stored in Auxiliary Memory

HEX	COLOR	HEX	COLOR
$0	Black	$4	Brown
$8	Magenta	$C	Orange
$1	Dark Blue	$5	Gray 2
$9	Purple	$D	Pink
$2	Dark Green	$6	Light Green
$A	Gray 1	$E	Yellow
$3	Medium Blue	$7	Aquamarine
$B	Light Blue	$F	White

graphics. By using double high-resolution graphics, you can create an ultra-high-resolution monochrome display that measures 560 dots wide by 192 dots high. Alternatively, you can create a 16-color display measuring 140 dots wide by 192 dots high that can display any dot on the screen in any of the 16 available colors.

The screen map that is used for double high-resolution graphics is similar to the one used for an 80-column text display. When an Apple IIc or IIe is in high-resolution mode, it interleaves screen data from High-Resolution Display Pages 1 and 1X, thus displaying two bytes in the space normally occupied by one. Pages HRP2 and HRP2X in auxiliary memory can also be used for the storage of double high-resolution screen data.

To use double high-resolution graphics, you need either any Apple IIc or an Apple IIe that is equipped to generate double low-resolution graphics. To get the maximum benefit from double high-resolution graphics, you also need either an 80-column monochrome monitor (for double high-resolution monochrome displays) or a high-quality, 80-column RGB monitor (for double high-resolution color displays). It is possible to display double high-res graphics in color on an ordinary 40-column color monitor or TV set, but single dots will sometimes appear more dimly than normal.

Mapping the Graphics Screens

Now that we've taken a brief glance at each of the graphics modes that can be generated by the Apple IIc and IIe, we're ready to examine in more detail the screen maps used by each of these graphics modes.

First, however, let's take a closer look at the 40-column text display illustrated in Figure 12-1. Notice that strings of hexadecimal numbers have been used to identify each column and each row on the map. These numbers look like column and row coordinates, but that's not really what they are. The numbers running down the side of the map are actually memory addresses—specifically, the addresses of the initial bytes in each column. Because these addresses are used to represent vertical (or Y) positions on the memory map, they are sometimes referred to as *Y addresses.* The numbers across the top of the map represent *offsets* that can be added to the map's Y addresses to determine the exact memory address of any byte displayed on the map. Since these numbers are used to represent the horizontal (or X) position on the map, they are sometimes referred to as *X offsets.*

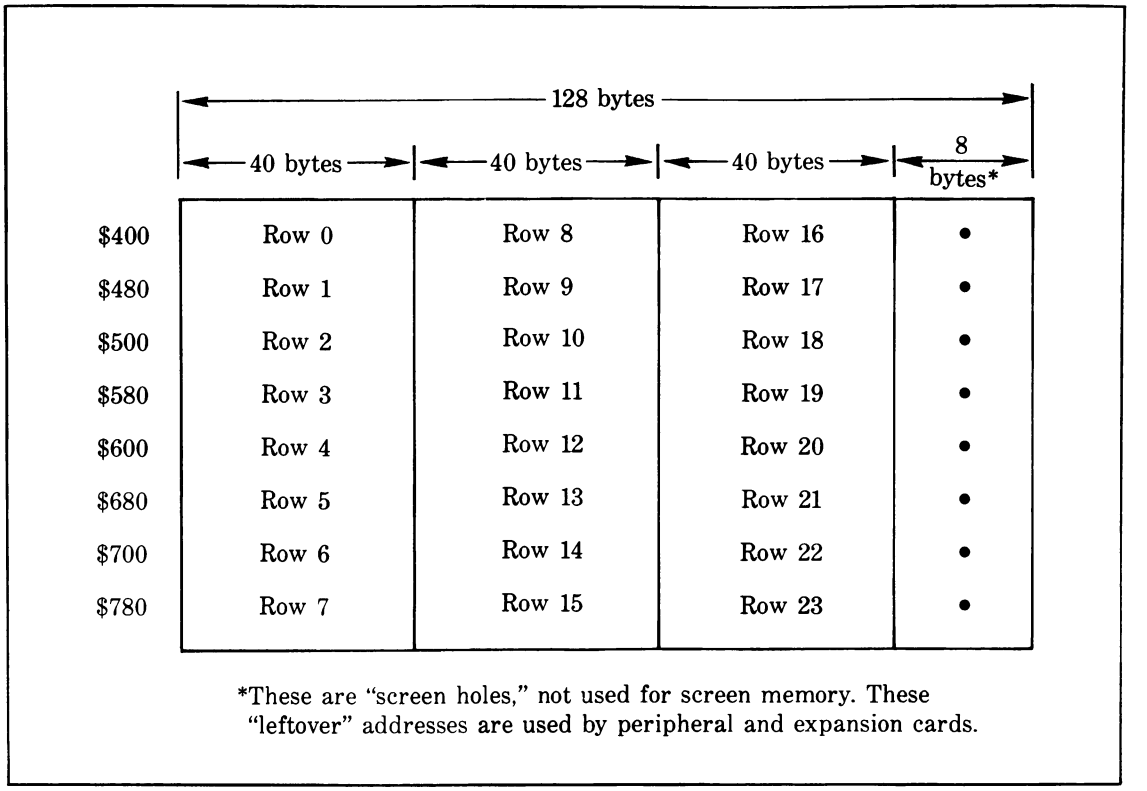

Figure 12-4. How screen display memory is stored in RAM

By using the X offsets and Y addresses on an Apple screen map, the exact memory location of any byte on the map can easily be pinpointed. Merely add the character's X offset to its Y address. That will give you the character's exact address in RAM.

If you look closely at the X offsets across the top of Figure 12-1, you will see that the columns on the map are numbered consecutively from $00 to $27—or from 0 to 39 in decimal notation. That numbering makes it easy to determine the screen column location of any character. It doesn't matter what row a character is on; if it's in a given column on the screen, its X offset will always be the same.

Notice that the Y addresses down the side of Figure 12-1 are *not* consecutively numbered. Instead, they are arranged in three batches of eight rows each. The first eight rows on the screen map are numbered $400 through $780. The next eight rows are numbered $428 through $7A8. The eight rows at the bottom of the screen are numbered $450 through $7D0.

At first glance, this numbering system might seem strange and cumbersome. Actually, however, it does make some sense. If your computer's screen memory were not split up, but were simply dropped into a solid, 960-byte block of RAM, X offsets and Y addresses could not be used to locate the characters on the screen. Characters in the same column on the screen would hardly ever have the same X offset, and characters in the same row on the screen would not necessarily have the same Y address. Therefore, the somewhat confusing X-offset and Y-address system that is used for your computer's 40-column screen display is clearly preferable to no system at all.

Figure 12-4 shows what the Apple IIc/IIe 40-column screen map looks like when it is displayed as a consecutive block of memory. As you can see, the address-and-offset system designed for your computer is quite memory-efficient. This system allows a 960-byte screen map to be placed into 1024 bytes of memory, leaving only 64 bytes unused. Yet each character on the screen can be located by adding a column offset to a row address.

There is one more point about screen mapping worth mentioning. The systems used for mapping low-resolution and high-resolution graphics are based on the same principles as those used in mapping your Apple's 40-column text screen. Figure 12-3 clearly shows that the screen map used for low-resolution graphics is the same as the one used for 40-column text graphics. The only difference is that each byte on the screen has been split horizontally into two pixels.

13

Game Paddles, Joysticks, and The Apple Mouse

To become an expert graphics programmer for the Apple IIc or the Apple IIe, it is essential to have an understanding of how to write programs for hand controllers, such as *game paddles*, *joysticks*, and the *Apple mouse*. In this chapter, you'll learn how hand controllers work and how they are used in graphics programs. You'll also be provided with two type-and-run programs that demonstrate how to program hand controllers in assembly language. One of these programs will illustrate the use of game paddles and joysticks. The other program will illustrate the use of the Apple mouse.

The Evolution of the Hand Controller

The first kind of hand controller to be used with Apple computers was the game paddle. Because the game paddle derives its name from its function, not from its looks, it bears little resemblance to other kinds of paddles. The devices are called paddles because they were used to simulate ping-pong paddles in the early arcade game *Pong*. Electronically, though, a game paddle is really a variable resistor; mechanically, it's usually a rotary knob attached to a rectangular base.

Game paddles always come in pairs, and the pair of paddles that you get in a set are designed to be connected to the same I/O port. Thus, the paddles in a set always share the same port when they are connected to an Apple IIc or IIe.

The Apple IIe has two kinds of ports to which game paddles can be connected. One of these ports is an internal 16-pin DIP socket that is installed on the Apple IIe motherboard and is labeled "GAME I/O". The other paddle port available to IIe owners is a D-type miniature connector on the computer's back panel. Both of these ports can also be used for the installation of joystick controls.

The Apple IIc has only one port—a D-type connector on the back panel—to which game paddles can be connected. This port can also be used as a joystick connector and as a socket for a mouse control.

Paddles, Joysticks, and Mice Compared

Since a game paddle is nothing but a rotary controller, it can draw a line or move an object in only one direction on a computer screen. A joystick can be moved in any direction and can therefore move an object in any direction on a screen. However, a joystick does not usually offer as much control over positioning an object on a screen as does a game paddle.

A mouse, when properly programmed, combines the advantages of a game paddle and a joystick. A mouse can move an object anywhere on a screen, and the position of the object can be very accurately controlled.

How Game Paddles Work

We have learned that a game paddle is actually a variable resistor controlled by a rotary knob. Since paddles are usually connected in pairs,

the Apple IIc/IIe has two analog inputs to which paddles can be connected. Before a program can read these inputs, it must set a timing circuit that is connected to them. This circuit can be set by accessing a soft switch situated at memory address $C070. When the timing circuit is set, the high bits of four other memory registers—registers $C064 through $C067—are each set to 1. Two of these registers (located at $C064 and $C065) correspond to the two game paddles that can be connected to the Apple IIc/IIe. These two game paddles are numbered Paddle 0 and Paddle 1.

If no game paddles or other I/O devices are connected to the Apple IIc/IIe's game port when Soft Switch $C070 is read, the high bits of memory registers $C064 through $C067 may remain set indefinitely. If, however, game paddles or other devices are connected to these inputs, the high bits of registers $C064 and $C067 will, after a period of time ranging from a fraction of a millisecond to about three milliseconds, change back to 0—and stay there. The exact amount of time that it takes for each of these bit changes to occur depends upon the resistance applied to the circuit that corresponds to each affected memory register. Therefore, if Game Paddle 0 is turned far to the left when Soft Switch $C070 is accessed, it won't take long for the high bit of register $C064 to change from a 1 to a 0. If Paddle 0 is set at a high position, the bit change will take longer. Similarly, if Paddle 0 is turned to the left, the high bit of register $C065 will change from 1 to 0 very quickly, and turning the paddle to the right will slow down the bit change.

To read the states of Paddles 0 and 1, a program must first access Soft Switch $C070. Then the program must set up a timing loop and use this loop to determine how long it takes the high bits of memory registers $C064 and $C067 to drop from 1 to 0.

This procedure would be quite a job for a programmer. Fortunately, however, the Apple IIe and Apple IIc are provided wih a built-in subroutine that can take care of much of the work involved in reading a pair of game paddles. This routine is called PREAD, and its starting address is $FDB1E.

To use the PREAD subroutine, you have to store a number from 0 through 3 in your microprocessor's X register. If you're using the PREAD routine to read a pair of game paddles, you can store either a 0 or a 1 in the X register, depending upon which paddle you want to read. Then you can invoke the PREAD subroutine by doing a JSR to memory address $FB1E. When your computer returns from the subroutine, a number ranging from $00 to $FF will be stored in your microprocessor's Y register. That number will reflect the state of the game paddle that you want to read.

As you might guess, timing is critical when you're writing this kind of program. There's also a further complication in this particular programming situation. Because it can take as long as three milliseconds for the high bits of registers $C064 through $C067 to drop from 1 to 0, there must be a delay of at least three milliseconds between the reading of register $C064 and the reading of register $C065. In other words, after you've read the state of one paddle, you must wait at least three milliseconds before you try to read the other one. This situation exists whether you write your own program for reading a pair of game paddles or use your computer's built-in PREAD routine.

How Joysticks Work

Game paddles aren't used much anymore in Apple programs, but the *joystick*—a descendant of the game paddle—is still very popular. There are many similarities between programs written for paddles and programs written for joysticks. In fact, many programs that were originally written for game paddles will also work with a joystick, and some programs written for joysticks will also work with game paddles.

As previously mentioned, joysticks and game paddles can be plugged into the same I/O ports on both the Apple IIc and the Apple IIe. Programs designed to be used with joysticks are written in exactly the same way as programs intended for use with game paddles. In a program designed to be used with a joystick, memory register $C064 is used to read the left-to-right motion of the stick, and memory register $C065 is used to read the stick's up-and-down motion. In other words, the horizontal axis of the joystick is treated as Paddle 0 and the vertical axis of the joystick is treated as Paddle 1 (although a few programs will reverse paddle axes).

When you want to read a joystick in an assembly-language program, you can either write your own routine (using registers $C070, $C064, and $C065) or use your computer's built-in PREAD routine. No matter which option you select, though, you have to be careful about timing. When you use a joystick in a program, you can read the setting of either axis repeatedly, as rapidly as you like. However, you must always program a delay of at least three milliseconds between the time you read one axis and the time you read the other.

Two other memory addresses that are often used in joystick programs are $C061, sometimes called Switch Input 0, and $C062, some-

times called Switch Input 1. These addresses can be used to read the two pushbuttons on a joystick controller or the OPEN-APPLE and CLOSED-APPLE keys on the Apple IIc/IIe keyboard. When Switch 0 on a joystick is pressed or the OPEN-APPLE key is pressed, bit 7 of $C061 goes from 1 to 0. When Switch 1 on a joystick is pressed or the CLOSED-APPLE key is pressed, bit 7 of $C062 goes from 1 to 0. The state of the switches on a joystick (or the state of the OPEN-APPLE and CLOSED-APPLE keys) can therefore be read by testing bit 7 (the sign bit) of memory registers $C061 and $C062 to determine whether the number in that location is greater than 127.

Using a Joystick in a Graphics Program

Before you type, assemble, and run an assembly-language program that illustrates how a joystick can be used in a low-resolution graphics program, let's take note of some facts about low-resolution graphics that were not covered in Chapter 12.

We learned in Chapter 12 that low-resolution graphics are a video graphics mode that can generate up to 16 colors in a screen display measuring 40 pixels wide by 48 pixels high. You may also remember that the column coordinates on a low-resolution screen map are laid out consecutively, but the row coordinates are not. Instead, like the row coordinates on all screen maps used by Apple II-series computers, the row coordinates used in low-resolution graphics are arranged in three groups of eight rows each, with the three groups of rows interleaved together.

To write programs in high-resolution graphics, it is helpful to understand this screen mapping system. Chapter 14 will cover high-resolution screen mapping in some detail. However, it is possible to write assembly-language programs using low-resolution graphics without being concerned with the intricacies of Apple IIc/IIe screen mapping. The designers of your Apple have provided a series of built-in subroutines designed to relieve you of much of the effort involved in low-resolution graphics programming:

- *CLRSCR* is a subroutine that will clear your computer's low-resolution screen to black. The CLRSCR routine starts at memory address $F832. To call it, you write a statement like this:

```
JSR $F832
```

If your computer is in low-resolution mode when you issue this command, the screen will be cleared. If it is in 40-column text mode, the screen will be filled with "@" characters, because the ASCII code of the character "@" is $00—the same number used for the color black in the low-resolution graphics mode!

- *SETCOL*, a subroutine that starts at memory address $F864, can set the color that will be used for plotting operations in low-resolution graphics. To use the SETCOL routine, you load the accumulator with the code number of the color to be plotted and then do a JSR to $F864. (The code numbers of all 16 of the colors used in low-resolution graphics were listed in Chapter 12.)
- *PLOT*, a subroutine that begins at memory address $F800, can be used to plot a pixel in any low-resolution color at any location on the low-resolution screen. Before the PLOT routine is called, a row coordinate ranging from 0 to 39 must be placed in the Y register, and a column coordinate ranging from 0 to 47 must be placed in the accumulator. A single pixel will then be displayed on the screen in the specified location. The color of the pixel can be determined by using the SETCOL subroutine before the PLOT routine is called. The column and row coordinates used by PLOT run in consecutive order, without using the interleaved system employed in do-it-yourself plotting routines.

SKETCHER: A Low-Resolution Joystick Program

Program 13-1 is an assembly-language program called SKETCHER. It was written using a Merlin Pro assembler, but it can also be typed and assembled using an Apple ProDOS assembler. It uses a number of the programming techniques that have been described in this chapter, plus a few others that will be covered before the chapter ends. You will need a joystick to run the program, of course. After you have run the program, we will examine it line by line.

Program 13-1
THE SKETCHER PROGRAM

```
1 *
2 * SKETCHER
3 *
4  ORG $8000
5 *
6 * ADDRESSES OF SOFT SWITCH1ES
7 *
```

```
 8 TEXTOFF EQU $C050
 9 HIRESOFF EQU $C056
10 CARDOFF EQU $C00C
11 MIXEDOFF EQU $C052
12 PAGE2OFF EQU $C054
13 TIMER EQU $C070
14 SWITCH1 EQU $C061
15 SWITCH2 EQU $C062
16 *
17 * ADDRESSES OF BUILT-IN FIRMWARE ROUTINES
18 *
19 CLRSCR EQU $F832
20 SETCOL EQU $F864
21 PLOT EQU $F800
22 PREAD EQU $FB1E
23 *
24 * DEFINE CONSTANTS
25 *
26 PRODL EQU $0300
27 PRODH EQU PRODL+1
28 MPRL EQU PRODH+1
29 MPRH EQU MPRL+1
30 MPDL EQU MPRH+1
31 MPDH EQU MPDL+1
32 XCOORD EQU MPDH+1
33 YCOORD EQU XCOORD+1
34 XAXIS EQU YCOORD+1
35 YAXIS EQU XAXIS+1
36 *
37  JMP START
38 *
39 MULT16 LDA #0
40  STA PRODL
41  STA PRODH
42  LDX #16
43 SHIFT ASL PRODL
44  ROL PRODH
45  ASL MPRL
46  ROL MPRH
47  BCC NOADD
48  CLC
49  LDA MPDL
50  ADC PRODL
51  STA PRODL
52  LDA MPDH
53  ADC PRODH
54  STA PRODH
55 NOADD DEX
56  BNE SHIFT
57  RTS
58 *
59 * DELAY LOOP ROUTINE
60 *
61 WAIT LDX #$FF
62 LOOP DEX
63  BNE LOOP
64  RTS
```

```
 65 *
 66 * MAIN PROGRAM STARTS HERE
 67 *
 68 * PUT COMPUTER IN LOW-RES GRAPHICS MODE
 69 *
 70 START STA TEXTOFF
 71  STA HIRESOFF
 72  STA CARDOFF
 73  STA MIXEDOFF
 74  STA PAGE2OFF
 75 *
 76 * CLEAR SCREEN
 77 *
 78  JSR CLRSCR
 79 *
 80 * SET COLOR OF SCREEN DOT TO PINK
 81 *
 82  STA TIMER
 83  LDA #11 ;PINK
 84  JSR SETCOL
 85 *
 86 * READ JOYSTICK COORDINATES
 87 *
 88 RDSTICK EQU *
 89  JSR WAIT
 90  JSR WAIT
 91  LDX #0
 92  JSR PREAD
 93  STY XAXIS
 94 *
 95  JSR WAIT
 96  JSR WAIT
 97  JSR WAIT
 98  JSR WAIT
 99  LDX #1
100  JSR PREAD
101  STY YAXIS
102 *
103 * CHECK STATUS OF JOYSTICK SWITCHES
104 *
105  LDA SWITCH2
106  BMI START
107  LDA SWITCH1
108  BMI SKIP
109  LDA #0 ;BLACK
110  JSR SETCOL
111  JSR ERASE
112  LDA #11 ;PINK
113  JSR SETCOL
114 *
115 * PLOT SCREEN DOT
116 *
117 SKIP LDA #0
118  STA MPDH
119  STA MPRH
120  LDA XAXIS
121  STA MPDL
```

```
122   LDA #40
123   STA MPRL
124   JSR MULT16
125   LDA PRODH
126   STA XCOORD
127  *
128   LDA #0
129   STA MPDH
130   STA MPRH
131   LDA YAXIS
132   STA MPDL
133   LDA #48
134   STA MPRL
135   JSR MULT16
136   LDA PRODH
137   STA YCOORD
138  *
139   LDA YCOORD
140   LDY XCOORD
141   JSR PLOT
142   JMP RDSTICK
143 ERASE LDA YCOORD
144   LDY XCOORD
145   JSR PLOT
146   RTS
```

How the SKETCHER Program Works

The SKETCHER program starts at line 37, with a JMP instruction that skips to line 70. Beginning at line 70, five soft switches are set. These switches turn off your computer's high-resolution mode, its 80-column firmware, its mixed mode (so there will be no text at the bottom of the screen), and its auxiliary memory. Next, in line 78, the low-resolution screen is cleared to black with a JSR CLRSCR statement.

In line 82, the soft switch at $C070 is set, starting a three-millisecond countdown so that the game paddle (or joystick) inputs at $C064 and $C065 can be read. Next, a JSR SETCOL statement is used to set the color of the low-resolution pixel to pink.

The heart of the SKETCHER program is the segment that extends from line 88 to line 101. During this sequence, a subroutine called WAIT (which begins at line 61) is called several times to activate the three-millisecond delay that must be inserted between the reading of one joystick axis and the reading of the other. The remaining commands in the sequence are straightforward. In lines 91 through 93, the X register is loaded with a 0, and the monitor routine PREAD is used to read the X axis of the joystick. In lines 99 through 101, the X register is loaded with a 1, and the PREAD routine is used to read the state of the joy-

stick's Y axis. When this segment of code ends, the position of the joystick's X axis (expressed as a number ranging from $00 to $FF) is in a memory register labeled XAXIS, and the position of the joystick's Y axis (also expressed as a number ranging from $00 through $FF) is in a memory register labeled YAXIS.

