SAMS 22444

Commodore 64™/128™ Assembly Language Programming

Mark Andrews

Commodore 64/128 Assembly Language Programming

Commodore 64/128 Assembly Language Programming

by Mark Andrews

Howard W. Sams & Co.
A Division of Macmillan, Inc.
4300 West 62nd Street, Indianapolis, IN 46268 USA

FIRST EDITION
THIRD PRINTING — 1986

International Standard Book Number: 0-672-22244-5
Library of Congress Catalog Card Number: 85-61476

Illustrated by: *Jill Martin*

Printed in the United States of America

Contents

Introduction: Ahoy, Commodore! vii
How This Is Different from Other Assembly Language Programming Books

Part One 1
Principles and Techniques of Assembly Language Programming

1 **Introducing Assembly Language** 3
What Assembly Language Is and What It's For

2 **By the Numbers** 19
The Binary, Hexadecimal, and Decimal Notation Systems

3 **In the Chips** 35
Inside Your Commodore

4 **Writing an Assembly Language Program** 45
Three Popular Commodore Assemblers

5 **Running an Assembly Language Program** 69
From a BASIC Program and on Its Own

6 **Addressing the Commodore** 91
Telling Your Computer Where to Go

7 **Looping and Branching** 109
GOSUBs and GOTOs in Assembly Language

8 **Programming Bit by Bit** 129
Single-Bit Operations on Binary Numbers

9 **Assembly Language Math** 143
Addition, Subtraction, Multiplication, and Division

Part Two 161
Assembly Language Graphics and Sound

10 **Memory Magic** 163
The Memory Map of the Commodore

11 **High-Resolution Commodore Graphics** 185
Joystick Operations

12 **Drawing Pictures in High-Resolution Graphics** 201
Some Advanced Features of Assembly Language

13 **Customizing a Character Set** 219
Copying Characters from ROM into RAM

14 **Programming Sprites in Assembly Language** 241
Animating Sprites on the Screen

15 **Commodore 64/128 Music and Sound** 261
An Introduction to Interrupt Operation

Appendix A 281
The 6510 Instruction Set

Appendix B 295
Additional Programs

Bibliography 314

Index 315

Introduction
Ahoy, Commodore!

How This Is Different from Other Assembly Language Programming Books

This book is different from others you may have seen in several ways:

- This is the first Assembly language book written for owners of both the Commodore 64 and the Commodore 128, the newest full-size computer in the Commodore line. All the programs, instructions, and examples in this book are applicable both to the Commodore 64 and to the Commodore 128, when the C128 is used in its C64 mode. Most of the programs will also run on the Commodore 128 in its C128 mode, since the main microprocessors in the C64 and the C128 are compatible. The C128 does have certain new and advanced features, such as 80-column color graphics and a built-in machine-language monitor, that are not specifically covered. However, this book will provide you with a thorough understanding of Commodore computers in general and the C64 and C128 in particular, so that you should have no trouble mastering the C128's special features on your own later.

- This book contains a large collection of Assembly language routines, many of them graphics-related, that are extremely useful, as well as interesting and entertaining. All these programs can be typed, assembled, and executed on either a Commodore 64 or a Commodore 128. If you type and assemble them while you read this book, as we suggest you do, you will have a useful library of Assembly language routines that you can incorporate easily into your own BASIC and Assembly language programs. There are programs for designing your own character sets, for writing joystick-controlled action games, and for drawing on the screen in high-resolution graphics. There is a program that prints headline-size characters on your monitor and one for creating animated sprite graphics. There are routines for using interrupts and raster interrupts and for programming music and sound. As a bonus, there is a collection of interactive tutorial programs for converting numbers from one base to another, for intermixing BASIC and machine-language programs, and, in short, for making your Commodore do just about everything it can. In other words, this is the most *complete* book available on Commodore Assembly language.

- As you read this book, you will notice that it is written in English, not computerese. It was written by a Commodore owner for Commodore owners, not by an electronics professor for engineering students or professional programmers. If you understand even just a little BASIC, then you will be able to understand this book. In fact, it may well be the easiest to understand Assembly language textbook around.

- You can also see that this is not just another reference book about the 6502 chip, the granddaddy of the 6510 microprocessor used in the Commodore 64 and the 8502 chip used in the Commodore 128. It also is not just another manual on 6502 Assembly language. It is specifically about Commodore Assembly language. So, when you use this book for its intended purpose, you will not have to try to figure out a 6502 Assembly language book, an editor/assembler instruction manual, and a *Commodore 64 Programmer's Reference Guide* all at the same time. Instead, for the very first time in one book, you will find a complete hands-on guide that tells you just about everything you need to know about Commodore 64 or Commodore 128 Assembly language programming.

- This book contains detailed instructions on how to use the *three* most popular Commodore 64 assemblers on the market: the Commodore 64 macro assembler, manufactured by Commodore; the Merlin 64 assembler, from Roger Wagner Publishing, Inc.; and the Panther C64, from Panther Computer Corporation.

- Finally, from start to finish, the format of this book is custom-tailored make the study of Assembly language as painless as possible. The following tells about how the book is designed.

ASSEMBLY LANGUAGE DEMYSTIFIED

In Chapter 1, you will be introduced to Assembly language and will learn exactly how Assembly differs from other programming languages.

In Chapter 2, you will start learning about bits, bytes, and binary numbers—the building blocks program designers use to create Assembly language programs. You will find some easy-to-use BASIC programs to convert numbers from one base to another automatically, which will help take the mystery out of hexadecimal and binary numbers.

In Chapter 3, you will learn all about the 6510/6502 microprocessor chip, the "brain" of your Commodore.

In Chapter 4, you will write your first Assembly language program.

If you are intrigued by these prospects—and you must be, or you would not have read even this far—then you are probably anxious to start learning Assembly language. And I am anxious to start helping you learn

Assembly language: the most fundamental, most exciting, and most rewarding of all computer programming languages.

GETTING YOUR GEAR TOGETHER

Before you set out on