In lines 105 through 113, the states of the trigger buttons on the joystick are checked to see if either button is being pressed. If the side button on the joystick (called Switch 1 in this program) is being pressed, the cursor will draw a line as it moves across the screen. If the top button on the joystick (Switch 2) is pressed, the screen will be cleared. If neither switch is pressed, movement of the joystick will cause the cursor to move around on the screen without drawing a line.

Lines 117 through 146 move the cursor and—if Switch 2 is being pressed—draw lines on the screen. In this sequence of code, a 16-bit multiplication routine is used to translate the numbers stored in XAXIS and YAXIS into two valid screen coordinates: a horizontal (or X) coordinate, ranging from 0 through 39, and a vertical (or Y) coordinate, ranging from 0 through 47.

In lines 117 through 137, an interesting shortcut is used to make this conversion. First, the value stored in XAXIS is multiplied by 40. Then the high-order byte of the resulting product is stored in a memory register labeled XCOORD.

Next, the value stored in YAXIS is multiplied by 48. Finally, the high-order byte of the product that results from this operation is stored in a memory register labeled YCOORD. Then these two coordinates are used to position the cursor on the screen.

This may sound like an odd way to calculate coordinates, but it makes a lot of sense in this application. Extracting the high byte of a 16-bit number and leaving behind the low byte of the number is like dividing the 16-bit number by 256. When the value stored in XAXIS (a number ranging from $00 to $FF) is multiplied by 40 and the product thus derived is divided by 256, XAXIS is converted into a number ranging from 0 through 39. That number—a legal screen coordinate—can then be stored in XCOORD.

In lines 116 through 125, when the value in YAXIS is multiplied by 48 and the resulting product is divided by 256, the value of YAXIS is converted into a legal screen coordinate ranging from 0 through 47. That number is then stored in YCOORD.

The plotting routine in the SKETCHER program is the short segment of code that extends from line 139 to line 146. In this segment of code, the value of YCOORD is stored in the accumulator, the value of

XCOORD is stored in the Y register, and the monitor subroutine PLOT is called. Then, in line 142, the program jumps to the line labeled RDSTICK, and another joystick reading and plotting sequence begins.

Disadvantages of Joysticks

When you run the SKETCHER program, you'll probably notice that it has a couple of minor shortcomings. For example, if your joystick is an auto-centering model (and most joysticks are), the screen cursor will always return to the center of the screen when the joystick is released. Some joysticks have a defeatable self-centering feature so you can switch off the auto-centering mechanism and eliminate this problem. If you have a joystick with non-defeatable self-centering, such as the model marketed by Apple for the IIc and the IIe, then your joystick will return automatically to the center position when you let it go.

If you move your joystick quickly when you run the SKETCHER program, you may notice another problem: gaps in the lines that the program draws on the screen. These gaps are caused by unavoidable delays in the SKETCHER program, especially the three-millisecond delay that has to be inserted between the reading of one joystick axis and the reading of the other.

The problems of auto-centering and gaps in lines can be solved in several ways. One way to take care of both problems simultaneously is to simplify the way in which a joystick is read. Instead of trying to keep track of the exact settings of a joystick at all times, you can use a program that merely reads the direction in which the joystick is being held. As illustrated in Program 13-2, such a program can be written quite easily, even in BASIC.

Program 13-2
JOYSTICK.BAS
A low-resolution joystick program

```
10  REM  **** JOYSTICK.BAS ****
20  REM  **** LOW-RESOLUTION JOYSTICK PROGRAM ****
30  GR : POKE  - 16302,0: CALL  - 1998: COLOR= 11: REM  TURN
    OFF MIXED MODE AND CLEAR TEXT WINDOW TO BLACK
40 XM =  PDL (0):X =  INT (XM / 6.4):YM =  PDL (1):Y =  INT
    (YM / 5.3): REM  CONVERT JOYSTICK READINGS INTO LOW-RES
    COLUMN AND ROW COORDINATES, AND PRINT A PIXEL AT
    MIDSCREEN
50  IF  PEEK (49250) > 127 THEN 30: REM  IF SWITCH 1 IS
    PRESSED, THEN RESTART
60  IF  PEEK (49249) > 127 THEN 80: REM  IF SWITCH 0 IS
    PRESSED, THEN DRAW A LINE
70  COLOR= 0: PLOT TX,TY: COLOR= 11: REM  OTHERWISE, ERASE
    AND REPLOT
```

```
75  REM  **** STICK-PLOTTING ROUTINES START HERE ****
80  PLOT X,Y:TX = X:TY = Y: REM  TX AND TY ARE USED FOR
      TEMPORARY STORAGE OF X AND Y COORDINATES
90  GOSUB 190: IF XP > XM - 5 THEN 120
100 X = X - 1: IF X < 0 THEN X = 0
110  GOTO 140
120  IF XP < XM + 5 THEN 140
130 X = X + 1: IF X > 39 THEN X = 39
140  IF YP > YM - 5 THEN 160
150 Y = Y - 1: IF Y < 0 THEN Y = 0
160  IF YP < YM + 5 THEN 50
170 Y = Y + 1: IF Y > 47 THEN Y = 47
180  GOTO 50
185  REM  **** SUBROUTINE FOR READING JOYSTICK ****
190 XP =  PDL (0):YP =  PDL (1): RETURN
```

Program 13-2, entitled JOYSTICK.BAS, works much like the assembly-language SKETCHER listing presented in Program 13-1. However, as you'll discover if you type and run the JOYSTICK.BAS program, it is not plagued by the two problems in the SKETCHER routine, holes in the lines and auto-centering.

Unfortunately, a price had to be paid for these advantages. The JOYSTICK.BAS program runs more slowly than the SKETCHER program, and it offers the user considerably less control over the position of the cursor. When you run the JOYSTICK.BAS program, you can't place the cursor anywhere you like on the screen. Instead, you can move it in only eight discrete directions: up, down, left, right, and the diagonals. And, you have no control over the speed at which the cursor moves.

You could speed up the JOYSTICK.BAS program, of course, by translating it into assembly language. That wouldn't do much, however, to increase your control over the cursor on the screen.

In some applications, this limitation of control over the speed and positioning of the cursor is not important. In arcade-style games, for example, it is rarely necessary—and sometimes not even desirable—to have absolute control over the cursor's direction and speed. However, in other applications, such as computer-art programs, a high degree of control over the cursor is important.

The Apple Mouse

The Apple mouse, as pointed out earlier in this chapter, combines the benefits of a course-charting joystick program with those of a program in which an absolute plotting system is used. Program 13-3, called MOUSKETCH, uses a mouse to draw pictures on a low-resolution screen. It works in much the same way as the SKETCHER program.

MOUSKETCH uses a mouse to move a blinking cursor around on the screen, and if the button on the mouse is pressed, the cursor will draw a line as it moves. There is only one button on an Apple mouse, so the button cannot be used to clear the screen. However, if either the OPEN-APPLE key or the CLOSED-APPLE key is pressed, the screen will clear and the program will start over.

If you own an Apple IIe, you'll need a mouse card installed in Auxiliary Slot 4 for the MOUSKETCH program to run properly as written. If the mouse card is installed in some other slot, you'll have to modify the program as described later in this chapter.

When you assemble and run the MOUSKETCH program, you may notice that it has two clear advantages over the SKETCHER program. First, because a mouse is not a self-centering device, the cursor does not return to the middle of the screen when you release the mouse. The second (and probably more important) advantage is that the MOUSKETCH program will almost never leave a gap in a line that it's drawing, no matter how rapidly you sweep the cursor across your screen.

Program 13-3
THE MOUSEKETCH PROGRAM

```
 1 *
 2 * MOUSKETCH
 3 *
 4  ORG $8000
 5 *
 6  JMP GO
 7 *
 8 SLOT EQU $FB
 9 XPSN EQU $3B8
10 YPSN EQU $438
11 BUTTON EQU $6B8
12 *
13 * MOUSE ENTRY OFFSET POINTS
14 *
15 SETMS EQU $12
16 READMS EQU $14
17 INITMS EQU $19
18 CLAMPMS EQU $17
19 *
20 * ADDRESSES OF SOFT SWITCHES
21 *
22 TEXTOFF EQU $C050
23 HIRESOFF EQU $C056
24 FWOFF EQU $C00C
25 MIXEDOFF EQU $C052
26 PAGE2OFF EQU $C054
27 *
28 * MOUSE FIRMWARE ROUTINES
29 *
30 CLRSCR EQU $F832
31 SETCOL EQU $F864
```

```
32 PLOT EQU $F800
33 *
34 * DEFINE CONSTANTS
35 *
36 PRODL EQU $0300
37 PRODH EQU PRODL+1
38 MPRL EQU PRODH+1
39 MPRH EQU MPRL+1
40 MPDL EQU MPRH+1
41 MPDH EQU MPDL+1
42 XCOORD EQU MPDH+1
43 YCOORD EQU XCOORD+1
44 XAXIS EQU YCOORD+1
45 YAXIS EQU XAXIS+1
46 *
47 * SIGNATURE BYTES
48 *
49 CN EQU $C4
50 NO EQU $40
51 *
52 * SELF-MODIFYING ROUTINE
53 *
54 SETFW JMP $0000
55 *
56 * MAIN PROGRAM STARTS HERE
57 *
58 GO EQU *
59  JSR INIT ;INITIALIZE SCREEN
60  LDA #$00
61  STA SLOT
62  LDA #CN
63  STA SLOT+1
64  LDY #INITMS
65  JSR CALLFW
66  JSR CLAMP
67  LDY #SETMS
68  LDA #$01 ;SET PASSIVE MODE
69  JSR CALLFW
70 *
71 * MAIN LOOP
72 *
73 DOIT EQU *
74  LDY #READMS
75  JSR CALLFW
76  JSR DRAW
77  JMP DOIT
78 *
79 * SUBROUTINES START HERE
80 *
81 CALLFW EQU *
82  PHA
83  LDA (SLOT),Y
84  LDX #CN
85  LDY #NO
86  STA SETFW+1
87  STX SETFW+2
88  PLA
```

```
 89  JSR SETFW
 90  RTS
 91 *
 92 * DRAW SCREEN DOT
 93 *
 94 DRAW EQU *
 95 *
 96 * CHECK MOUSE BUTTON AND APPLE KEYS
 97 *
 98  LDA $C061
 99  BMI GO
100  LDA $C062
101  BMI BEGIN
102  LDX SETFW+2
103  LDA BUTTON,X
104  BMI LEAP
105  LDA #0 ;BLACK
106  JSR SETCOL
107  JSR PLOTIT
108  LDA #11 ;PINK
109  JSR SETCOL
110 *
111 * NOW DRAW DOT
112 *
113 LEAP LDX SETFW+2
114  LDA XPSN,X
115  STA XAXIS
116  LDA YPSN,X
117  STA YAXIS
118 *
119  LDA #0
120  STA MPDH
121  STA MPRH
122  LDA XAXIS
123  STA MPDL
124  LDA #40
125  STA MPRL
126  JSR MULT16
127  LDA PRODH
128  STA XCOORD
129 *
130  LDA #0
131  STA MPDH
132  STA MPRH
133  LDA YAXIS
134  STA MPDL
135  LDA #48
136  STA MPRL
137  JSR MULT16
138  LDA PRODH
139  STA YCOORD
140 *
141 PLOTIT LDA YCOORD
142  LDY XCOORD
143  JSR PLOT
144  RTS
145 *
```

```
146 CLAMP EQU *
147  LDA #$00
148  STA $478
149  STA $578
150  STA $5F8
151  LDA #$FF
152  STA $4F8
153  LDY #CLAMPMS
154  LDA #0
155  JSR CALLFW
156  LDY #CLAMPMS
157  LDA #1
158  JSR CALLFW
159  RTS
160 *
161 MULT16 LDA #0
162  STA PRODL
163  STA PRODH
164  LDX #16
165 SHIFT ASL PRODL
166  ROL PRODH
167  ASL MPRL
168  ROL MPRH
169  BCC NOADD
170  CLC
171  LDA MPDL
172  ADC PRODL
173  STA PRODL
174  LDA MPDH
175  ADC PRODH
176  STA PRODH
177 NOADD DEX
178  BNE SHIFT
179  RTS
180 *
181 * PUT COMPUTER IN LOW-RES MODE
182 *
183 INIT STA TEXTOFF
184  STA HIRESOFF
185  STA FWOFF
186  STA MIXEDOFF
187  STA PAGE2OFF
188 *
189 * CLEAR SCREEN
190 *
191  JSR CLRSCR
192 *
193 * SET COLOR OF SCREEN DOT
194 *
195  LDA #11 ;PINK
196  JSR SETCOL
197  RTS
```

A Close Look at the Apple Mouse

The MOUSKETCH program begins at line 6 with a jump to line 58, the actual start of the program. At line 59, a subroutine labeled INIT is

called, and the low-resolution screen is initialized in much the same way as in the SKETCHER program. Then, in lines 59 through 69, the firmware that comes with the Apple mouse is initialized.

To understand this initialization process, it helps to know something about the operation and design of the Apple mouse. A mouse is a "smart" I/O device; it comes with a complex set of firmware that makes it far more intelligent than a simple resistor-based device such as a joystick or a game paddle.

One difference between the Apple mouse and a simple joystick is that the Apple mouse can be operated in a number of different modes:

- In *movement interrupt mode*, the mouse generates an interrupt each time it is moved. A program will then read data generated by the mouse only when that data changes, rather than keeping constant surveillance over the mouse to determine whether it has been moved. This mode can be useful in high-performance programs when processing time is at a premium. The intricacies of the movement interrupt mode are, however, beyond the scope of the introductory material presented in this chapter.
- In *button interrupt mode*, an interrupt is generated when the mouse button is pressed.
- *Movement/button interrupt mode* combines the two modes just described.
- *Screen refresh interrupt mode* examines mouse data every sixtieth of a second, during the vertical blank interval that takes place between video screen refreshes. This mode can eliminate the flickering that sometimes results when objects are moved around on a screen during the non-blank part of a video cycle.
- *Passive* or *transparent mode* uses an interrupt system built into the mouse firmware to update mouse data automatically. The transparent mode is the simplest mouse operating mode available, and it is the mode that we will concentrate on during the rest of this chapter.

Subroutines Provided With the Apple Mouse

To determine which mode will be used for mouse operations, a value called a *mode byte* must be loaded into your microprocessor's accumulator. Then a JSR must be done to a subroutine called SETMOUSE, which is built into the firmware that operates the Apple mouse. Before the SETMOUSE routine is invoked, however, a preliminary routine called INITMOUSE must be called.

In addition to SETMOUSE and INITMOUSE, a number of other important mouse-related subroutines are built into the firmware that runs the mouse. To help programmers access these subroutines, the mouse firmware also contains a table that points to the entry address of each routine. In a mouse-equipped Apple IIe or IIc, this table occupies memory addresses $Cn12 through $Cn19, with the variable *n* equating to the number of the slot where the mouse is installed. In an Apple IIc, this slot is always Slot 4. In an Apple IIe, the slot number can vary, depending upon which expansion slot has been used for the mouse card.

Table 13-1 is a list of address pointers that are built into the Apple mouse firmware. In assembly-language programs, these pointers can be used to locate the low bytes of the starting addresses of mouse-related subroutines. The functions of these routines will be described later in this chapter.

In each address listed in Table 13-1, the high byte is always $Cn, with *n* equating to the mouse's slot number. For example, if the mouse connected to your computer uses Slot 4, you can calculate the address for the routine INITMOUSE by adding the contents of address $C419 to the literal value $C400. You can determine the starting address of the SETMOUSE routine by adding the contents of address $C412 to the literal value $C400.

Once you know the starting address of a mouse routine you want to use, you can call the routine by storing certain values in your microprocessor's X and Y registers and then jumping to the mouse subroutine

Table 13-1. Mouse Subroutine Low-Byte Address Table

The following addresses hold the low bytes of the starting addresses of mouse firmware routines:

$Cn12	SETMOUSE
$Cn13	SERVMOUSE
$Cn14	READMOUSE
$Cn15	CLEARMOUSE
$Cn16	POSMOUSE
$Cn17	CLAMPMOUSE
$Cn18	HOMEMOUSE
$Cn19	INITMOUSE

that you want to call. These are the values that must be stored in the X and Y registers before any mouse routine (except the SERVEMOUSE routine) can be called:

- The value $Cn (with *n* equating to the mouse's slot number) must be stored in the X register.
- The value $n0 (with *n* equating to the mouse's slot number) must be stored in the Y register.

Once the X and Y registers have been loaded in this fashion, the desired subroutine can be called.

In lines 49 and 50 of the MOUSKETCH program, two constants (labeled CN and N0) are assigned the values $C4 and $40 respectively. If you own an Apple IIe and have a mouse card installed in a slot other than Slot 4, you can alter these values in the program to reflect the slot in which your card is installed.

Calling the INITMOUSE Routine

Before you can use the Apple mouse, you must initialize it by calling the INITMOUSE routine. In the MOUSKETCH program, INITMOUSE is called in lines 60 through 65. In lines 59 through 63, the value $C400 is loaded into a pair of 8-bit Page Zero registers labeled SLOT and SLOT+1. Then, using the offset table defined in lines 15 through 18, the offset pointer for the INITMOUSE routine (that is, the literal value 19) is loaded into the 6502B/65C02 Y register. Next, a JSR instruction is used to jump to a subroutine labeled CALLFW (an abbreviation for "call firmware"). This subroutine starts at line 61.

When the CALLFW routine is called, the contents of the accumulator are pushed onto the stack for safekeeping. Then, in line 83, an interesting operation takes place. With the help of indirect indexed addressing, the value of the Y register—which in this case is the literal number 19—is added to the contents of memory address $C400. The number resulting from this calculation equates to a memory address. That address, as you can see by examining Table 13-1, is $C419, which holds the low byte of the starting address of INITMOUSE. In other words, in line 83 the low byte of the starting address of the INITMOUSE routine is loaded into the accumulator.

Next, in lines 84 and 85 of the MOUSKETCH program, the X register is loaded with the value $Cn and the Y register is loaded with the value $n0. In lines 86 and 87, a complex programming technique called *address modification* is used to JSR to the INITMOUSE routine. Though difficult to explain (and perhaps even more difficult to grasp),

address modification is often used in assembly-language programming because it can lead to significant savings in both memory and processing time.

In line 86, the value in the accumulator—which is the low byte of the starting address of the INITMOUSE routine—is stored in a memory address labeled SETFW+1. In line 87 the value of the X register (the value $Cn) is stored in an address labeled SETFW+2.

To find these three memory addresses—SETFW, SETFW+1, and SETFW+2—move back to line 54 of the MOUSKETCH program; they're all there.

Line 54 contains a three-byte statement: JMP $0000. That's a three-byte instruction because it contains a one-byte instruction and a two-byte operand. If you break down the three bytes of the statement, you'll see that, when the MOUSKETCH program is assembled into machine language, the machine-language equivalent of the mnemonic jump will be stored in the memory address labeled SETFW. The value $00 will be stored in each of the two memory registers that will follow memory address SETFW. In other words, memory address SETFW+1 and memory address SETFW+2 will each hold a 0 when the MOUSKETCH program is assembled into machine language.

But the 0's stored in SETFW+1 and SETFW+2 at assembly time are actually dummy values. In lines 86 and 87, as we have just seen, new values are stored in these two memory addresses. In fact, by the time the subroutine labeled CALLFW ends, the starting address of the INITMS routine has replaced the dummy 0's originally stored in SETFW+1 and SETFW+2.

Once all this is done, the original value of the A register is restored from the stack, and a JSR instruction is used to jump to memory address SETFW. These two operations take place in lines 88 and 89.

When the statement JSR SETFW is encountered in line 89, the address of the *next* instruction in the program is pushed onto the stack. (This always happens when a JSR instruction is used in a program.) Next, the program jumps first to line 54 and then to the address now stored in SETFW+1 and SETFW+2—which is, as we have seen, the starting address of the INITMS subroutine. The INITMS subroutine ends with an RTS instruction, which, like all RTS instructions, will pull its return address off the stack. In this case, the return address that is retrieved from the stack will be the address of another RTS instruction—the one at line 90 of the MOUSKETCH program. (The address of this RTS instruction was pushed onto the stack at line 89.) Once the program finds its way back to the RTS instruction in line 90, that instruc-

tion will end the CALLFW routine. The program will then return to where it was before the CALLFW subroutine was called. In other words, the RTS instruction at line 90 will move the program back to line 66.

At line 66, the MOUSKETCH program jumps to a subroutine labeled CLAMP. This routine, which starts at line 146, is used to set the minimum and maximum values that can be returned by the X axis and the Y axis of the Apple mouse. When the Apple mouse is first initialized, the minimum and maximum values that can be returned for the values of both X and Y range from $00 to $3FF. In the MOUSKETCH program, the CLAMP subroutine is used to change the maximum values of both X and Y to $FF—the same values that are returned by Apple-compatible joysticks and game paddles. In the MOUSKETCH program, the boundaries for mouse data were changed to these new values so that routines originally written for the SKETCHER program could also be used in MOUSKETCH. This procedure saved considerable programming time when MOUSKETCH was being written.

In the CLAMP routine, a mouse firmware subroutine called CLAMPMS is used to set the new maximum and minimum values of the mouse's X and Y positions. Before the CLAMPMOUSE routine is called, the accumulator must be loaded with either a 0 or a 1. If a 0 is in the accumulator when CLAMPMOUSE is called, then CLAMPMOUSE will set new values for the mouse's X coordinates. If a 1 is in the accumulator, CLAMPMOUSE will change the limits of the mouse's Y coordinates.

When CLAMPMOUSE is executed, the contents of four memory addresses become the new high and low boundaries of the screen coordinates of the mouse. These four addresses and their functions are

```
$478    Low byte of new low boundary
$4F8    Low byte of new high boundary
$578    High byte of new low boundary
$5F8    High byte of new high boundary
```

During the CLAMP subroutine, CLAMPMOUSE is called twice—once for each mouse coordinate. Then the program returns to line 67, where a routine called SETMOUSE is called. In the MOUSKETCH program, SETMOUSE is used to put the mouse into passive mode.

Before the SETMOUSE routine is called, a value called a mode byte must be stored in the accumulator. Table 13-2 lists the mode bytes that can be used with the SETMOUSE subroutine.

Table 13-2. Mode Bytes Used With the SETMOUSE Routine

$00	Turn mouse off
$01	Set transparent mode
$03	Set movement interrupt mode
$05	Set button interrupt mode
$07	Set movement/button interrupt mode
$08-$0F	Bytes used to set screen refresh interrupt modes for the Apple IIc mouse

After the CLAMP subroutine is called, the MOUSKETCH program moves to its main loop, labeled DOIT, which begins at line 73. In the DOIT loop, a mouse firmware routine called READMOUSE is used to read the status of the joystick. Then the program jumps to a subroutine labeled PLOTDOT, which begins at line 94. The PLOTDOT routine works almost exactly like the RDSTICK routine in the SKETCHER program; it checks the OPEN-APPLE and CLOSED-APPLE keys to see whether the screen should be cleared and then checks the mouse button to see whether a line is to be drawn as the cursor moves. When the PLOTDOT subroutine ends, a JMP instruction is used to start the DOIT loop again.

14

Apple Graphics

If you know BASIC—even a little BASIC—you can write some fairly impressive programs in low-resolution graphics. To create eye-catching, fast-action, high-resolution programs, though, you almost have to use assembly language. In this chapter you'll learn the principles of both high-resolution and double high-resolution graphics. You'll also have an opportunity to type, assemble, and execute two high-resolution programs. The first program will allow you to type characters on your computer screen when your Apple is in high-resolution mode. The second program will enable you to type headline-size characters on a high-resolution screen. These characters will be displayed on the screen in color, if you have a color monitor, and they'll be four times as large as the screen characters that your computer normally displays.

These two programs will make sense once you have a fundamental understanding of how the Apple IIc and the Apple IIe generate their high-resolution screen displays.

Figure 14-1 is a screen map of High-Resolution Display Page 1, the block of RAM most often used as screen memory in the Apple IIc and the Apple IIe. This block of RAM extends from memory address $2000 to memory address $3FFF.

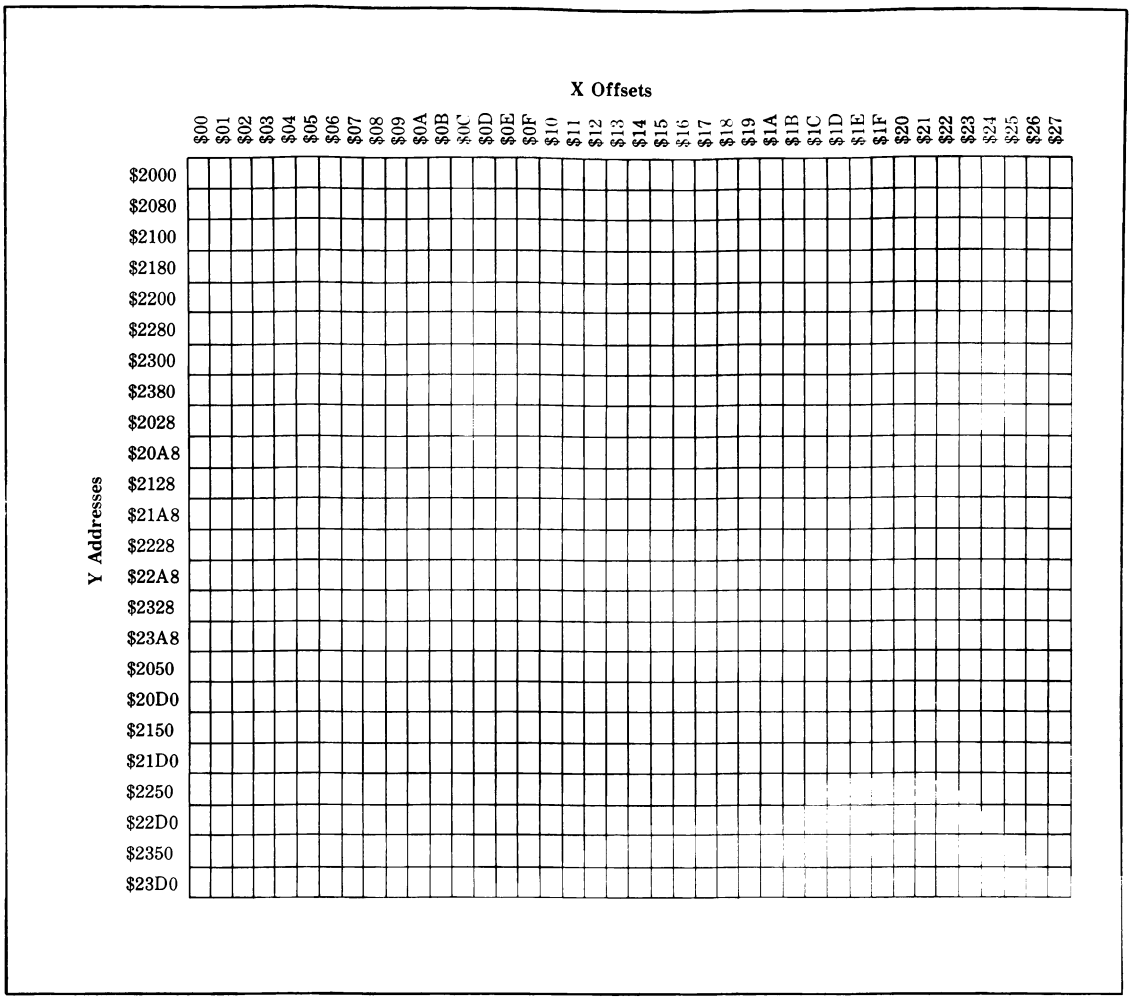

Figure 14-1. A high-resolution graphics screen map

Notice that the screen map illustrated by Figure 14-1 is quite similar to the 40-column text map illustrated in Chapter 12. The high-resolution map in Figure 14-1, like the text-display map that was shown in Chapter 12, is made up of 1024 rectangles arranged in a grid measuring 40 columns wide by 24 columns deep. The same kind of system is used on both maps for locating individual rectangles. On each map, the first byte of each horizontal row is identified with a two-byte starting address, and each vertical column is identified with a one-byte offset. To pinpoint the location of a given rectangle on either map, you add a column offset, or X offset, to a row address or Y address. The result will be the starting address of the rectangle being accessed.

The row addresses listed down the side of Figure 14-1 are arranged in exactly the same way as the Y addresses that were listed on the 40-column text map presented in Chapter 12. The Y addresses in Figure 14-1, like those of the text map in Figure 12-1, are arranged in three groups of eight rows each. On the high-resolution map in Figure 14-1, the first group of row addresses extends from memory address $2000 to memory address $2380, and the second group extends from $2028 to $23A8. The third group of addresses extends from $2050 to $23D0.

At first glance, the layouts of these two maps look exactly alike. However, there is one important difference in the way the two maps are interpreted in computer programs. The text map in Figure 12-1 is just what it looks like: a map of 1024 bytes of data. However, the 1024-square map in Figure 14-1 actually represents 8192 bytes of data—or $2000 bytes in hexadecimal notation. The next section will explain how 8912 bytes can be fit into 1024 screen squares.

How the Apple IIc/IIe Creates a Screen Display

When you want the Apple IIc/IIe to display a character on a 40-column text screen, you can simply store an Apple ASCII code number for that character on the screen map illustrated in Figure 12-1. Your Apple IIc or IIe will then convert that character into a certain pattern of dots and display that pattern in the appropriate position on the screen.

Figure 14-2 illustrates the dot pattern generated and displayed by the Apple IIc/IIe operating system when the key for the letter A is pressed on the computer keyboard.

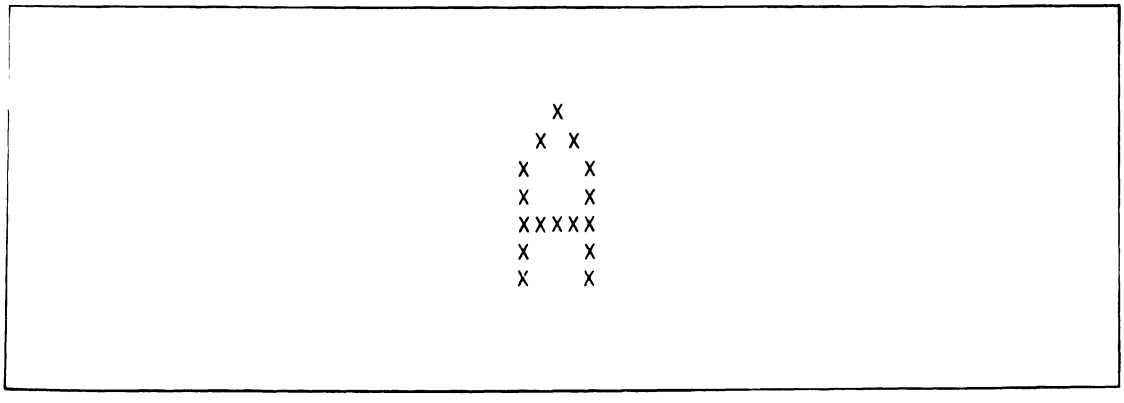

Figure 14-2. Screen dot pattern for the letter A

When your computer is in 40-column text mode, each rectangle on its screen map represents a single character code, or just one byte of data. But, when this byte of data is passed along to the Apple IIc/IIe character generating circuitry, it is converted into eight bytes (64 bits) of data. These eight bytes are then used to create a dot pattern on the screen.

When your computer is in high-res mode, however, it generates what is known as a bit-mapped screen display. When bit-mapping is used to create an Apple IIc/IIe screen display, the computer programmer has direct control over each dot that is displayed on the screen. Instead of storing one byte in a screen-map location and leaving it up to the computer to convert that byte into a bit pattern, the programmer must store eight bytes of data in *each* of the 1064 rectangles shown on the memory map. Bit-mapping a high-resolution screen, therefore, requires eight times 1024 bytes, or 8192 bytes. (That's $2000 bytes in hexadecimal notation.)

Since 8192 bytes are needed to bit-map a high-res screen, and since there are only 1024 cells on the map in Figure 14-1, it is obvious that some kind of trick must be used before the map in Figure 14-1 can be used to map a high-resolution screen. Since eight lines of dots on a high-resolution screen occupy the same space on the display as one row of characters, the total number of dot lines on a high-res display is eight times 24, or 192.

To display all of these rows on a high-resolution screen, the subset consisting of all of the first lines of dots in the first group of eight rows on the high-res screen map is stored in the first 1024 bytes of the block of memory being used for a high-resolution display. The subset consisting of all of the second lines of dots is stored in the second 1024 bytes of screen memory—and so on, for a total of eight times 1024, or 8192 bytes. In other words, each block of 1024 bytes in the high-resolution display page contains one line of dots out of every group of eight rows.

There is a short (if not simple) equation that you can use to locate the exact memory address of any byte on a high-resolution screen. Program 14-1 is a short BASIC program that contains this equation. In this program, Q stands for "quotient," R stands for "remainder," and VP stands for the vertical position of a dot on a screen, expressed as a line number ranging from 0 to 192. When the equation in the program is solved, it will yield the value of YP, which stands for "Y position." In this equation, the Y position is the starting address of the line on the screen map that contains the desired byte. Once this value has been determined, you can add the byte's X offset to the value of YP. The result of this calculation is the byte's exact memory address.

Program 14-1
Formula for Locating a Line Number on a High-Resolution Screen*

```
10 Q1 = INT (VP / 8):R1 = VP - Q1 * 8
20 Q2 = INT (Q1 / 8):R2 = Q1 - Q2 * 8
30 YP = 8192 + Q2 * 40 + R2 + 128 + R1 * 1024
```

* Adapted from a program in *Graphically Speaking*, a book by Mark Pelczarski (Softalk Books, 830 Fourth Avenue, Geneva, Illinois 60134, 1983). Used by permission.

If you wanted to write a high-resolution graphics program in BASIC, you could use an equation like the one in Program 14-1 every time you wanted to plot a dot on the screen. It would make much more sense, however, to use the equation to set up a table of all 192 Y addresses on a high-resolution screen. Then your program could locate the Y address of a screen line at any time by simply consulting that table.

Program 14-2
Program for Creating a Y-Address Lookup Table*

```
10  FOR VP = 0 TO 191
20 Q1 = INT (VP / 8):R1 = VP - Q1 * 8
30 Q2 = INT (Q1 / 8):R2 = Q1 - Q2 * 8
40 YP = 8192 + Q2 * 40 + R2 + 128 + R1 * 1024
50  POKE 28672 + VP, INT (YP / 256)
60  POKE 28864 + VP,YP -  INT (YP / 256) * 256
70  NEXT VP
```

* Adapted from a program in *Graphically Speaking*, a book by Mark Pelczarski (Softalk Books, 830 Fourth Avenue, Geneva, Illinois 60134, 1983). Used by permission.

Program 14-2 is a BASIC program that creates a table of Y addresses. The program then stores that table in a segment of RAM beginning at memory address $7000 (28672 in decimal notation). The low byte of each Y address is stored in the block of memory that begins at $7000, and the high byte of each Y address is stored in a second block of memory that begins at $70C0 (28864 in decimal notation). This strange-sounding program actually makes using the table easier, since the offset that fetches the high byte of a Y address can also be used to fetch the low byte.

When you use a program like Program 14-2 to create a table, you can store the table on a disk as a binary file with a ProDOS command such as

```
BSAVE TABLE,A28672,L384
```

As you may know, the number following the A in this command is the address where you want to store the table in memory. The number

after the L is the length, in bytes, of the block of data that you want to store. This technique can be used for storing machine-language programs on a disk, as well as for storing data. Of course, once a program or data table has been stored on a disk, it can be easily retrieved from the disk and incorporated into other machine-language programs. Further details on this process can be found in Apple's *ProDOS Technical Manual* and other ProDOS manuals.

You may type and run Program 14-2 if you like, but there's really no need to. The same routine, translated into assembly language, appears in the next program we'll be examining. That program, called PRNTCHRS, is presented in Program 14-3. PRNTCHRS was created using a Merlin Pro assembler. With minor modifications, it will also work when typed and run using an Apple ProDOS assembler.

Program 14-3
The PRNTCHRS Program

```
 1 *
 2 * PRNTCHRS.S
 3 *
 4  ORG $6FFD
 5 *
 6  JMP START
 7 *
 8 KYBD EQU $C000
 9 STROBE EQU $C010
10 *
11 TEMPLO EQU $06
12 TEMPHI EQU TEMPLO+1
13 CBASLO EQU TEMPHI+1
14 CBASHI EQU CBASLO+1
15 TABPTR EQU $4A
16 *
17 FILVAL EQU $300
18 TABSIZ EQU FILVAL+1
19 QUOT1 EQU TABSIZ+1
20 QUOT2 EQU QUOT1+1
21 REMDR1 EQU QUOT2+1
22 REMDR2 EQU REMDR1+1
23 YLINE EQU REMDR2+1
24 PRODL EQU YLINE+2
25 PRODH EQU PRODL+1
26 MPRL EQU PRODH+1
27 MPRH EQU MPRL+1
28 MPDL EQU MPRH+1
29 MPDH EQU MPDL+1
30 TOTAL EQU MPDH+1
31 XPSN EQU TOTAL+2
32 YPSN EQU XPSN+1
33 *
34 KILOBYTE EQU 1024
35 SCRTOP EQU $2000
36 *
```

```
37 PTRH EQU $8000
38 PTRL EQU $80C0
39 *
40 CHRTAB EQU $7000
41 *
42  HEX 2024282C3034383C2024282C3034383C
43  HEX 2125292D3135393D2125292D3135393D
44  HEX 22262A2E32363A3E22262A2E32363A3E
45  HEX 23272B2F33373B3F23272B2F33373B3F
46  HEX 0000000000000000808080808080808080
47  HEX 0000000000000000808080808080808080
48  HEX 0000000000000000808080808080808080
49  HEX 0000000000000000808080808080808080
50  HEX 2828282828282828A8A8A8A8A8A8A8A8
51  HEX 2828282828282828A8A8A8A8A8A8A8A8
52  HEX 2828282828282828A8A8A8A8A8A8A8A8
53  HEX 2828282828282828A8A8A8A8A8A8A8A8
54  HEX 5050505050505050D0D0D0D0D0D0D0D0
55  HEX 5050505050505050D0D0D0D0D0D0D0D0
56  HEX 5050505050505050D0D0D0D0D0D0D0D0
57  HEX 505050505050505 0D0D0D0D0D0D0D0
58  HEX D000000000000000001C1C1C1C001C1C
59  HEX 0036362436000000000123F3F123F3F12
60  HEX 000C3F033F303F0C000027170F3C3A39
61  HEX 00060902042A112E000C0C080C000000
62  HEX 00381C0E0E0E1C38000E1C3838381C0E
63  HEX 00082A1C3E1C2A08000C0C3F3F0C0C00
64  HEX 0000000000001C180C00003E3E000000
65  HEX 000000000001C1C006070381C0E0703
66  HEX 001E33333333331E003C363330303030
67  HEX 001E3F33380E3F3F001E3F303E303F1E
68  HEX 00383C36333F3030003F3F031F303F1E
69  HEX 001C06031F33331E003F3F30180C0C0C
70  HEX 001E3F331E333F1E001E3F333E303E1E
71  HEX 00001C1C00001C1C00001C1C00001C18
72  HEX 0C70381C0E1C387000003E3E003E3E00
73  HEX 00070E1C381C0E07001E3F33180C00
74  HEX 0C007F7F7F7F7F7F7F7F1E3F333F3F33
75  HEX 33001F3F331F333F1F001E3F3303333F
76  HEX 1E001F3F3333333F1F003F3F031F033F
77  HEX 3F003F3F031F1F0303001E3F033B333F
78  HEX 1E003333333F3F3333003F3F0C0C0C3F
79  HEX 3F00B0B0B0B0B3BF9E80333B1F0F1F3B
80  HEX 330003030303033F3F0063776B636363
81  HEX 63003333373F3B3333001E3F3333333F
82  HEX 1E001F3F333F1F0303001E3F23232B13
83  HEX 2E001F3F333F1F3B33001E3F031E303F
84  HEX 1E003F3F0C0C0C0C0C0C0033333333333F
85  HEX 1E0033333333331E0C006363636B6B77
86  HEX 630033331E0C1E3333003333331E0C0C
87  HEX 0C003F3F180C063F3F001F1F0303031F
88  HEX 1F0003070E1C387060001F1F1818181F
89  HEX 1F000C1E3F000000000080808080
90  HEX 80FFFF060C183000000000000001E303E
91  HEX 333E0003031F3333331F0000001E3303
92  HEX 331E0030303E3333333E0000001E331F
93  HEX 031E001C36061F0606060000001E3333
94  HEX 3E301E03031F33333333000000C000C0C
```

```
 95   HEX 0C0C0030003030303333 1E0303331B0F
 96   HEX 1B33000E0C0C0C0C0C1E0000003F5B5B
 97   HEX 5B5B0000001F333333330000001E3333
 98   HEX 331E0000001F33331F030300003E3333
 99   HEX 3E3030000001F330303030000001E031E
100   HEX 301E0006061F0606361C000000333333
101   HEX 333E0000003333331E0C0000006D6D6D
102   HEX 6D7E000000331E0C1E33000000333333
103   HEX 3E301E00003F180C063F001C1E060706
104   HEX 1E1C000C0C0C0C0C0C0C0C0C0E1E183818
105   HEX 1E0E000000281400000000FFFFFFFF
106  *
107  * SET UP TABLE OF SCREEN ROW COORDINATES
108  *
109  START LDY #0
110  YLOOP CPY #192
111   BCC CONT
112   JMP EXIT
113  *
114  * DIVIDE Y BY 8
115  *
116  CONT TYA
117   LSR
118   LSR
119   LSR
120   STA QUOT1
121  *
122  * GET REMAINDER OF DIVISION BY 8
123  *
124   TYA
125   AND #7
126   STA REMDR1
127  *
128  * DIVIDE QUOT1 BY 8
129  *
130   LDA QUOT1
131   LSR
132   LSR
133   LSR
134   STA QUOT2
135  *
136  * GET REMAINDER
137  *
138   LDA QUOT1
139   AND #7
140   STA REMDR2
141  *
142  * CALCULATE LOW BYTE OF Y ADDRESS
143  *
144   LDA #0
145   STA MPRH
146   STA MPDH
147   LDA QUOT2
148   STA MPRL
149   LDA #40
150   STA MPDL
151   JSR MULT16
152   LDA PRODL
```

```
153   STA TOTAL
154   LDA PRODH
155   STA TOTAL+1
156  *
157   LDA #0
158   STA MPRH
159   STA MPDH
160   LDA REMDR2
161   STA MPRL
162   LDA #128
163   STA MPDL
164   JSR MULT16
165   CLC
166   LDA PRODL
167   ADC TOTAL
168   STA TOTAL
169   LDA PRODH
170   ADC TOTAL+1
171   STA TOTAL+1
172  *
173   LDA #0
174   STA MPRH
175   LDA REMDR1
176   STA MPRL
177   LDA #<KILOBYTE
178   STA MPDL
179   LDA #>KILOBYTE
180   STA MPDH
181   JSR MULT16
182   CLC
183   LDA PRODL
184   ADC TOTAL
185   STA TOTAL
186   LDA PRODH
187   ADC TOTAL+1
188   STA TOTAL+1
189  *
190   CLC
191   LDA #<SCRTOP
192   ADC TOTAL
193   STA PTRL,Y
194   LDA #>SCRTOP
195   ADC TOTAL+1
196   STA PTRH,Y
197  *
198   INY
199   JMP YLOOP
200  *
201  EXIT EQU *
202  *
203  * INITIALIZE SCREEN DISPLAY
204  *
205  INIT STA $C050 ;TURN OFF TEXT MODE
206   STA $C052 ;TURN OFF MIXED MODE
207   STA $C057 ;TURN ON HI-RES MODE
208  *
209  * CLEAR SCREEN
210  *
```

```
211  LDA #0
212  STA FILVAL
213  LDA #$00
214  STA TABPTR
215  STA TABSIZ
216  LDA #$20
217  STA TABPTR+1
218  STA TABSIZ+1
219  JSR BLKFIL
220 *
221 * PRINT CHARACTER ON SCREEN
222 *
223  LDA #0
224  STA XVALUE
225  STA XPSN
226  STA YVALUE
227  STA YPSN
228 PRINTIT LDA KYBD
229  CMP #$80
230  BCC PRINTIT
231  STA STROBE
232  AND #$7F ;CLEAR HIGH BIT
233  STA CHAR
234  LDX XPSN
235  STX XVALUE
236  LDY YPSN
237  STY YVALUE
238  JSR PRNTCHRS
239 *
240 * RESET SCREEN COORDINATES
241 *
242  LDX XPSN
243  INX
244  CPX #40
245  BCC NEXT
246  CLC
247  LDA YPSN
248  ADC #8
249  CMP #185
250  BCC ITSOK
251  LDA #0
252 ITSOK STA YPSN
253  LDX #0
254 NEXT STX XPSN
255  JMP PRINTIT
256 *
257 * 16-BIT MULTIPLICATION ROUTINE
258 *
259 MULT16 LDA #0
260  STA PRODL
261  STA PRODH
262  LDX #16
263 SHIFT ASL PRODL
264  ROL PRODH
265  ASL MPRL
266  ROL MPRH
267  BCC NOADD
268  CLC
```

```
269  LDA MPDL
270  ADC PRODL
271  STA PRODL
272  LDA MPDH
273  ADC PRODH
274  STA PRODH
275 NOADD DEX
276  BNE SHIFT
277  RTS
278 *
279 * PRNTCHRS ROUTINE
280 *
281 PRNTCHRS LDA #<CHRTAB
282  STA CBASLO
283  LDA #>CHRTAB
284  STA CBASHI
285  LDA CHAR
286  LSR
287  LSR
288  LSR
289  LSR
290  LSR
291  CLC
292  ADC CBASHI
293  STA CBASHI
294  LDA CHAR
295  AND #$1F
296  ASL
297  ASL
298  ASL
299  CLC
300  ADC CBASLO
301  STA CBASLO
302  LDX #0
303 LOOP LDY YVALUE
304  LDA PTRL,Y
305  STA TEMPLO
306  LDA PTRH,Y
307  STA TEMPHI
308  TXA
309  TAY
310  LDA (CBASLO),Y
311  LDY XVALUE
312  STA (TEMPLO),Y
313  INC YVALUE
314  INX
315  CPX #8
316  BNE LOOP
317  RTS
318 *
319 * BLOCK FILL ROUTINE
320 *
321 BLKFIL LDA FILVAL
322  LDX TABSIZ+1
323  BEQ PARTPG
324  LDY #0
325 FULLPG STA (TABPTR),Y
326  INY
```

```
327   BNE FULLPG
328   INC TABPTR+1
329   DEX
330   BNE FULLPG
331 PARTPG LDX TABSIZ
332   BEQ FINI
333   LDY #0
334 PARTLP STA (TABPTR),Y
335   INY
336   DEX
337   BNE PARTLP
338 FINI RTS
339 *
340 CHAR DFB 0
341 XVALUE DFB 0
342 YVALUE DFB 0
343 *
```

Examining the PRNTCHRS Program

The PRNTCHRS program includes a long block of hexadecimal numbers that extends from line 42 to line 105. This section is actually a table of bit data that equates to a set of text characters similar to the ones built into the character-generator ROM of the Apple IIc. Predesigned character sets such as the one used in this program are provided with a number of graphics utility packages. For example, the character set in the PRNTCHRS program is included in *The Complete Graphics System* from Penguin Software and is used here by permission. If you have a graphics package that includes a character set, you can substitute that set for the one in the PRNTCHRS program. You can even write your own, if you like.

If you use a substitute character set, it should be loaded into RAM starting at memory address $7000. If you use a character set that starts at some other address, you can copy it onto a disk, load it back into memory starting at $7000, and then copy it back onto a disk as a new file with a starting address of $7000. Once you have a character set that starts at $7000, you can substitute it for the one in the PRNTCHRS program by deleting lines 42 through 105 of the program, loading your own character set into RAM, and simply running the program. If you do use a substitute character set, be sure *not* to delete line 40, which tells the program where to look for a character set in memory. If you leave that line out of the program, the character set you use will never be found.

The PRNTCHRS program begins at line 6 with a JMP instruction that hops over the character set and goes to line 109. In the section of the program that begins there and extends through line 201, a Y-

address lookup table is created and stored in a block of memory that starts at $8000. The routine that creates this table works exactly like the one in Program 14-2. The table-creating routine in the PRNTCHRS program is much longer than the routine in Program 14-2, but it works much faster because it is written in assembly language.

When the PRNTCHRS program finishes creating and storing its Y-address lookup table, it initializes a high-resolution graphics screen and clears High-Resolution Display Page 1 to black by storing a 0 in every byte on Display Page 1. To clear the screen, the program uses a block-fill subroutine that starts at line 321. This is a handy routine, since it can rapidly fill any block of RAM with any desired value. Using a variable called FILVAL (fill value) and fancy low-byte and high-byte addressing techniques, it does its job in two stages. First, it fills as many 1K pages (not display pages) as possible with the value of FILVAL. Then it fills in any remaining partial page.

The heart of the PRNTCHRS program is the section that extends from line 223 to line 228. The first thing that happens in this block of code is that the screen coordinates for the first character to be displayed—represented by the variables XVALUE and YVALUE—are set to 0. The first character that is typed will therefore be displayed in the upper-left corner of the screen.

After the screen coordinates are set, a commonly used keyboard-reading algorithm begins. Two important memory registers appear in this algorithm: memory address $C000, which is labeled KYBD, and memory address $C010, which is labeled STROBE. KYBD is a ROM address that can be checked to see whether a key has been pressed, and STROBE is a soft switch that can be used to clear a keyboard character after the character has been read.

When a key is pressed on an Apple IIc/IIe keyboard, several things happen. First, the ASCII code for the character that has been typed is stored in the accumulator. Then bit 7 of memory register $C000 is set. By keeping an eye on memory address $C000, a program can look to see whether a character has been typed. As soon as a character has been typed, its ASCII code can be fetched from the accumulator.

In the PRNTCHRS program, memory address $C000 is checked in lines 228 and 229. In line 228, the accumulator is loaded with the value of $C000; in line 229, that value is compared with the literal value #$80. If the content of $C000 is less than 80, then bit 7 of $C000 has not been set, indicating that no character has been typed. If no character has been typed, the program will loop back to line 228. If bit 7 of $C000 is clear, however, a character has been typed, so the program will clear

the keyboard strobe by accessing Soft Switch $C010. (A write operation is used for this operation in the PRNTCHRS program but, because of the peculiar way in which Apple II soft switches work, a read operation would also clear the strobe.) Clearing the strobe also clears the keyboard so another key can be read.

When a key has been pressed and the keyboard strobe has been cleared, the high bit of the value in the accumulator is cleared and the resulting value is stored in a variable called CHAR. The high bit of the accumulator is cleared because the character set in the PRNTCHRS program does not include any reverse, flashing, or alternate-font characters. In the Apple ASCII system, such characters are assigned ASCII code numbers that have their high bits set. Since there are no such characters in the PRNTCHRS character set, the high bit of the ASCII code in the accumulator is cleared.

Once the ASCII code for a character has been stored in the variable CHAR, the bit pattern corresponding to that character must be found in the character set that begins at memory address $7000. Then the character's bit pattern can be displayed on the screen.

The process of searching out a character's bit pattern starts at line 281. In lines 281 through 283, the starting address of the character-set table—which is called CHRTAB—is loaded into a pair of variables called CBASLO (character base-low) and CBASHI (character base-high). Then the accumulator is loaded with the ASCII code stored in the variable CHAR, and a series of LSR instructions is used to divide that value by 32. This operation is carried out because the character-set table fills four 1K pages of memory, and each page holds the bit data for 32 characters. Since the character table is arranged in ASCII-code order, dividing a character's ASCII number by 32 will tell us what page it is on in the character table. Then, since it takes eight bytes to form a character, the remainder multiplied by eight will give us the character's location on that page.

In lines 294 through 301, a neat trick is used to calculate the remainder that must be multiplied by eight. In this segment of the program, a logical AND operation is performed on the literal number $1F (decimal 31) and the ASCII code of the number stored in CHAR. Since 31 is one less than 32—or 0001 1111 in binary notation—using AND on 31 with any number will yield the remainder of a division of that number.

In lines 299 through 392, the quotient of the division problem we have just performed is added to the high byte of CHRTAB—the starting address of the character table. The remainder of our division by 32

is added to the low byte of CHRTAB and stored in CBASLO. When these two operations are complete, the high-order byte of the address of the character we're seeking will be stored in the variable CBASHI, and the low byte will be stored in the variable CBASLO.

In the next section of the PRNTCHRS program, the variables XPSN and YPSN (X position and Y position) are used to represent the current screen coordinates of the last character displayed on the screen, while XVALUE and YVALUE represent the X offset of the next character to be printed. Two constants, PTRL and PTRH (pointer-low and pointer-high), hold the starting addresses of the low-byte and high-byte Y-lookup tables stored at memory addresses $8000 and $80C0. Two more variables, TEMPLO and TEMPHI, hold the starting address of the bit data that will be used to display the desired character on the screen.

The rest of the PRNTCHRS program uses indirect addressing, along with loops that increment and decrement registers, to display the bit patterns of characters on a screen. Using these techniques, the program fetches consecutive bytes of character data from the character table at $7000. Then, using the same X offset and a series of different Y addresses, the program stores those eight bytes in a neat stack that forms a character on the screen.

Improving the PRNTCHRS Program

As you can see when you type, assemble, and run the PRNTCHRS program, it could use some improvement. It has no backspacing feature for making corrections, and it offers no easy way to place a character in any specific position on the screen. Furthermore, there is no cursor, so there's no way to tell exactly where a character will be displayed when it is placed on the screen.

If you run the PRNTCHRS program on a color television set or a color monitor, you'll encounter one other problem: the letters that the program produces won't be pure white. Instead, most of them have muddy-looking colors around the edges. Unfortunately, this "rainbow effect" is difficult to avoid when small characters are displayed on the Apple IIc/IIe screen during high-resolution mode. As we shall soon see, the problem is caused by the color-generating techniques that Apple II computers use when they are in high-resolution mode. For reasons that will be made clear later in this chapter, dots that are turned on in certain dot columns on the Apple screen sometimes show up in one color while dots in the next column are displayed in another color.

The problem does not affect the Apple IIc/IIe when it is in text

mode, which uses different techniques to generate a video display. If you want to mix text and graphics on a high-resolution screen, however, there are two simple ways to avoid the rainbow effect. You can use a monochrome monitor, or you can improve the resolution by displaying larger characters on the screen.

Program 14-4, called HEADLINES, offers at least partial solutions to the problems presented in this section. HEADLINES, written on a Merlin Pro assembler, includes a backspace feature for error correction, and the characters that it produces, because of their size, can be placed easily anywhere on the screen. Its giant-sized characters also solve the rainbow problem.

Program 14-4
The HEADLINES Program

```
 1 *
 2 * HEADLINES.S
 3 *
 4  ORG $6FFD
 5 *
 6  JMP START
 7 *
 8 KYBD EQU $C000
 9 STROBE EQU $C010
10 *
11 TEMPLO EQU $06
12 TEMPHI EQU TEMPLO+1
13 CBASLO EQU TEMPHI+1
14 CBASHI EQU CBASLO+1
15 TABPTR EQU $4A
16 *
17 FILVAL EQU $300
18 TABSIZ EQU FILVAL+1
19 QUOT1 EQU TABSIZ+1
20 QUOT2 EQU QUOT1+1
21 REMDR1 EQU QUOT2+1
22 REMDR2 EQU REMDR1+1
23 YLINE EQU REMDR2+1
24 PRODL EQU YLINE+2
25 PRODH EQU PRODL+1
26 MPRL EQU PRODH+1
27 MPRH EQU MPRL+1
28 MPDL EQU MPRH+1
29 MPDH EQU MPDL+1
30 TOTAL EQU MPDH+1
31 XPSN EQU TOTAL+2
32 YPSN EQU XPSN+1
33 COUNT EQU YPSN+1
34 OB EQU COUNT+1
35 NB1 EQU OB+1
36 NB2 EQU NB1+1
37 YCOUNT EQU NB2+1
38 *
```

```
39 KILOBYTE EQU 1024
40 SCRTOP EQU $2000
41 *
42 PTRH EQU $4000
43 PTRL EQU $40C0
44 *
45 CHRTAB EQU *
46 *
47  HEX 2024282C3034383C2024282C3034383C
48  HEX 2125292D3135393D2125292D3135393D
49  HEX 22262A2E32363A3E22262A2E32363A3E
50  HEX 23272B2F33373B3F23272B2F33373B3F
51  HEX 00000000000000000808080808080808
52  HEX 00000000000000000808080808080808
53  HEX 00000000000000000808080808080808
54  HEX 00000000000000000808080808080808
55  HEX 2828282828282828A8A8A8A8A8A8A8A8
56  HEX 2828282828282828A8A8A8A8A8A8A8A8
57  HEX 2828282828282828A8A8A8A8A8A8A8A8
58  HEX 2828282828282828A8A8A8A8A8A8A8A8
59  HEX 505050505050505ODODODODODODODODO
60  HEX 505050505050505ODODODODODODODODO
61  HEX 505050505050505ODODODODODODODODO
62  HEX 505050505050505ODODODODODODODO
63  HEX DOOOOOOOOOOOOOOOO1C1C1C1C001C1C
64  HEX 00363624360000000123F3F123F3F12
65  HEX 000C3F033F303F0C000027170F3C3A39
66  HEX 00060902042A112E000C0C080C000000
67  HEX 00381C0E0E0E1C38000E1C3838381C0E
68  HEX 00082A1C3E1C2A08000C0C3F3F0C0C00
69  HEX 0000000000001C180C00003E3E000000
70  HEX 0000000000001C1C006070381C0E0703
71  HEX 001E33333333331E003C363330303030
72  HEX 001E3F33380E3F3F001E3F303E303F1E
73  HEX 00383C36333F3030003F3F031F303F1E
74  HEX 001C06031F33331E003F3F30180C0C0C
75  HEX 001E3F331E333F1E001E3F333E303E1E
76  HEX 00001C1C00001C1C00001C1C00001C18
77  HEX 0C70381C0E1C387000003E3E003E3E00
78  HEX 00070E1C381C0E07001E3F33180C00
79  HEX 0C007F7F7F7F7F7F7F7F1E3F333F3F33
80  HEX 33001F3F331F333F1F001E3F3303333F
81  HEX 1E001F3F3333333F1F003F3F031F033F
82  HEX 3F003F3F031F1F0303001E3F033B333F
83  HEX 1E003333333F3F3333003F3F0C0C0C3F
84  HEX 3F00B0B0B0B0B3BF9E80333B1F0F1F3B
85  HEX 330003030303033F3F0063776B636363
86  HEX 63003333373F3B3333001E3F3333333F
87  HEX 1E001F3F333F1F0303001E3F23232B13
88  HEX 2E001F3F333F1F3B33001E3F031E303F
89  HEX 1E003F3F0C0C0C0C0C0C0033333333333F
90  HEX 1E00333333333331E0C006363636B6B77
91  HEX 630033331E0C1E3333003333331E0C0C
92  HEX 0C003F3F180C063F3F001F1F0303031F
93  HEX 1F0003070E1C387060001F1F1818181F
94  HEX 1F000C1E3F0000000000808080808O
95  HEX 80FFFF060C1830000000000001E303E
```

```
 96  HEX 333E0003031F3333331F0000001E3303
 97  HEX 331E0030303E3333333E0000001E331F
 98  HEX 031E001C36061F0606060000001E3333
 99  HEX 3E301E03031F33333333000000C000C0C
100  HEX 0C0C00300030303033331E0303331B0F
101  HEX 1B33000E0C0C0C0C0C1E0000003F5B5B
102  HEX 5B5B0000001F33333333000000E3333
103  HEX 331E0000001F33331F030300003E3333
104  HEX 3E303000001F330303030000001E031E
105  HEX 301E0006061F0606361C000000333333
106  HEX 333E0000003333331E0C0000006D6D6D
107  HEX 6D7E000000331E0C1E33000000333333
108  HEX 3E301E00003F180C063F001C1E060706
109  HEX 1E1C000C0C0C0C0C0C0C0C0E1E183818
110  HEX 1E0E000000281400000000FFFFFFFF
111 *
112 * SET UP TABLE OF SCREEN ROW COORDINATES
113 *
114 START LDY #0
115 YLOOP CPY #192
116  BCC CONT
117  JMP EXIT
118 *
119 * DIVIDE Y BY 8
120 *
121 CONT TYA
122  LSR
123  LSR
124  LSR
125  STA QUOT1
126 *
127 * GET REMAINDER OF DIVISION BY 8
128 *
129  TYA
130  AND #7
131  STA REMDR1
132 *
133 * DIVIDE QUOT1 BY 8
134 *
135  LDA QUOT1
136  LSR
137  LSR
138  LSR
139  STA QUOT2
140 *
141 * GET REMAINDER
142 *
143  LDA QUOT1
144  AND #7
145  STA REMDR2
146 *
147 * CALCULATE LOW BYTE OF Y ADDRESS
148 *
149  LDA #0
150  STA MPRH
151  STA MPDH
152  LDA QUOT2
153  STA MPRL
```

```
154   LDA #40
155   STA MPDL
156   JSR MULT16
157   LDA PRODL
158   STA TOTAL
159   LDA PRODH
160   STA TOTAL+1
161  *
162   LDA #0
163   STA MPRH
164   STA MPDH
165   LDA REMDR2
166   STA MPRL
167   LDA #128
168   STA MPDL
169   JSR MULT16
170   CLC
171   LDA PRODL
172   ADC TOTAL
173   STA TOTAL
174   LDA PRODH
175   ADC TOTAL+1
176   STA TOTAL+1
177  *
178   LDA #0
179   STA MPRH
180   LDA REMDR1
181   STA MPRL
182   LDA #<KILOBYTE
183   STA MPDL
184   LDA #>KILOBYTE
185   STA MPDH
186   JSR MULT16
187   CLC
188   LDA PRODL
189   ADC TOTAL
190   STA TOTAL
191   LDA PRODH
192   ADC TOTAL+1
193   STA TOTAL+1
194  *
195   CLC
196   LDA #<SCRTOP
197   ADC TOTAL
198   STA PTRL,Y
199   LDA #>SCRTOP
200   ADC TOTAL+1
201   STA PTRH,Y
202  *
203   INY
204   JMP YLOOP
205  *
206  EXIT EQU *
207  *
208  * INITIALIZE SCREEN DISPLAY
209  *
210  INIT STA $C050 ;TURN OFF TEXT MODE
211   STA $C052 ;TURN OFF MIXED MODE
```

```
212  STA $C057 ;TURN ON HI-RES MODE
213 *
214 * CLEAR SCREEN
215 *
216  LDA #0
217  STA FILVAL
218  LDA #$00
219  STA TABPTR
220  STA TABSIZ
221  LDA #$20
222  STA TABPTR+1
223  STA TABSIZ+1
224  JSR BLKFIL
225 *
226 * PRINT CHARACTER ON SCREEN
227 *
228  LDA #0
229  STA XVALUE
230  STA XPSN
231  STA YVALUE
232  STA YPSN
233 *
234 SHOWCHRS JSR PRINTIT
235  JSR RESET
236  JMP SHOWCHRS
237 *
238 PRINTIT LDA KYBD
239  CMP #$80
240  BCC PRINTIT ;NO KEY PRESSED; TRY AGAIN
241  STA STROBE
242  AND #$7F ;CLEAR HIGH BIT
243  CMP #8 ;LEFT ARROW PRESSED?
244  BEQ BACKUP
245  CMP #127 ;DELETE KEY
246  BNE JUMP
247 *
248 BACKUP JSR MOVEBACK
249  LDA #32 ;SPACE
250  STA CHAR
251  LDA XPSN
252  STA XVALUE
253  LDA YPSN
254  STA YVALUE
255  JSR PRNTCHRS
256  JSR MOVEBACK
257  RTS
258 *
259 MOVEBACK LDX XPSN
260  DEX
261  DEX
262  BPL LEAP
263  LDA YPSN
264  SEC
265  SBC #16
266  STA YPSN
267  LDX #38
268 LEAP STX XPSN
269  RTS
```

```
270 *
271 JUMP CMP #32 ;CONTROL KEY PRESSED?
272  BCC PRINTIT
273  STA CHAR
274  LDX XPSN
275  STX XVALUE
276  LDY YPSN
277  STY YVALUE
278  JSR PRNTCHRS
279  RTS
280 *
281 * RESET SCREEN COORDINATES
282 *
283 RESET LDX XPSN
284  INX
285  INX
286  CPX #40
287  BCC NEXT
288  CLC
289  LDA YPSN
290  ADC #16
291  CMP #185
292  BCC ITSOK
293  LDA #0
294 ITSOK STA YPSN
295  LDX #0
296 NEXT STX XPSN
297  RTS
298 *
299 * 16-BIT MULTIPLICATION ROUTINE
300 *
301 MULT16 LDA #0
302  STA PRODL
303  STA PRODH
304  LDX #16
305 SHIFT ASL PRODL
306  ROL PRODH
307  ASL MPRL
308  ROL MPRH
309  BCC NOADD
310  CLC
311  LDA MPDL
312  ADC PRODL
313  STA PRODL
314  LDA MPDH
315  ADC PRODH
316  STA PRODH
317 NOADD DEX
318  BNE SHIFT
319  RTS
320 *
321 * PRNTCHRS ROUTINE
322 *
323 PRNTCHRS LDA #<CHRTAB
324  STA CBASLO
325  LDA #>CHRTAB
326  STA CBASHI
327  LDA CHAR
```

```
328   LSR
329   LSR
330   LSR
331   LSR
332   LSR
333   CLC
334   ADC CBASHI
335   STA CBASHI
336   LDA CHAR
337   AND #$1F
338   ASL
339   ASL
340   ASL
341   CLC
342   ADC CBASLO
343   STA CBASLO
344   LDX #0
345   STX COUNT
346 LOOP LDY YVALUE
347   LDA PTRL,Y
348   STA TEMPLO
349   LDA PTRH,Y
350   STA TEMPHI
351   TXA
352   TAY
353   LDA (CBASLO),Y
354   STA OB
355 *
356 * SPLIT ORIGINAL BYTE INTO NB1 AND NB2
357 *
358   ASL OB
359 *
360   LDY #3
361 LOOPA ASL OB
362   ROL NB2
363   SEC
364   ROL NB2
365   DEY
366   BNE LOOPA
367   ASL OB
368   ROL NB2
369 *
370   LDY #3
371 LOOPB SEC
372   ROL NB1
373   ASL OB
374   ROL NB1
375   DEY
376   BNE LOOPB
377   SEC
378   ROL NB1
379 *
380   LDA NB1
381   ORA #$80
382   STA NB1
383   LDA NB2
384   ORA #$80
```

```
385  STA NB2
386 *
387 * BACK TO ORIGINAL PROGRAM NOW
388 *
389  LDA #2
390  STA YCOUNT
391 *
392 TWICE LDA NB1
393  LDY XVALUE
394  STA (TEMPLO),Y
395  INC XVALUE
396  LDA NB2
397  LDY XVALUE
398  STA (TEMPLO),Y
399  LDX XPSN
400  STX XVALUE
401  INC YVALUE
402 *
403  LDX YCOUNT
404  DEX
405  STX YCOUNT
406  BEQ HOP
407 *
408  LDX YVALUE
409  LDA PTRL,X
410  STA TEMPLO
411  LDA PTRH,X
412  STA TEMPHI
413  JMP TWICE
414 *
415 HOP LDX COUNT
416  INX
417  STX COUNT
418  CPX #8
419  BEQ SKIP
420  JMP LOOP
421 SKIP RTS
422 *
423 * BLOCK FILL ROUTINE
424 *
425 BLKFIL LDA FILVAL
426  LDX TABSIZ+1
427  BEQ PARTPG
428  LDY #0
429 FULLPG STA (TABPTR),Y
430  INY
431  BNE FULLPG
432  INC TABPTR+1
433  DEX
434  BNE FULLPG
435 PARTPG LDX TABSIZ
436  BEQ FINI
437  LDY #0
438 PARTLP STA (TABPTR),Y
439  INY
440  DEX
441  BNE PARTLP
```

```
442 FINI RTS
443 *
444 CHAR DFB 0
445 XVALUE DFB 0
446 YVALUE DFB 0
447 *
```

To understand how the HEADLINES program works, it helps to know something about the way the Apple IIc and the Apple IIe generate color displays on a monitor screen. Figure 14-3 shows how your computer's color-generating system works.

As you can see in Figure 14-3, each character displayed on a high-resolution screen is represented by eight bytes of bit-map data. However, only seven bits in each byte are actually displayed on the screen. The high-order bit is not displayed, but is used as a color bit. In each byte on a screen map, the color bit determines what colors will be used when the other bits in the same byte are displayed on the screen.

With the help of the color bit in each byte, six colors can be displayed on an Apple IIc/Apple IIe high-resolution screen: black, white, green, violet, orange, and blue.

If the bits in a byte do not have to be displayed in color, but only in black and white, the setting of the color bit in that byte does not matter. If two screen dots situated next to each other are turned on—that is, if the two adjacent bits that represent them are set to 1—both dots will be displayed in white. However, if two screen dots situated next to each other are turned off—if the two adjacent bits that represent them are cleared to 0—both dots will be displayed in black.

When colors other than black and white are to be displayed on a screen, bits are not set or cleared in pairs. Instead, only one bit is set for each two bits to be displayed on the screen. If an even-numbered bit is set and the bit on its right is cleared, then both bits will be displayed in the color indicated by the set bit. If an even-numbered bit is cleared and the bit on its right is set, both bits will still be displayed in color, but the colors will be different.

As we have seen, the color bit of a byte is also a factor in determining a screen color. If the color bit of a byte is set, the colors generated by the other bits in that byte will be orange and blue. If the color bit of a byte is cleared, the colors generated by the other bits in that byte will be green and violet.

Still another factor that is used to determine screen colors is the column position of a given byte on the screen. If a byte is in an even-numbered screen column, then even-numbered bits will produce one color and odd-numbered bits will produce another; if a byte is in an odd-numbered screen column, these colors will be reversed. This may

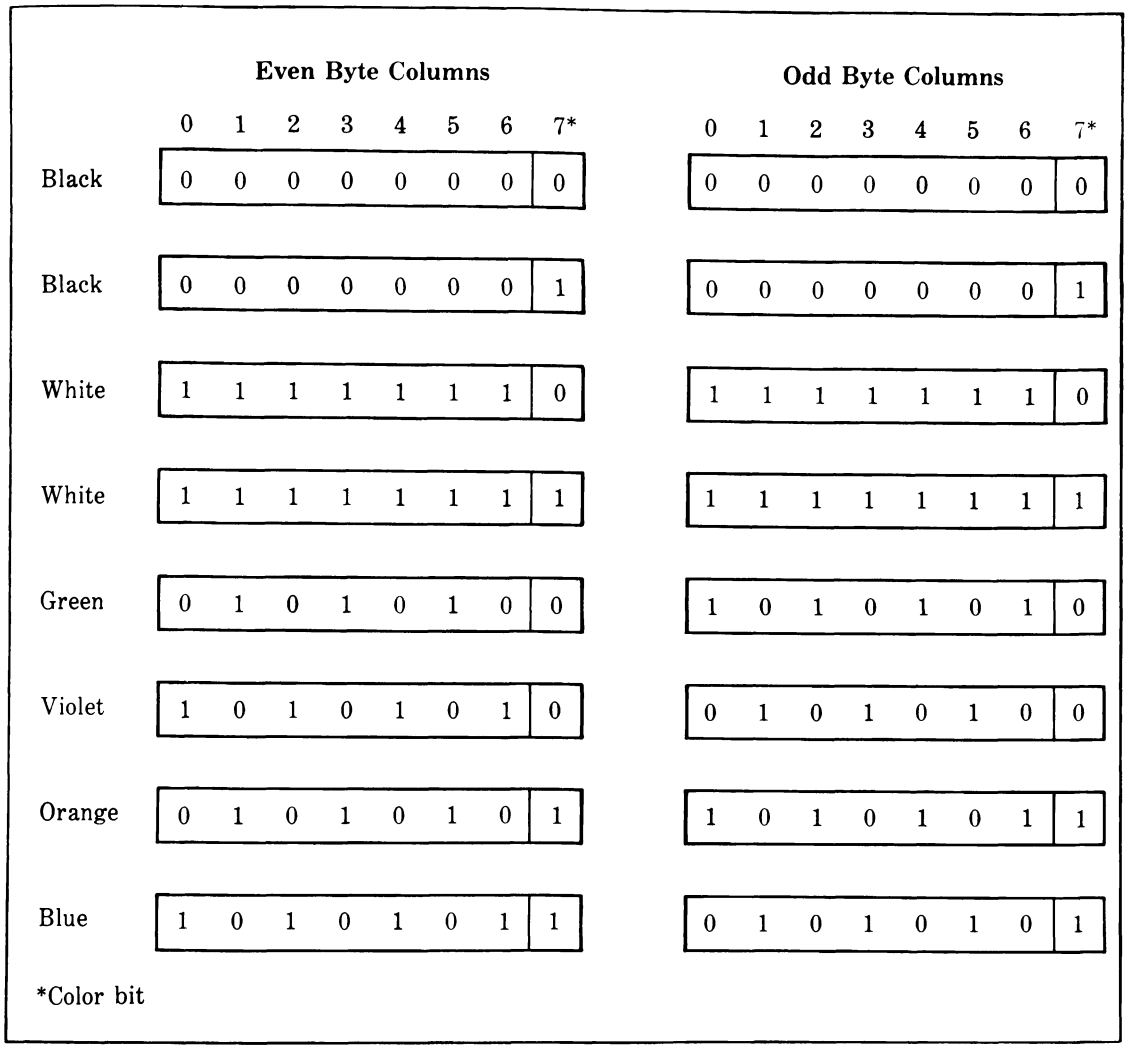

Figure 14-3. How the Apple IIc and IIe generate colors on the screen

sound like a strange way to lay out a screen map, but it does make sense. Since only seven bits in each byte are displayed on the screen, each screen column contains an odd number of dots. If the same on-off pattern were used to generate the colors of even-numbered and odd-numbered bytes, then a pair of set bits or a pair of cleared bits would sit next to each other at each byte change, and the result would be either a pair of white dots or a pair of black dots. This problem has been avoided by using alternating dot patterns to produce the same color in even-numbered and odd-numbered columns on the screen. Thus a given on-off dot pattern can be used to generate a single color all the way across the screen.

How Screen Data Is Stored in RAM

Before we move on to a line-by-line analysis of the HEADLINES program, there is one more point that should be noted. As strange as this may sound, dots are reproduced on the Apple IIc/IIe screen in *the reverse order from the way they are stored in RAM.* When a byte is stored in RAM, bit 0 (the low-order bit) is on the right and bit 7 (the high-order bit) is on the left. On an Apple IIc/IIe screen display, however, the screen dot represented by bit 0 is on the left and the screen dot represented by bit 6 is on the right. (Bit 7, the color bit, is not displayed on the screen.)

Figure 14-4 shows how a dot pattern for the letter B might look on a screen, what that dot pattern would look like when converted into on-off screen data, and how the eight bytes used for the character would actually be stored in memory. As you can see, the bytes stored in RAM mirror their corresponding bit patterns on the Apple IIc/IIe screen.

How the HEADLINES Program Works

The HEADLINES program is quite similar to the PRNTCHRS program. In fact, it's the same program, with a few added improvements.

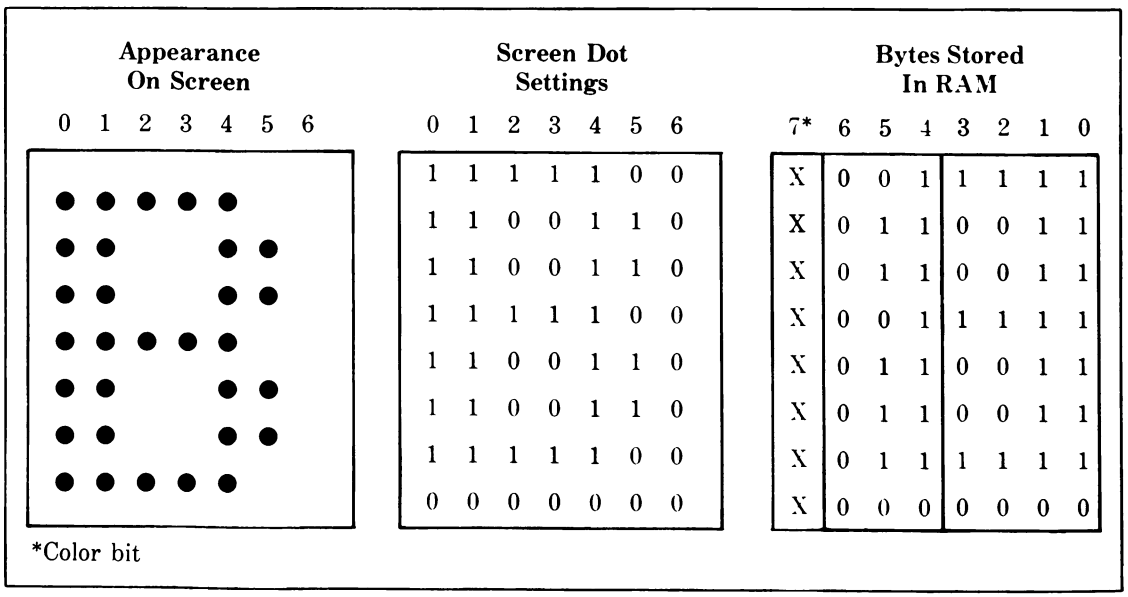

Figure 14-4. How a screen character is stored in RAM

The most obvious difference between PRNTCHRS and HEADLINES is that the characters displayed by HEADLINES are four times as large as those generated by PRNTCHRS, although they're generated by exactly the same character set that was used for the PRNTCHRS program.

Another difference between the two programs is that the characters in the HEADLINES program are displayed against a blue background rather than the black background used in the PRNTCHRS program. Also, the characters are pure white with clean outlines, not vaguely defined and rainbow-edged like the characters in the PRNTCHRS program.

Still another difference between the programs is that HEADLINES has a built-in backspacing feature for error-correcting. Although the HEADLINES program doesn't use a cursor, it usually isn't too difficult to figure out where the next character will appear on the screen, since the program creates its blue background as it moves along.

The system that is used to blow up characters in the HEADLINES program is not really very difficult to understand. In lines 356 through 421 of Program 14-4, each byte to be displayed on the screen is expanded into two bytes. The memory address in which the original byte is stored is labeled OB, and the two new bytes that are used to store the expanded byte are labeled NB1 and NB2.

In lines 356 through 421, OB is expanded into NB1 and NB2 with a series of ASL and ROL instructions. As these ASL and ROL operations take place, a set bit is inserted after each bit in the original byte. That insertion keeps the white characters in OB white but changes their background to a color. The program also sets the color byte of each bit, sets the even-numbered bits in even-numbered bytes, and sets the odd-numbered bits in odd-numbered bytes. These settings make the background of the displayed characters blue.

Once an original byte has been expanded into a pair of bytes labeled NB1 and NB2, the two new bytes are displayed next to each other on the screen, making the character twice as wide as it is in the character set used by the program. After NB1 and NB2 are displayed, their Y position is incremented to the address of the next Y line on the screen and they are displayed again, one pair of bytes right under the other. Thus each character on the screen is twice as high as it is in the character set being used.

When a complete character has been displayed in this fashion, a routine called RESET (lines 283 to 297) is used to set the screen variables

XPSN and YPSN to the proper settings for the display of another character. If the DELETE key or the left ARROW key is pressed, however, a routine called BACKUP (lines 248 to 269) is used to back up an invisible cursor, to clear the last character typed by displaying a space, and to back up again so that a new character can be typed. This backspacing feature makes it easy to correct typing errors when you use the HEADLINES program.

Although HEADLINES was designed as a demonstration program, it could be expanded quite easily into a useful utility. With the addition of a simple screen-dump routine, screens created by the program could be stored on a disk and then used as title screens and for other kinds of eye-catching screen displays.

Double High-Resolution Graphics

With the advent of the Apple IIc and the Apple IIe, the world of double high-resolution graphics has been opened to Apple II programmers. If you have an Apple IIc or a properly equipped Apple IIe, you can use double high-resolution graphics to display up to 16 colors on a high-resolution screen or to increase the horizontal resolution of a monochrome screen to 560 dots.

Furthermore, the color of each dot in a double high-resolution display can be controlled individually. Thus there are no restrictions on what colors can be placed next to each other within a byte—and the only restriction on the number of colors that can be displayed within a byte is the actual number of dots available.

The Apple IIc and IIe use a technique to generate double high-resolution graphics that resembles the one employed to produce an 80-column text display. When double high-resolution graphics are being used, the Apple IIc/IIe fetches display data from the same screen map that is used in standard high-resolution graphics: High-Resolution Display Page 1, which extends from $2000 to $3FFF. In the double high-resolution graphics mode, though, the data retrieved from Display Page 1 comes from both main and auxiliary memory. In a double high-resolution display, bytes from main and auxiliary memory are interleaved in the same way that bytes are interleaved in an 80-column text display. A double high-resolution screen, like an 80-column text screen, measures 80 bytes wide. Even-numbered byte columns, starting with Column $00, come from auxiliary memory. Odd-numbered byte columns starting with Column $01 come from main memory.

How Double High-Resolution Colors Are Programmed

The double high-resolution graphics mode generates colors differently from standard high-resolution graphics. In double high-resolution graphics, just as in standard high-resolution graphics, only seven bits of each byte are displayed on the screen. However, in a double high-resolution display, the high-order bit of each byte on the screen is not a color bit. In fact, in double high-resolution graphics the high-order bit of each byte is not significant. It is neither shown on the screen nor used as a color bit. It is simply not used at all.

Instead of using a color bit to set the color of each byte on the screen, the double high-resolution graphics mode sets the color of each dot on the screen in a very direct manner. A double high-resolution color display has an effective horizontal resolution of 140 dots—exactly the same resolution as a standard high-resolution display. However, a double high-resolution display has much more memory at its disposal than a standard high-resolution display. Thus, in a double high-resolution display, four bits of memory (instead of a single color bit) are used to determine the color of each dot on the screen. Table 14-1 lists the colors used in double high-resolution graphics, along with the code number of each color expressed in both hexadecimal and binary notation.

In double high-resolution graphics, as in standard high-resolution graphics, the bits used for dot patterns are displayed in reverse order

Table 14-1. Colors Used in Double High-Resolution Graphics

Color	Hex Code	Binary Code
Black	$0	0000
Dark red	$1	0001
Dark blue	$2	0010
Violet	$3	0011
Dark green	$4	0100
Gray 1	$5	0101
Dark blue	$6	0110
Light blue	$7	0111
Brown	$8	1000
Orange	$9	1001
Gray 2	$A	1010
Pink	$B	1011
Green	$C	1100
Yellow	$D	1101
Light green	$E	1110
White	$F	1111

from the way they are stored in RAM. Therefore, the dot patterns used to represent colors in double high-resolution graphics are the mirror images of their actual bit patterns. Table 14-2 lists the binary code for each color used in double high-resolution graphics, along with the dot pattern that is used to display each color on the screen.

Figure 14-5 illustrates how this color data is displayed and encoded in the 80 bytes of screen memory that are used to store each line of dots on the screen in RAM. Since only seven bits in each byte of screen memory are used, the bits that are used to generate color codes do not line up with the bytes in which they are stored. That fact makes double high-resolution graphics programs somewhat difficult to write. Before a program can display a given dot in a desired color on a screen, it must carry out a series of rather complex calculations. First, it must calculate the column number of the byte in which the dot will appear. Then it must determine whether the data used to display the desired column will come from main memory or auxiliary memory. The program must also calculate the position of the desired bit within the desired byte. Then comes the most difficult problem. Since the dot patterns used to generate colors don't line up very well with the boundaries of the bytes displayed on the screen, the arrangements of dots needed to draw a color differ from byte to byte. Specifically, there are four different dots that can be used to express each of the 16 colors used in double high-resolution graphics, and the dot pattern that must be used depends upon the column number of the byte in which the dot will appear.

Table 14-2. Dot Patterns of Colors Used in Double High-Resolution Graphics

Color	Binary Code	Dot Pattern
Black	0000	0000
Dark red	0001	1000
Dark blue	0010	0100
Violet	0011	1100
Dark green	0100	0010
Gray 1	0101	1010
Dark blue	0110	0110
Light blue	0111	1110
Brown	1000	0001
Orange	1001	1001
Gray 2	1010	0101
Pink	1011	1101
Green	1100	0011
Yellow	1101	1011
Light green	1110	0111
White	1111	1111

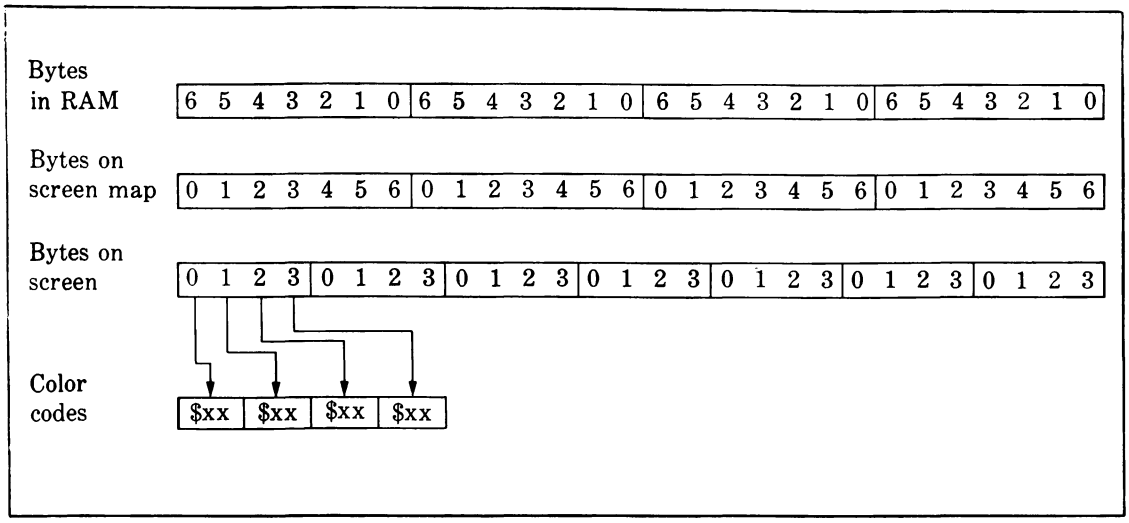

Figure 14-5. How RAM bytes and color codes are displayed in double high-resolution graphics

One way to simplify the writing of a double high-resolution graphics program is to set up a table containing all of the color-code numbers used in double high-res graphics. That table can then be inserted into a machine-language program and consulted for the appropriate color-code number each time a color is used. Table 14-3 lists all 16 of the colors available in double high-resolution graphics, together with the four code numbers that are used for each color.

Table 14-3. Double High-Resolution Colors and Code Numbers

Color	Aux. Memory, Even Columns	Main Memory, Even Columns	Aux. Memory, Odd Columns	Main Memory, Odd Columns
Black	$00	$00	$00	$00
Magenta	$08	$11	$22	$44
Brown	$44	$08	$11	$22
Orange	$4C	$19	$33	$66
Dark green	$22	$44	$08	$11
Gray 1	$2A	$55	$2A	$55
Green	$66	$4C	$19	$33
Yellow	$6E	$5D	$3B	$77
Dark blue	$11	$22	$44	$08
Purple	$19	$33	$66	$4C
Gray 2	$55	$2A	$55	$2A
Pink	$5D	$3B	$77	$6E
Medium blue	$33	$66	$4C	$19
Light blue	$3B	$77	$6E	$5D
Aqua	$77	$6E	$5D	$3B
White	$7F	$7F	$7F	$7F

Writing a Program in Double Hi-Res Graphics Program 14-5, titled DUBHIRES, is a type-and-run program that demonstrates how colors are programmed in double high-resolution graphics. It was written using a Merlin Pro assembler—but, like the other programs in this chapter, it can also be typed and run on an Apple ProDOS assembler with relatively minor modifications.

Program 14-5
A Double High-Resolution Graphics Program

```
 1 *
 2 * DUBHIRES.S
 3 *
 4 * THIS PROGRAM DISPLAYS ALL 16 OF THE COLORS THAT ARE
 5 * AVAILABLE IN APPLE IIC/IIE DOUBLE HIGH-RESOLUTION
     GRAPHICS.
 6 *
 7  ORG $8000
 8 *
 9  JMP INIT
10 *
11 KILOBYTE EQU 1024
12 SCRTOP EQU $2000
13 *
14 LOOKHI EQU $9000
15 LOOKLO EQU $90C0
16 *
17 * SOFT SWITCHES
18 *
19 COL80 EQU $C00D
20 TEXTOFF EQU $C050
21 MIXEDOFF EQU $C052
22 HIRES EQU $C057
23 STORE80 EQU $C001
24 PAGE2ON EQU $C055
25 PAGE2OFF EQU $C054
26 DHIRES EQU $C05E
27 *
28 * USER-DEFINED CONSTANTS
29 *
30 SCRPTR EQU $06
31 TEMPLO EQU SCRPTR+2
32 TEMPHI EQU TEMPLO+1
33 TABPTR EQU $1A
34 COLUMN EQU TABPTR+1
35 ROW EQU COLUMN+2
36 *
37 CLRFLG EQU $300
38 CLRPTR EQU CLRFLG+1
39 FILVAL EQU CLRPTR+1
40 TABSIZ EQU FILVAL+1
41 QUOT1 EQU TABSIZ+1
42 QUOT2 EQU QUOT1+1
43 REMDR1 EQU QUOT2+1
44 REMDR2 EQU REMDR1+1
45 YLINE EQU REMDR2+1
```

```
 46 PRODL EQU YLINE+2
 47 PRODH EQU PRODL+1
 48 MPRL EQU PRODH+1
 49 MPRH EQU MPRL+1
 50 MPDL EQU MPRH+1
 51 MPDH EQU MPDL+1
 52 TOTAL EQU MPDH+1
 53 XPSN EQU TOTAL+2
 54 YPSN EQU XPSN+1
 55 CODENR EQU YPSN+1
 56 BARCNT EQU CODENR+1
 57 *
 58 COLORS EQU *
 59 *
 60 WHITE HEX 7F,7F,7F,7F
 61 BLACK HEX 00,00,00,00
 62 MAG HEX 08,11,22,44
 63 BROWN HEX 44,08,11,22
 64 ORANGE HEX 4C,19,33,66
 65 DGREEN HEX 22,44,08,11
 66 GRAY1 HEX 2A,55,2A,55
 67 GREEN HEX 66,4C,19,33
 68 YELLOW HEX 6E,5D,3B,77
 69 DBLUE HEX 11,22,44,08
 70 PURPLE HEX 19,33,66,4C
 71 GRAY2 HEX 55,2A,55,2A
 72 PINK HEX 5D,3B,77,6E
 73 MBLUE HEX 33,66,4C,19
 74 LBLUE HEX 3B,77,6E,5D
 75 AQUA HEX 77,6E,5D,3B
 76 *
 77 * SET UP DOUBLE HI-RES SCREEN
 78 *
 79 INIT LDA #0
 80  STA STORE80
 81  STA TEXTOFF
 82  STA MIXEDOFF
 83  STA PAGE2OFF
 84  STA HIRES
 85  STA COL80
 86  STA DHIRES
 87 *
 88 * INITIALIZE POINTERS
 89 *
 90  LDA #0
 91  STA COLUMN
 92  STA ROW
 93  STA BARCNT
 94 *
 95  LDA #<SCRTOP
 96  STA SCRPTR
 97  LDA #>SCRTOP
 98  STA SCRPTR+1
 99 *
100 * SET UP Y ADDRESS TABLE
101 *
102  JSR MAKETAB
103 *
```

```
104 * MAIN PROGRAM STARTS HERE
105 *
106  STA PAGE2ON
107  JSR PRNTROW
108  STA PAGE2OFF
109  JSR PRNTROW
110  RTS
111 *
112 PRNTROW LDY ROW
113  LDA LOOKLO,Y
114  STA TEMPLO
115  LDA LOOKHI,Y
116  STA TEMPHI
117 CLEARX LDX #0
118  LDY COLUMN
119 NXTCOL STA PAGE2ON
120  JSR PRINT
121  STA PAGE2OFF
122  JSR PRINT
123  INY
124  CPY #40 ;80 COLUMNS DONE YET?
125  BCS ROWDUN
126  STY COLUMN
127  CPX #4
128  BCS CLEARX
129  JMP NXTCOL
130 ROWDUN LDX ROW
131  INX
132  CPX #192
133  BCS ALLDUN
134  JSR SETBAR
135  STX ROW
136  LDA #0
137  STA COLUMN
138  JMP PRNTROW
139 ALLDUN RTS
140 *
141 * SET UP TABLE OF SCREEN ROW COORDINATES
142 *
143 MAKETAB LDY #0
144 YLOOP CPY #192
145  BCC MOVEON
146  JMP EXIT
147 *
148 * DIVIDE Y BY 8
149 *
150 MOVEON TYA
151  LSR
152  LSR
153  LSR
154  STA QUOT1
155 *
156 * GET REMAINDER OF DIVISION BY 8
157 *
158  TYA
159  AND #7
160  STA REMDR1
```

```
161 *
162 * DIVIDE QUOT1 BY 8
163 *
164  LDA QUOT1
165  LSR
166  LSR
167  LSR
168  STA QUOT2
169 *
170 * GET REMAINDER
171 *
172  LDA QUOT1
173  AND #7
174  STA REMDR2
175 *
176 * CALCULATE LOW BYTE OF Y ADDRESS
177 *
178  LDA #0
179  STA MPRH
180  STA MPDH
181  LDA QUOT2
182  STA MPRL
183  LDA #40
184  STA MPDL
185  JSR MULT16
186  LDA PRODL
187  STA TOTAL
188  LDA PRODH
189  STA TOTAL+1
190 *
191  LDA #0
192  STA MPRH
193  STA MPDH
194  LDA REMDR2
195  STA MPRL
196  LDA #128
197  STA MPDL
198  JSR MULT16
199  CLC
200  LDA PRODL
201  ADC TOTAL
202  STA TOTAL
203  LDA PRODH
204  ADC TOTAL+1
205  STA TOTAL+1
206 *
207  LDA #0
208  STA MPRH
209  LDA REMDR1
210  STA MPRL
211  LDA #<KILOBYTE
212  STA MPDL
213  LDA #>KILOBYTE
214  STA MPDH
215  JSR MULT16
216  CLC
217  LDA PRODL
```

```
218   ADC TOTAL
219   STA TOTAL
220   LDA PRODH
221   ADC TOTAL+1
222   STA TOTAL+1
223 *
224   CLC
225   LDA #<SCRTOP
226   ADC TOTAL
227   STA LOOKLO,Y
228   LDA #>SCRTOP
229   ADC TOTAL+1
230   STA LOOKHI,Y
231 *
232   INY
233   JMP YLOOP
234 *
235 EXIT RTS
236 *
237 * 16-BIT MULTIPLICATION ROUTINE
238 *
239 MULT16 LDA #0
240   STA PRODL
241   STA PRODH
242   LDX #16
243 SHIFT ASL PRODL
244   ROL PRODH
245   ASL MPRL
246   ROL MPRH
247   BCC NOADD
248   CLC
249   LDA MPDL
250   ADC PRODL
251   STA PRODL
252   LDA MPDH
253   ADC PRODH
254   STA PRODH
255 NOADD DEX
256   BNE SHIFT
257   RTS
258 *
259 * SELF-MODIFYING COLOR-PRINTING ROUTINE
260 *
261 PRINT LDA COLORS,X
262   STA (TEMPLO),Y
263   INX
264   RTS
265 *
266 * ROUTINE TO SET UP NEW COLOR BAR
267 *
268 SETBAR TXA
269   PHA
270   LDX BARCNT
271   INX
272   CPX #12
273   BCC LEAP
274   LDA PRINT+1
275   CLC
```

```
276   ADC #4
277   STA PRINT+1
278   LDA PRINT+2
279   ADC #0
280   STA PRINT+2
281   LDX #0
282 LEAP STX BARCNT
283   PLA
284   TAX
285   RTS
```

The DUBHIRES program starts at line 9 with a jump instruction that hops over a block of introductory data and goes to line 79, labeled INIT. This block of instructions, the first segment of executable code in the DUBHIRES program, turns on a group of soft switches that initialize the double high-resolution graphics mode. All of these soft switches reside on Page $C0 of the Apple IIc/IIe memory. The switch labeled DHIRES, situated at memory address $C05E, controls an annunciator that permits the use of double high-resolution graphics rather than text in what would ordinarily be the Apple IIc/IIe's 80-column text mode. (The uses of the other switches used in lines 79 through 86 were explained in Chapter 12.) The TEXTOFF switch, as its name implies, turns off your computer's text mode and completes the job of enabling the use of graphics. The MIXEDOFF switch turns off the text window that is sometimes displayed at the bottom of an Apple IIc/IIe text screen. HIRES sets up a high-resolution graphics screen, COL80 turns on your computer's 80-column firmware, and STORE80 determines whether the PAGE2OFF switch will be used to switch between Text and High-Resolution Display Pages 1 and 2 or between the text and high-resolution display pages in main and auxiliary memory. Since the double high-resolution graphics mode interleaves the display pages in main and auxiliary memory, the STORE80 setting used in the DUBHIRES program is the one that switches between main and auxiliary memory.

In lines 88 through 99 of the DUBHIRES program, some important pointers are initialized. Then, in line 102, a subroutine that sets up a Y-address lookup table is called. This subroutine, called MAKETAB, works just like table-making routines that were used in the PRNTCHRS and HEADLINES programs earlier in this chapter. In the DUBHIRES program, however, the subroutine is used to look up addresses in two segments of memory—the high-resolution display page in main memory and the high-resolution display page in auxiliary memory.

The main part of the DUBHIRES program is only five lines long; it extends from line 106 to line 110. Yet it makes use of the long table-

making routine beginning at line 143, plus three other subroutines: one that prints single rows of colors, one that chooses the code number for each color used, and one that divides the colors being displayed on the screen into 16 horizontal color bars.

The subroutine that prints single rows of colors is labeled PRNTROW and extends from line 112 to line 139. This block of code uses a simple loop to draw a colored line from the left of the screen to the right, calling the MAKETAB routine to determine the memory address in which each byte in the line should be stored in order to make it appear in the correct position on the screen.

The colors used in the DUBHIRES program are taken from a color table that appears in lines 58 through 69. The color codes provided by this table are looked up and printed on the screen by a subroutine called PRINT, which appears in line 261. The PRINT subroutine is called from line 120 of the DUBHIRES program.

PRINT is noteworthy because it uses a recursive programming technique called *address modification*. Assembly-language routines that use address-modification techniques are sometimes called *self-modifying* routines.

When the DUBHIRES program is first loaded into memory, the PRINT subroutine uses indexed addressing to print the color white on a screen. As you can see by looking at the PRINT routine, this color comes from a line of data labeled COLOR—specifically, the string of data in line 60 of the DUBHIRES program.

After the PRINTROW and PRINT routines have been used to print a white bar on the screen, a routine called SETBAR is called. The SETBAR routine uses a loop to change the color being displayed so that a bar of another color can be shown. This is where the technique of address modification comes in.

To understand how address modification works, it helps to consider how assembly language and machine language are related. As we have seen, line 261 of the DUBHIRES program is labeled PRINT. Thus, when the program is assembled into machine language, the first byte of machine code in the line labeled PRINT will be a machine-language instruction that equates to the assembly-language mnemonic LDA.

Now look at the statement LDA COLORS,X in Line 261. This statement was written using an addressing mode known as *absolute indexed addressing*. According to the rules of AppleIIc/IIe assembly-language addressing, the absolute indexed addressing mode always uses a two-byte operand. Thus the operand COLORS,X is a two-byte operand.

When the DUBHIRES program is assembled into machine lan-

guage, the assembly-language instruction LDA (which means "load the accumulator") will be converted into the machine-language op code $BD. When the op code $BD is encountered in a machine-language program, the value of the X register is added to the address specified by the two-byte operand that follows the op code, and the result of this calculation becomes the effective address of the instruction LDA. Once the effective address has been determined, the value stored in that address is loaded into the accumulator.

Since the operand COLORS,X follows an instruction which has been labeled PRINT, the first byte of the two-byte operand COLORS,X could also be referred to by the designation PRINT+1. Similarly, the second byte of the operand COLORS,X could be referred to as PRINT+2. And that is exactly how these two bytes are referred to in lines 274 through 280, the section of the DUBHIRES program which makes use of the technique of address modification.

Once you understand how address modification works, it isn't hard to understand how the technique is used in the DUBHIRES program. In Line 274, the literal number 4 is simply added to the contents of the two bytes referred to as PRINT+1 and PRINT+2 (in other words, to the operand of the LDA instruction in line 261). Then, the next time the PRINT subroutine is executed, the accumulator will be loaded not with the original value of the COLORS,X, but with that value *plus 4*. The next time the SETBAR routine is called, the value of COLORS,X will again be incremented by four, and so on.

Self-modifying code, when properly used, can save both time and memory in an assembly-language program. It can also serve as a handy alternative to indirect indexed addressing when the Y register is being used for other purposes, as it is in the PRINT routine of the DUBHIRES program.

A

Assembly-Language To Machine-Language Conversion Chart

Mnemonic	Address	Format	Function
ADC	61	Ind,X	Add with carry
ADC	65	Zpg	
ADC	69	Imm	
ADC	6D	Abs	
ADC	71	Ind,Y	
ADC	72*	(Zpg)*	
ADC	75	Zpg,X	
ADC	79	Abs,Y	
ADC	7D	Abs,X	

*(in first column) New mnemonic
(in second column) New machine-language op code
(in third column) New address mode

Mnemonic	Address	Format	Function
AND	21	Ind,X	Logical AND
AND	25	Zpg	
AND	29	Imm	
AND	2D	Abs	
AND	31	Ind,Y	
AND	32*	(Zpg)*	
AND	35	Zpg,X	
AND	39	Abs,Y	
AND	3D	Abs,X	
ASL	06	Zpg	Arithmetic shift left
ASL	0A	Acc	
ASL	0E	Abs	
ASL	16	Zpg,X	
ASL	1E	Abs,X	
BCC	90	Rel	Branch if carry clear
BCS	B0	Rel	Branch if carry set
BEQ	F0	Rel	Branch if equal to zero
BIT	24	Zpg	Compare memory bits with accumulator
BIT	2C	Abs	
BIT	34*	Zpg,X	
BIT	3C*	Abs,X	
BIT	89*	Imm	
BMI	30	Rel	Branch on minus
BNE	D0	Rel	Branch if not equal to zero
BPL	10	Rel	Branch on plus
BRA*	80*	Rel	Branch always
BRK	00	Imp	Force break
BVC	50	Rel	Branch if overflow clear
BVS	70	Rel	Branch if overflow set
CLC	18	Imp	Clear carry flag

*(in first column) New mnemonic
(in second column) New machine-language op code
(in third column) New address mode

Mnemonic	Address	Format	Function
CLD	D8	Imp	Clear decimal flag
CLI	58	Imp	Clear interrupt flag
CLV	B8	Imp	Clear overflow flag
CMP	C1	Ind,X	Compare memory with accumulator
CMP	C5	Zpg	
CMP	C9	Imm	
CMP	CD	Abs	
CMP	D1	Ind,Y	
CMP	D2*	(Zpg)*	
CMP	D5	Zpg,X	
CMP	D9	Abs,Y	
CMP	DD	Abs,X	
CPX	E0	Imm	Compare memory with X register
CPX	E4	Zpg	
CPX	EC	Abs	
CPY	C0	Imm	Compare memory with Y register
CPY	C4	Zpg	
CPY	CC	Abs	
DEA or DEC A*	3A*	Acc	Decrement accumulator
DEC	C6	Zpg	Decrement memory
DEC	CE	Abs	
DEC	D6	Zpg,X	
DEC	DE	Abs,X	
DEX	CA	Imp	Decrement X register
DEY	88	Imp	Decrement Y register
EOR	41	Ind,X	Exclusive EOR
EOR	45	Zpg	
EOR	49	Imm	
EOR	4D	Abs	

*(in first column) New mnemonic
(in second column) New machine-language op code
(in third column) New address mode

Mnemonic	Address	Format	Function
EOR	51	Ind,Y	
EOR	52*	(Zpg)*	
EOR	55	Zpg,X	
EOR	59	Abs,Y	
EOR	5D	Abs,X	
INA or INC A*	1A*	Acc	Increment accumulator
INC	E6	Zpg	Increment memory
INC	EE	Abs	
INC	F6	Zpg,X	
INC	FE	Abs,X	
INX	E8	Imp	Increment X register
INY	C8	Imp	Increment Y register
JMP	4C	Abs	Jump to address
JMP	6C	(Abs)	
JMP	7C*	Abs(Ind,X)*	
JSR	20	Abs	Jump to subroutine
LDA	A1	Ind,X	Load accumulator
LDA	A5	Zpg	
LDA	A9	Imm	
LDA	AD	Abs	
LDA	B1	Ind,Y	
LDA	B2*	(Zpg)*	
LDA	B5	Zpg,X	
LDA	B9	Abs,Y	
LDA	BD	Abs,X	
LDX	A2	Imm	Load X register
LDX	A6	Zpg	
LDX	AE	Abs	
LDX	B6	Zpg,Y	
LDX	BE	Abs,Y	
LDY	A0	Imm	Load Y register
LDY	A4	Zpg	

* (in first column) New mnemonic
(in second column) New machine-language op code
(in third column) New address mode

Mnemonic	Address	Format	Function
LDY	AC	Abs	
LDY	B4	Zpg,X	
LSR	46	Zpg	Logical shift right
LSR	4A	Acc	
LSR	4E	Abs	
LSR	56	Zpg,X	
LSR	5E	Abs,X	
NOP	EA	Imp	No operation
ORA	01	Ind,X	Logical OR
ORA	05	Zpg	
ORA	09	Imm	
ORA	0D	Abs	
ORA	11	Ind,Y	
ORA	12*	(Zpg)*	
ORA	15	Zpg,X	
ORA	19	Abs,Y	
ORA	1D	Abs,X	
PHA	48	Imp	Push accumulator
PHP	08	Imp	Push processor (P) register
PHX*	DA*	Imp	Push X register
PHY*	5A*	Imp	Push Y register
PLA	68	Imp	Pull accumulator
PLP	28	Imp	Pull processor status (P) register
PLX*	FA*	Imp	Pull X register
PLY*	7A*	Imp	Pull Y register
ROL	26	Zpg	Rotate left
ROL	2A	Acc	
ROL	2E	Abs	
ROL	3E	Abs,X	
ROL	36	ZX	

*(in first column) New mnemonic
(in second column) New machine-language op code
(in third column) New address mode

Mnemonic	Address	Format	Function
ROR	66	Zpg	Rotate right
ROR	6A	Acc	
ROR	6E	Abs	
ROR	76	Zpg,X	
ROR	7E	Abs,X	
RTI	40	Imp	Return from interrupt
RTS	60	Imp	Return from subroutine
SBC	E1	Ind,X	Subtract with carry
SBC	E5	Zpg	
SBC	E9	Imm	
SBC	ED	Abs	
SBC	F1	Ind,Y	
SBC	F2*	(Zpg)*	
SBC	F5	Zpg,X	
SBC	F9	Abs,X	
SBC	FD	Abs,Y	
SEC	38	Imp	Set carry flag
SED	F8	Imp	Set decimal flag
SEI	78	Imp	Set interrupt flag
STA	81	Ind,X	Store accumulator
STA	85	Zpg	
STA	8D	Abs	
STA	91	Ind,Y	
STA	92*	(Zpg)*	
STA	95	Zpg,X	
STA	99	Abs,Y	
STA	9D	Abs,X	
STX	86	Zpg	Store X register
STX	8E	Abs	
STX	96	Zpg,Y	
STY	84	Zpg	
STY	8C	Abs	
STY	94	Zpg,X	

* (in first column) New mnemonic
(in second column) New machine-language op code
(in third column) New address mode

Mnemonic	Address	Format	Function
STZ*	64*	Zpg	Store zero
STZ*	74*	Zpg,X	
STZ*	9C*	Abs	
STZ*	9E*	Abs,X	
TAX	AA	Imp	Transfer A to X
TAY	A8	Imp	Transfer A to Y
TRB*	14*	Zpg	Test and reset bits
TRB*	1C*	Abs	
TSB*	04*	Zpg	Test and set bits
TSB*	0C*	Abs	
TSX	BA	Imp	Transfer stack pointer to X
TXA	8A	Imp	Transfer X to A
TXS	9A	Imp	Transfer X to stack pointer
TYA	98	Imp	Transfer Y to A

* (in first column) New mnemonic
(in second column) New machine-language op code
(in third column) New address mode

B

Machine-Language To Assembly-Language Conversion Chart

Object Code	Mnemonic	Address
00	BRK	Imp
01	ORA	Ind,X
04*	TSB*	Zpg
05	ORA	Zpg
06	ASL	Zpg
08	PHP	Imp
09	ORA	Imm
0A	ASL	Acc

* (in first column) New mnemonic
(in second column) New machine-language op code
(in third column) New address mode

Object Code	Mnemonic	Address
0C*	TSB*	Abs
0D	ORA	Abs
0E	ASL	Abs
10	BPL	Rel
11	ORA	Ind,Y
12*	ORA	(Zpg)*
14*	TRB*	Zpg
15	ORA	Zpg,X
16	ASL	Zpg,X
18	CLC	Imp
19	ORA	Abs,Y
1A*	INA or INC A*	Acc
1C*	TRB*	Abs
1D	ORA	Abs,X
1E	ASL	Abs,X
20	JSR	Abs
21	AND	Ind,X
24	BIT	Zpg
25	AND	Zpg
26	ROL	Zpg
28	PLP	Imp
29	AND	Imm
2A	ROL	Acc
2C	BIT	Abs
2D	AND	Abs
2E	ROL	Abs
30	BMI	Rel
31	AND	Ind,Y
32*	AND	(Zpg)*
34*	BIT	Zpg,X
35	AND	Zpg,X
36	ROL	ZX
38	SEC	Imp
39	AND	Abs,Y

* (in first column) New mnemonic
(in second column) New machine-language op code
(in third column) New address mode

Object Code	Mnemonic	Address
3A*	DEA or DEC A*	Acc
3C*	BIT	Abs,X
3D	AND	Abs,X
3E	ROL	Abs,X
40	RTI	Imp
41	EOR	Ind,X
45	EOR	Zpg
46	LSR	Zpg
48	PHA	Imp
49	EOR	Imm
4A	LSR	Acc
4C	JMP	Abs
4D	EOR	Abs
4E	LSR	Abs
50	BVC	Rel
51	EOR	Ind,Y
52*	EOR	(Zpg)*
55	EOR	Zpg,X
56	LSR	Zpg,X
58	CLI	Imp
59	EOR	Abs,Y
5A*	PHY*	Imp
5D	EOR	Abs,X
5E	LSR	Abs,X
60	RTS	Imp
61	ADC	Ind,X
64*	STZ*	Zpg
65	ADC	Zpg
66	ROR	Zpg
68	PLA	Imp
69	ADC	Imm
6A	ROR	Acc
6C	JMP	(Abs)
6D	ADC	Abs
6E	ROR	Abs

* (in first column) New mnemonic
(in second column) New machine-language op code
(in third column) New address mode

Object Code	Mnemonic	Address
70	BVS	Rel
71	ADC	Ind,Y
72*	ADC	(Zpg)*
74*	STZ*	Zpg,X
75	ADC	Zpg,X
76	ROR	Zpg,X
78	SEI	Imp
79	ADC	Abs,Y
7A*	PLY*	Imp
7C*	JMP	Abs(Ind,X)*
7D	ADC	Abs,X
7E	ROR	Abs,X
80*	BRA*	Rel
81	STA	Ind,X
84	STY	Zpg
85	STA	Zpg
86	STX	Zpg
88	DEY	Imp
89*	BIT	Imm
8A	TXA	Imp
8C	STY	Abs
8D	STA	Abs
8E	STX	Abs
90	BCC	Rel
91	STA	Ind,Y
92*	STA	(Zpg)*
94	STY	Zpg,X
95	STA	Zpg,X
96	STX	Zpg,Y
98	TYA	Imp
99	STA	Abs,Y
9A	TXS	Imp
9C*	STZ*	Abs
9D	STA	Abs,X
9E*	STZ*	Abs,X

* (in first column) New mnemonic
(in second column) New machine-language op code
(in third column) New address mode

Object Code	Mnemonic	Address
A0	LDY	Imm
A1	LDA	Ind,X
A2	LDX	Imm
A4	LDY	Zpg
A5	LDA	Zpg
A6	LDX	Zpg
A8	TAY	Imp
A9	LDA	Imm
AA	TAX	Imp
AC	LDY	Abs
AD	LDA	Abs
AE	LDX	Abs
B0	BCS	Rel
B1	LDA	Ind,Y
B2*	LDA	(Zpg)*
B4	LDY	Zpg,X
B5	LDA	Zpg,X
B6	LDX	Zpg,Y
B8	CLV	Imp
B9	LDA	Abs,Y
BA	TSX	Imp
BC	LDY	Abs,X
BC	LDY	Abs,X
BD	LDA	Abs,X
BE	LDX	Abs,Y
C0	CPY	Imm
C1	CMP	Ind,X
C4	CPY	Zpg
C5	CMP	Zpg
C6	DEC	Zpg
C8	INY	Imp
C9	CMP	Imm
CA	DEX	Imp
CC	CPY	Abs
CD	CMP	Abs

* (in first column) New mnemonic
(in second column) New machine-language op code
(in third column) New address mode

Object Code	Mnemonic	Address
CE	DEC	Abs
D0	BNE	Rel
D1	CMP	Ind,Y
D2*	CMP	(Zpg)*
D5	CMP	Zpg,X
D6	DEC	Zpg,X
D8	CLD	Imp
D9	CMP	Abs,Y
DA*	PHX*	Imp
DD	CMP	Abs,X
DE	DEC	Abs,X
E0	CPX	Imm
E1	SBC	Ind,X
E4	CPX	Zpg
E5	SBC	Zpg
E6	INC	Zpg
E8	INX	Imp
E9	SBC	Imm
EA	NOP	Imp
EC	CPX	Abs
ED	SBC	Abs
EE	INC	Abs
F0	BEQ	Rel
F1	SBC	Ind,Y
F2*	SBC	(Zpg)*
F5	SBC	Zpg,X
F6	INC	Zpg,X
F8	SED	Imp
F9	SBC	Abs,Y
FA*	PLX*	Imp
FD	SBC	Abs,X
FE	INC	Abs,X

* (in first column) New mnemonic
(in second column) New machine-language op code
(in third column) New address mode

C

The 65C02 Instruction Set

	ADC	Add Memory to Accumulator with Carry
	AND	"AND" Memory with Accumulator
	ASL	Shift One Bit Left
	BCC	Branch on Carry Clear
	BCS	Branch on Carry Set
	BEQ	Branch on Result Zero
*	BIT	Test Memory Bits with Accumulator
	BMI	Branch on Result Minus
	BNE	Branch on Result Not Zero
	BPL	Branch on Result Plus
•	BRA	Branch Always
	BRK	Force Break
	BVC	Branch on Overflow Clear
	BVS	Branch on Overflow Set
	CLC	Clear Carry Flag
	CLD	Clear Decimal Mode
	CLI	Clear Interrupt Disable Bit
	CLV	Clear Overflow Flag
*	CMP	Compare Memory and Accumulator
	CPX	Compare Memory and Index X
	CPY	Compare Memory and Index Y
*	DEC	Decrement by One
	DEX	Decrement Index X by One
	DEY	Decrement Index Y by One
*	EOR	"Exclusive-or" Memory with Accumulator
*	INC	Increment by One
	INX	Increment Index X by One
	INY	Increment Index Y by One
*	JMP	Jump to New Location
	JSR	Jump to New Location Saving Return Address
*	LDA	Load Accumulator with Memory
	LDX	Load Index X with Memory
	LDY	Load Index Y with Memory
	LSR	Shift One Bit Right
	NOP	No Operation
*	ORA	"OR" Memory with Accumulator
	PHA	Push Accumulator on Stack
	PHP	Push Processor Status on Stack
•	PHX	Push Index X on Stack
•	PHY	Push Index Y on Stack
	PLA	Pull Accumulator from Stack
	PLP	Pull Processor Status from Stack
•	PLX	Pull Index X from Stack
•	PLY	Pull Index Y from Stack
	ROL	Rotate One Bit Left
	ROR	Rotate One Bit Right
	RTI	Return from Interrupt
	RTS	Return from Subroutine
*	SBC	Subtract Memory from Accumulator with Borrow
	SEC	Set Carry Flag
	SED	Set Decimal Mode
	SEI	Set Interrupt Disable Bit
*	STA	Store Accumulator in Memory
	STX	Store Index X in Memory
	STY	Store Index Y in Memory
•	STZ	Store Zero in Memory
	TAX	Transfer Accumulator to Index X
	TAY	Transfer Accumulator to Index Y
•	TRB	Test and Reset Memory Bits with Accumulator
•	TSB	Test and Set Memory Bits with Accumulator
	TSX	Transfer Stack Pointer to Index X
	TXA	Transfer Index X to Accumulator
	TXS	Transfer Index X to Stack Pointer
	TYA	Transfer Index Y to Accumulator

Note: • = New Instruction
* = Old Instruction with New Addressing Modes

D

65C02 Op Code Table

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK	ORA ind, X			TSB • zpg	ORA zpg	ASL zpg		PHP	ORA imm	ASL A		TSB • abs	ORA abs	ASL abs		0
1	BPL rel	ORA ind, Y	ORA* ind		TRB • zpg	ORA zpg, X	ASL zpg, X		CLC	ORA abs, Y	INC* A		TRB • abs	ORA abs, X	ASL abs X		1
2	JSR abs	AND ind, X			BIT zpg	AND zpg	ROL zpg		PLP	AND imm	ROL A		BIT abs	AND abs	ROL abs		2
3	BMI rel	AND ind, Y	AND* ind		BIT * zpg, X	AND zpg, X	ROL zpg, X		SEC	AND abs, Y	DEC* A		BIT * abs, X	AND abs, X	ROL abs, X		3
4	RTI	EOR ind, X				EOR zpg	LSR zpg		PHA	EOR imm	LSR A		JMP abs	EOR abs	LSR abs		4
5	BVC rel	EOR ind, Y	EOR* ind			EOR zpg, X	LSR zpg, X		CLI	EOR abs, Y	PHY•			EOR abs, X	LSR abs, X		5
6	RTS	ADC ind, X			STZ • zpg	ADC zpg	ROR zpg		PLA	ADC imm	ROR A		JMP ind	ADC abs	ROR abs		6
7	BVS rel	ADC ind, Y	ADC* ind		STZ • zpg, X	ADC zpg, X	ROR zpg, X		SEI	ADC abs, Y	PLY•		JMP * ind, X	ADC abs, X	ROR abs, X		7
8	BRA • rel	STA ind, X			STY zpg	STA zpg	STX zpg		DEY	BIT * imm	TXA		STY abs	STA abs	STX abs		8
9	BCC rel	STA ind, Y	STA* ind		STY zpg, X	STA zpg, X	STX zpg, Y		TYA	STA abs, Y	TXS		STZ • abs	STA abs, X	STZ • abs, X		9
A	LDY imm	LDA ind, X	LDX imm		LDY zpg	LDA zpg	LDX zpg		TAY	LDA imm	TAX		LDY abs	LDA abs	LDX abs		A
B	BCS rel	LDA ind, Y	LDA* ind		LDY zpg, X	LDA zpg, X	LDX zpg, Y		CLV	LDA abs, Y	TSX		LDY abs, X	LDA abs, X	LDX abs, Y		B
C	CPY imm	CMP ind, X			CPY zpg	CMP zpg	DEC zpg		INY	CMP imm	DEX		CPY abs	CMP abs	DEC abs		C
D	BNE rel	CMP ind, Y	CMP* ind			CMP zpg, X	DEC zpg, X		CLD	CMP abs, Y	PHX•			CMP abs, X	DEC abs, X		D
E	CPX imm	SBC ind, X			CPX zpg	SBC zpg	INC zpg		INX.	SBC imm	NOP		CPX abs	SBC abs	INC abs		E
F	BEQ rel	SBC ind, Y	SBC* ind			SBC zpg, X	INC zpg, X		SED	SBC abs, Y	PLX•			SBC abs, X	INC abs, X		F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

NOTE. • = New Instruction
* = Old Instruction with New Addressing Mode

Op Code Matrix Legend

Note:

E

65C02 Addressing Modes

Fifteen addressing modes are available to the user of the WDC W65SCXXX family of microprocessors. The addressing modes are described in the following paragraphs

Immediate Addressing
With immediate addressing, the operand is contained in the second byte of the instruction, no further memory addressing is required.

Absolute Addressing
For absolute addressing, the second byte of the instruction specifies the eight low order bits of the effective address while the third byte specifies the eight high order bits. Therefore, this addressing mode allows access to the total 65K bytes of addressable memory.

Zero Page Addressing
Zero page addressing allows shorter code and execution times by only fetching the second byte of the instruction and assuming a zero high address byte. The careful use of zero page addressing can result in significant increase in code efficiency

Implied Addressing
In the implied addressing mode, the address containing the operand is implicitly stated in the operation code of the instruction

Accumulator Addressing
This form of addressing is represented with a one byte instruction and implies an operation on the accumulator. The op codes are included under implied addressing

Zero Page Indexed Indirect Addressing: (IND, X)
With zero page indexed indirect addressing (usually referred to as Indirect X) the second byte of the instruction is added to the contents of the X index register, the carry is discarded. The result of this addition points to a memory location on page zero whose contents is the low order eight bits of the effective address. The next memory location in page zero contains the high order eight bits of the effective address. Both memory locations specifying the high and low order bytes of the effective address must be in page zero

Absolute Indexed Indirect Addressing (Jump Instruction Only)
With absolute indexed indirect addressing, the contents of the second and third instruction bytes are added to the X register. The result of this addition points to a memory location containing the lower-order eight bits of the effective address. The next memory location contains the higher-order eight bits of the effective address (opcode 7C).

Indirect Indexed Addressing :(IND), Y
This form of addressing is usually referred to as Indirect, Y. The second byte of the instruction points to a memory location in page zero. The contents of this memory location is added to the contents of the Y index register, the result being the low order eight bits of the effective address. The carry from this addition is added to the contents of the next page zero memory location, the result being the high order eight bits of the effective address.

Zero Page Indexed Addressing
Zero page absolute addressing is used in conjunction with the index register and is referred to as "Zero Page, X" or "Zero Page, Y." The effective address is calculated by adding the second byte to the contents of the index register. Since this is a form of "Zero Page" addressing, the content of the second byte references a location in page zero. Additionally, due to the "Zero Page" addressing nature of this mode, no carry is added to the high order eight bits of memory and crossing of page boundaries does not occur.

Absolute Indexed Addressing
Absolute indexed addressing is used in conjunction with X and Y index register and is referred to as "Absolute, X," and "Absolute, Y." The effective address is formed by adding the contents of X and Y to the address contained in the second and third bytes of the instruction. This mode allows the index register to contain the index or count value and the instruction to contain the base address. This type of indexing allows any location referencing and the index to modify multiple fields resulting in reduced coding and execution time.

Relative Addressing
Relative addressing is used only with branch instruction; it establishes a destination for the conditional branch

Zero Page Indirect Addressing: Indirect
In this form of addressing, the second byte of the instruction contains the low order eight bits of a memory location. The high order eight bits is always zero. The contents of the fully specified memory location is the low order byte of the effective address. The next memory location contains the high order byte of the effective address.

Absolute Indirect Addressing (Jump Instruction Only)
The second byte of the instruction contains the low order eight bits of a memory location. The high order eight bits of that memory location is contained in the third byte of the instruction. The contents of the fully specified memory location is the low order byte of the effective address. The next memory location contains the high order byte of the effective address which is loaded into the 16 bits of the program counter (op code 6C)

F

The 65802/65816 Instruction Set

A. The Original 6502 Instruction Set (151 Op Codes)

1	ADC	Add Memory to Accumulator with Carry
2	AND	"AND" Memory with Accumulator
3	ASL	Shift Left One Bit (Memory or Accumulator)
4	BCC	Branch on Carry Clear
5	BCS	Branch on Carry Set
6	BEQ	Branch on Result Zero
7	BIT	Test Bits in Memory with Accumulator
8	BMI	Branch on Result Minus
9	BNE	Branch on Result Not Zero
10	BPL	Branch on Result Plus
11	BRK	Force Break
12	BVC	Branch on Overflow Clear
13	BVS	Branch on Overflow Set
14	CLC	Clear Carry Flag
15	CLD	Clear Decimal Mode
16	CLI	Clear Interrupt Disable Bit
17	CLV	Clear Overflow Flag
18	CMP	Compare Memory and Accumulator
19	CPX	Compare Memory and Index X
20	CPY	Compare Memory and Index Y
21	DEC	Decrement Memory by One
22	DEX	Decrement Index X by One
23	DEY	Decrement Index Y by One
24	EOR	"Exclusive-or" Memory with Accumulator
25	INC	Increment Memory by One
26	INX	Increment Index X by One
27	INY	Increment Index Y by One
28	JMP	Jump to New Location
29	JSR	Jump to New Location Saving Return Address
30	LDA	Load Accumulator with Memory
31	LDX	Load Index X with Memory
32	LDY	Load Index Y with Memory
33	LSR	Shift One Bit Right (Memory or Accumulator)
34	NOP	No Operation
35	ORA	"OR" Memory with Accumulator
36	PHA	Push Accumulator on Stack
37	PHP	Push Processor Status on Stack
38	PLA	Pull Accumulator from Stack
39	PLP	Pull Processor Status from Stack
40	ROL	Rotate One Bit Left (Memory or Accumulator)
41	ROR	Rotate One Bit Right (Memory of Accumulator)
42	RTI	Return from Interrupt
43	RTS	Return from Subroutine
44	SBC	Subtract Memory from Accumulator with Borrow
45	SEC	Set Carry Flag
46	SED	Set Decimal Mode
47	SEI	Set Interrupt Disable Status
48	STA	Store Accumulator in Memory
49	STX	Store Index X in Memory
50	STY	Store Index Y in Memory
51	TAX	Transfer Accumulator to Index X
52	TAY	Transfer Accumulator to Index Y
53	TSX	Transfer Stack Pointer to Index X
54	TXA	Transfer Index X to Accumulator
55	TXS	Transfer Index X to Stack Register
56	TYA	Transfer Index Y to Accumulator

B. New W65SCXXX Instructions (13 Op Codes)

1	BRA	Branch Relative always
2	PLX	Pull X from Stack
3	PLY	Pull Y from Stack
4	PHX	Push X on Stack
5	PHY	Push Y on Stack
6	STZ	Store Zero in Memory (Direct, Direct, X Abs, Abs, X)
7	TRB	Test and Reset Memory Bits Determined by Accumulator A (Direct and Absolute)
8	TSB	Test and Set Memory Bits Determined by Accumulator A (Direct and Absolute)

C. New W65SCXXX Addressing Modes (14 Op Codes)

1	BIT	Test Bits in Memory with Accumulator (Direct, X, Absolute, X Immediate)
2	DEC	Decrement (Accumulator)
3	Group I	Instructions (Direct Indirect (8 Op Codes))
4	INC	Increment (Accumulator)
5	JMP	Jump to New Location (Absolute Indexed Indirect)

Continued

D. Group I Instructions with New Addressing Modes (48 Op Codes)

- Direct Indirect Long Indexed with Y (8 Op Codes)
- Direct Indirect Long (8 Op Codes)
- Absolute Long and Absolute Long Indexed with X (16 Op Codes)
- Stack Relative (8 Op Codes)
- Stack Relative Indirect Indexed Y (8 Op Codes)

1 ADC Add Memory to Accumulator with Carry
2. AND "AND" Memory with Accumulator
3 CMP Compare Memory and Accumulator
4 EOR "Exclusive-or" Memory with Accumulator
5 LDA Load Accumulator with Memory
6 ORA "Or" Memory with Accumulator
7 SBC Subtract Memory from Accumulator with Borrow
8 STA Store Accumulator in Memory

E. New Push and Pull Instructions (7 Op Codes)

1 PEA Push Effective Absolute Address or Immediate Data Word on Stack
2 PEI Push Effective Indirect Address or Direct Data Word on Stack
3 PER Push Effective Program Counter Relative Indirect Address or Program Counter Relative Data Word on Stack
4 PLB Pull Data Bank Register from Stack
5 PLD Pull Direct Register from Stack
6 PHB Push Data Bank Register on Stack
7 PHD Push Direct Register on Stack
8 PHK Push Program Bank Register on stack

F. Status Register Instructions (2 Op Codes)

1 REP Reset Status Bits Defined by Immediate Byte 1 = Reset
0 = Do not change
2 SEP Set Status Bits Defined by Immediate Byte 1 = Set
0 = Do not change

G. New Register Transfer Instructions (8 Op Codes)

1. TCD Transfer C Accumulator to Direct Register D
2 TDC Transfer Direct Register D to C Accumulator
3 TCS Transfer C Accumulator to Stack Register
4. TSC Transfer Stack Register to Accumulator C
5 TXY Transfer X to Y
6 TYX Transfer Y to X
7 XBA Exchange B and A
8 SCE Exchange Carry Bit C with Emulation Bit E

H. New Branch, Jump and Return Instructions (6 Op Codes)

1 BRL Branch Relative Long Always (16 Bit Relative—32768 to + 32767) (Addressing Mode)
2 JML Jump Indirect Long
3 JMP Jump Absolute Long
4 JSL Jump to Subroutine Long (Uses RTL for Return)
5. JSR Jump to Subroutine (Indexed Indirect)
6 RTL Return from Subroutine Long

I. New Block Move Instructions (2 Op Codes)

1 MVN Move Block from Source (X Addressed) to Destination (Y Addressed), Block Length Defined by C, X Y are Incremented.
2 MVP Move Block from Source (X Addressed) to Destination (Y Addressed), Block Length Defined by C, X, Y are Decremented

J. New Co-Processor Operations (1 Op Code)

1. COP Co-Processor Instruction with Associated COP Vector and ABORT Input Supports Co-Processing Function i e , Floating Point Processors, etc

K. New System Control Instructions (3 Op Codes)

1 STP Stop-the-clock Instruction Stops the Oscillator Input (or 02 Input) During 02 = 1 This Mode Is Released When RES Goes to a Zero System Initialization May Be Desired; However, if After RESET One Performed an RTI, Program Execution Begins With the Instruction Following the STP Op Code in Program Sequence
2 WAI Wait for Interrupt Pulls RDY Low and Is Cleared by IRQ or NMI Active Input.
3. WDM There is One Reserved Op Code Defined as WDM Which Will Be Used For Future Systems The W65SC816 Performs a No-Operation

Note: © Western Design Center, Inc.
Used by permission.

65816 Addressing Modes

Addressing Modes

Twenty-four addressing modes are available to the user of the W65SC816 family of microprocessors. The addressing modes are described in the following paragraphs

1. Immediate Addressing [imm]
With immediate addressing the operand is contained in the second byte (second and third byte for 16 bit data) of the instruction

2, 3. Absolute and Absolute Long Addressing [a], [al]
For absolute addressing the second byte of the instruction specifies the eight low order bits of the effective address while the third byte specifies the eight high order bits For absolute long addressing the fourth byte specifies the bank address The full 16.7 megabyte address space is addressed in the long mode In the short mode the bank address is specified by the data bank register.

4. Direct Addressing [d]
Direct addressing allows for shorter code and execution times by only fetching a second byte of instruction. The second byte is added to the direct register (D) value. When the direct register low (DL) is zero fastest execution occurs The bank address is always zero.

5. Accumulator Addressing [acc]
This form of addressing is represented with a one byte instruction and performs an operation on the accumulator(s)

6. Implied Addressing [imp]
In the implied addressing mode the address of the operand is implicitly stated in the operation code of the instruction.

7, 8. Direct Indirect Indexed and Direct Indirect Indexed Long Addressing [(d), y], [(dl), y]
This form of addressing is usually referred to as Indirect, Y The second byte of the instruction is added to the direct register and points to a memory location in bank zero The contents of this memory location and the byte following (the next byte is the bank address for the long mode) are added to the Y index register with the result being the effective address For the short mode the bank address is specified by the data bank register Note that when DL equals zero execution is fastest

9. Direct Indexed Indirect Addressing [(d,x)]
With direct indexed indirect addressing (usually referred to as Indirect, X) the second byte of the instruction is added to the contents of the direct register and then adding the X register value. The result of these additions points to a memory location on bank zero whose contents is the low order byte of the effective address with the byte following the high byte of the effective address. The bank address of the effective address is specified by the data bank register.

10, 11. Direct Indexed with X and Direct Indexed with Y Addressing [d,x], [d,y]
Direct indexed with X usually referred to as Direct, X and direct indexed with Y usually referred to as Direct, Y are two byte instructions The second byte is added to the direct register (D) and this result is added to the appropriate index register The bank address is always zero Execution is fastest when the low byte of the direct register (DL) is zero.

12, 13, 14. Absolute Indexed with X, Absolute Indexed Long with X, and Absolute Indexed with Y Addressing [a,x], [al,x], [a,y]
Absolute indexed addressing is used in conjunction with the X and Y index registers and is referred to as Absolute, X Absolute Long, X and Absolute, Y. The effective address is formed by adding the contents of the X or Y register to the second and third bytes of the instructions The bank address is specified by the data bank register except in the long mode the fourth byte specifies the bank address

15, 16. Program Counter Relative and Program Counter Relative Long Addressing [r], [rl]
Program counter relative addressing, usually referred to as relative and relative long addressing is used only with the branch instructions The second byte is added to the program counter which for relative creates a +128 or -127 byte offset. The second and third bytes are added to the program counter to create +32768 or -32767 byte offset for the branch always long operation

17. Absolute Indirect Addressing (Jump Instruction Only) [(a)]
The second and third bytes of the instruction contains the low and high order address bytes of a memory location located in bank zero This memory location and the byte following contain the effective address which is loaded into the program counter. The destination bank address is specified by the program bank register except for the JML instruction the third byte fetched is the destination bank address

Continued

18, 19. Direct Indirect and Direct Indirect Long Addressing [(d)], [(dl)]
In this form of addressing the second byte of the instruction is added to the direct register and the result points to a memory location in bank zero. The contents of this location and the following location (the next location is the bank address for the long mode) is the effective address. The bank address is specified by the data bank register for the direct indirect mode.

20. Absolute Indexed Indirect Addressing (Jump and Jump to Subroutine) [(a,x)]
With absolute indexed indirect addressing the second and third bytes of the instruction are added to the X index register contents. The result points to the low and (byte following) high order bytes which are loaded into the program counter. The bank address is specified by the program bank register.

21. Stack Addressing [s]
This addressing mode uses the stack register to address memory locations. The instructions which use the stack addressing include push, pull, interrupts, jump to subroutine, return from interrupt and return from subroutine. The bank address is always zero. Vectors are always pulled from bank 00. (See Compatibility Issues for 6502 Emulation)

22. Stack Relative Addressing [sr]
With stack relative addressing the second byte of the instruction is added to the stack register value. This effective address points to a data memory location on the stack. For 16 bit data the next location on the stack is the high byte of data. This addressing mode, in conjunction with using the push instructions, may be used to pass data to subroutines using the stack. The new TSC and TCS instructions provide fast stack modification. The direct register can be used for user stack functions. The bank register is always zero.

23. Stack Relative Indirect Indexed Addressing [(sr),y]
With stack relative indirect indexed with Y the second byte of the instruction is added to the stack register value. The address formed by this addition points to the low byte (the next location contains the high byte) of an indirect address. The Y register is added to this address to form the effective data address. This addressing mode, in conjunction with using the push effective address (PEA, PEI, PER) instructions, may be used to pass data addresses to subroutines using the stack. The new TSC and TCS instructions provide fast stack register modification. The direct register can be used for user stack functions. The data bank register is the bank address for the effective address.

24. Block Move Addressing [xyc]
This addressing mode is used for multiple byte moves forward (MVP) or backward (MVN). These three byte instructions use the X register for the source address, the Y register for the destination address and the C accumulator contains the number of bytes to be moved. The destination bank address is the second byte of the instruction with the source bank specified by the third byte. The data bank register is loaded with the destination bank value (second byte of the instruction).

Note: © Western Design Center, Inc.
Used by permission.

H

65816 Op Code Table

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	MSD
0	BRK s 2 8	ORA(d,x) 2 6	COP s 2*8	ORA sr 2*4	TSB d 2●5	ORA d 2 3	ASL d 2 5	ORA(dl) 2*6	PHP s 1 3	ORA imm 2 2	ASL acc 1 2	PHD s 1*4	TSB a 3●6	ORA a 3 4	ASL a 3 6	ORA al 4*5	0
1	BPL r 2 2	ORA(d),y 2 5	ORA (d) 2●5	ORA(sr),y 2*7	TRB d 2●5	ORA d,x 2 4	ASL d,x 2 6	ORA(dl),y 2*6	CLC imp 1 2	ORA a,y 3 4	INC acc 1●2	TCS imp 1*2	TRB a 3●6	ORA a,x 3 4	ASL a,x 3 7	ORA al,x 4*5	1
2	JSR a 3 6	AND(d,x) 2 6	JSL al 4*8	AND sr 2*4	BIT d 2 3	AND d 2 3	ROL d 2 5	AND(dl) 2*6	PLP s 1 4	AND imm 2 2	ROL acc 1 2	PLD s 1*5	BIT a 3 4	AND a 3 4	ROL a 3 6	AND al 4*5	2
3	BMI r 2 2	AND(d),y 2 5	AND(d) 2●5	AND(sr),y 2*7	BIT d,x 2●4	AND d,x 2 4	ROL d,x 2 6	AND(dl),y 2*6	SEC imp 1 2	AND a,y 3 4	DEC acc 1●2	TSC imp 1*2	BIT a,x 3●4	AND a,x 3 4	ROL a,x 3 7	AND al,x 4*5	3
4	RTI s 1 7	EOR(d,x) 2 6	WDM RESERVED*	EOR sr 2*4	MVP xyc 3*7	EOR d 2 3	LSR d 2 5	EOR(dl) 2*6	PHA s 1 3	EOR imm 2 2	LSR acc 1 2	PHK s 1*3	JMP a 3 3	EOR a 3 4	LSR a 3 6	EOR al 4*5	4
5	BVC r 2 2	EOR(d),y 2 5	EOR (d) 2●5	EOR(sr),y 2*7	MVN xyc 3*7	EOR d,x 2 4	LSR d,x 2 6	EOR(dl),y 2*6	CLI imp 1 2	EOR a,y 3 4	PHY s 1●3	TCD imp 1*2	JMP al 3*4	EOR a,x 3 4	LSR a,x 3 7	EOR al,x 4*5	5
6	RTS s 1 6	ADC(d,x) 2 6	PER s 3*6	ADC sr 2*4	STZ d 2●3	ADC d 2 3	ROR d 2 5	ADC(dl) 2*6	PLA s 1 4	ADC imm 2 2	ROR acc 1 2	RTL s 1*6	JMP (a) 3 5	ADC a 3 4	ROR a 3 6	ADC al 4*5	6
7	BVS r 2 2	ADC(d),y 2 5	ADC(d) 2●5	ADC(sr),y 2*7	STZ d,x 2●4	ADC d,x 2 4	ROR d,x 2 6	ADC(dl) y 2*6	SEI imp 1 2	ADC a,y 3 4	PLY s 1●4	TDC imp 1*2	JMP(a,x) 3●6	ADC a,x 3 4	ROR a,x 3 7	APC al,x 4*5	7
8	BRA r 2●2	STA(d,x) 2 6	BRL rl 3*3	STA sr 2*4	STY d 2 3	STA d 2 3	STX d 2 3	STA(dl) 2*6	DEY imp 1 2	BIT imm 2●2	TXA imm 1 2	PHB s 1*3	STY a 3 4	STA a 3 4	STX a 3 6	STA al 4*5	8
9	BCC r 2 2	STA(d),y 2 6	STA (d) 2●5	STA(sr),y 2*7	STY d,x 2 4	STA d,x 2 4	STX d,y 2 4	STA(dl),y 2*6	TYA imp 1 2	STA a,y 3 5	TXS imp 1 2	TXY imp 1*2	STZ a 3●4	STA a,x 3 5	STZ a,x 3●5	STA al,x 4*5	9
A	LDY imm 2 2	LDA(d,x) 2 6	LDX imm 2 2	LDA sr 2*4	LDY d 2 3	LDA d 2 3	LDX d 2 3	LDA(dl) 2*6	TAY imp 1 2	LDA imm 2 2	TAX imp 1 2	PLB s 1*4	LDY a 3 4	LDA a 3 4	LDX a 3 4	LDA al 4*5	A
B	BCS r 2 2	LDA(d),y 2 5	LDA(d) 2●5	LDA(sr),y 2*7	LDY d,x 2 4	LDA d,x 2 4	LDX d,y 2 4	LDA(dl)y 2*6	CLV imp 1 2	LDA a,y 3 4	TSX imp 1 2	TYX imp 1*2	LDY a,x 3 4	LDA a,x 3 4	LDX a,y 3 4	LDA al,x 4*5	B
C	CPY imm 2 2	CMP(d,x) 2 6	REP imm 2*3	CMP sr 2*4	CPY d 2 3	CMP d 2 3	DEC d 2 5	CMP(dl) 2*6	INY imp 1 2	CMP imm 2 2	DEX imm 1 2	WAI imp 1*3	CPY a 3 4	CMP a 3 4	DEC a 3 6	CMP al 4*5	C
D	BNE r 2 2	CMP(d),y 2 5	CMP (d) 2●5	CMP(sr),y 2*7	PEI s 2*6	CMP d,x 2 4	DEC d,x 2 6	CMP(dl),y 2*6	CLD imp 1 2	CMP a,y 3 4	PHX s 1●3	STP imp 1*3	JML (a) 3*6	CMP a,x 3 4	DEC a,x 3 7	CMP al x 4*5	D
E	CPX imm 2 2	SBC(d,x) 2 6	SEP imm 2*3	SBC sr 2*4	LPX d 2 3	SBC d 2 3	INC d 2 5	SBC(dl) 2 6	INX imp 1 2	SBC imm 2 2	NOP imp 1 2	XBA imp 1*3	CPX a '3 4	SBC a 3 4	INC a 3 6	SBC al 4*5	E
F	BEQ r 2 2	SBC(d),y 2 5	SBC (d) 2●5	SBC(sr),y 2*7	PEA s 3*5	SBC d,x 2 4	INC d,x 2 6	SBC(dl) y 2*6	SED imp 1 2	SBC a,y 3 4	PLX s 1●4	XCE imp 1*2	JSR(a,x) 3*6	SBC a,x 3 4	INC a,x 3 7	SBC al,x 4*5	F

*New W65SC816 Op Codes

● W65SC02 Op Codes

Op Code Matrix Legend

Note: © Western Design Center, Inc. Used by permission.

The ASCII Character Set For the Apple II

A. $00-3F: Reverse Video Characters

Hex	Dec	Screen
$00	0	@
$01	1	A
$02	2	B
$03	3	C
$04	4	D
$05	5	E
$06	6	F
$07	7	G
$08	8	H
$09	9	I
$0A	10	J
$0B	11	K
$0C	12	L
$0D	13	M
$0E	14	N
$0F	15	O
$10	16	P

Hex	Dec	Screen
$11	17	Q
$12	18	R
$13	19	S
$14	20	T
$15	21	U
$16	22	V
$17	23	W
$18	24	X
$19	25	Y
$1A	26	Z
$1B	27	[
$1C	28	\
$1D	29	]
$1E	30	’
$1F	31	
$20	32	SPACE
$21	33	!
$22	34	”
$23	35	#
$24	36	$
$25	37	%
$26	38	&
$27	39	’
$28	40	(
$29	41	)
$2A	42	*
$2B	43	+
$2C	44	,
$2D	45	-
$2E	46	.
$2F	47	/
$30	48	0
$31	49	1
$32	50	2
$33	51	3
$34	52	4
$35	53	5
$36	54	6
$37	55	7

Hex	Dec	Screen
$38	56	8
$39	57	9
$3A	58	:
$3B	59	;
$3C	60	<
$3D	61	=
$3E	62	>
$3F	63	?

B. $40-7F: Flashing Characters*

Hex	Dec	Screen
$40	64	@
$41	65	A
$42	66	B
$43	67	C
$44	68	D
$45	69	E
$46	70	F
$47	71	G
$48	72	H
$49	73	I
$4A	74	J
$4B	75	K
$4C	76	L
$4D	77	M
$4E	78	N
$4F	79	O
$50	80	P
$51	81	Q
$52	82	R
$53	83	S
$54	84	T
$55	85	U
$56	86	V
$57	87	W
$58	88	X

* ASCII characters $40 through $5F are displayed as special MouseText characters if mouse firmware is installed and active.

Hex	Dec	Screen
$59	89	Y
$5A	90	Z
$5B	91	[
$5C	92	\
$5D	93	]
$5E	94	’
$5F	95	—
$60	96	SPACE
$61	97	!
$62	98	”
$63	99	#
$64	100	$
$65	101	%
$66	102	&
$67	103	’
$68	104	(
$69	105	)
$6A	106	*
$6B	107	+
$6C	108	,
$6D	109	
$6E	110	.
$6F	111	/
$70	112	0
$71	113	1
$72	114	2
$73	115	3
$74	116	4
$75	117	5
$76	118	6
$77	119	7
$78	120	8
$79	121	9
$7A	122	:
$7B	123	;

$7C	124	<
$7D	125	=
$7E	126	>
$7F	127	?

C. $80-9F: Control Characters

Hex	Dec	Key
$80	128	@
$81	129	A
$82	130	B
$83	131	C
$84	132	D
$85	133	E
$86	134	F
$87	135	G
$88	136	H
$89	137	I
$8A	138	J
$8B	139	K
$8C	140	L
$8D	141	M
$8E	142	N
$8F	143	O
$90	144	P
$91	145	Q
$92	146	R
$93	147	S
$94	148	T
$95	149	U
$96	150	V
$97	151	W
$98	152	X
$99	153	Y
$9A	154	Z
$9B	155	[

Hex	Dec	Key
$9C	156	\
$9D	157	]
$9E	158	^
$9F	159	_

D. $A0-FF: Normal Characters

Hex	Dec	Key
$A0	160	SPC
$A1	161	!
$A2	162	"
$A3	163	#
$A4	164	$
$A5	165	%
$A6	166	&
$A7	167	'
$A8	168	(
$A9	169	)
$AA	170	*
$AB	171	+
$AC	172	,
$AD	173	
$AE	174	>
$AF	175	/
$B0	176	0
$B1	177	1
$B2	178	2
$B3	179	3
$B4	180	4
$B5	181	5
$B6	182	6
$B7	183	7
$B8	184	8
$B9	185	9
$BA	186	:
$BB	187	;
$BC	188	<
$BD	189	=
$BE	190	>

Hex	Dec	Key
$BF	191	?
$C0	192	@
$C1	193	A
$C2	194	B
$C3	195	C
$C4	196	D
$C5	197	E
$C6	198	F
$C7	199	G
$C8	200	H
$C9	201	I
$CA	202	J
$CB	203	K
$CC	204	L
$CD	205	M
$CE	206	N
$CF	207	O
$D0	208	P
$D1	209	Q
$D2	210	R
$D3	211	S
$D4	212	T
$D5	213	U
$D6	214	V
$D7	215	W
$D8	216	X
$D9	217	Y
$DA	218	Z
$DB	219	[
$DC	220	\
$DD	221	]
$DE	222	'
$DF	223	—
$E0	224	'
$E1	225	a
$E2	226	b
$E3	227	c
$E4	228	d
$E5	229	e
$E6	230	f

Hex	Dec	Key
$E7	231	g
$E8	232	h
$E9	233	i
$EA	234	j
$EB	235	k
$EC	236	l
$ED	237	m
$EE	238	n
$EF	239	o
$F0	240	p
$F1	241	q
$F2	242	r
$F3	243	s
$F4	244	t
$F5	245	u
$F6	246	v
$F7	247	w
$F8	248	x
$F9	249	y
$FA	250	z
$FB	251	[
$FC	252	/
$FD	253	]
$FE	254	'
$FF	255	Rubout

Bibliography

Andrews, Mark. *Atari Roots: A Guide to Atari Assembly Language.* Chatsworth, CA: Datamost, 1984.

———*The Apple IIe User's Guide.* New York: Macmillan, 1983.

———*Programming the Commodore 64/128 in Assembly Language.* Indianapolis, IN: Howard W. Sams & Co., Inc., 1985.

Apple II Reference Manual (For IIe Only). Cupertino, CA: Apple Computer, Inc., 1982.

The Apple IIc Reference Manual. Cupertino, CA: Apple Computer, Inc., 1984.

Apple Mouse IIc User's Manual. Cupertino, CA: Apple Computer, Inc., 1984.

AppleMouse II User's Manual (For the Apple IIe, II Plus, and II). Cupertino, CA: Apple Computer, Inc., 1983.

Extended 80-Column Text/AppleColor Adaptor Card Manual. Cupertino, CA: Apple Computer, Inc., 1984.

Findley, Robert. *6502 Software Gourmet Guide & Cookbook.* Rochelle Park, NJ: Hayden Book Co., Inc., 1979.

Leventhal, Lance A. *6502 Assembly Language Programming.* Berkeley, CA: Osborne/McGraw-Hill, 1979.

Little, Gary B. *Inside the Apple IIe.* Bowie, MD: Brady Communications Co., Inc., 1985.

Maurer, W. Douglas. *Apple Assembly Language.* Rockville, MD: Computer Science Press, Inc., 1984.

Pelczarski, Mark. *Graphically Speaking.* Geneva, IL: Softalk Books, 1983.

ProDOS Assembler Tools. Cupertino, CA: Apple Computer, Inc., 1984.

ProDOS Technical Reference Manual for the Apple II Family. Cupertino, CA: Apple Computer, Inc., 1983.

Wagner, Roger. *Assembly Lines: The Book: A Beginner's Guide to 6502 Programming on the Apple II.* Santee, CA: Roger Wagner Publishing, Inc., 1984.

Zaks, Rodnay. *Programming the 6502.* Berkeley, CA: Sybex, 1983.

Trademarks

The following names are trademarked products of the corresponding companies.

Apple®	Apple Computer, Inc.
Atari®	Warner Communications
Commodore 64™	Commodore Business Machines
The Complete Graphics System™	Penguin Software
CP/M®	Digital Research
The Graphics Magician™	Penguin Software
IBM®	International Business Machines, Inc.
Macintosh™	Apple Computer, Inc.*
Merlin Pro™	Roger Wagner Publishing, Inc.
ORCA/M®	The Byte Works, Inc.
Ping Pong®	Harvard Table Tennis
ProDOS®	Apple Computer, Inc.
Radio Shack®	Radio Shack, A Division of Tandy Corp.
Sourceror™	Roger Wagner Publishing, Inc.
Texas Instruments®	Texas Instruments, Inc.

*Macintosh is a trademark of Macintosh Laboratory, Inc., licensed to Apple, Inc., and is being used with expressed permission of its owner.

Index

Z

APPLE® ROOTS

ASSEMBLY LANGUAGE PROGRAMMING

For beginning programmers and experienced computer users who are unfamiliar with the 65C02 instruction set, **Apple® Roots** is a complete assembly language programming guide to the Apple® IIc and enhanced Apple® IIe.

Apple® Roots provides thorough coverage of

- Assembly language programming with the Apple® ProDOS® disk operating system.
- Apple ProDOS Assembler Tools and their usage.
- The 65C02/6502B instruction set.
- Binary numbers and 65C02 arithmetic.
- Assembly language graphics and sound.

With the detailed descriptions, numerous hands-on exercises, and practical examples in **Apple® Roots,** you'll soon be writing assembly language programs that take full advantage of your Apple IIc and enhanced IIe.

Mark Andrews has written several acclaimed books in his assembly language series including **Atari® Roots** (Datamost) and **C-64™ Roots** (Sams), as well as **The Apple® IIe User's Guide, The C-64™ User's Guide** and **The IBM® PC User's Guide** (Macmillan). In addition, he is a frequent contributor to **A+, Popular Computing, Electronics,** and other magazines.

- *Apple and ProDOS are registered trademarks of Apple Computer, Inc.*
- *C-64 is a trademark of Commodore Business Machines, Inc.*
- *IBM is a registered trademark of IBM Corp.*

ISBN 0-07-881130-9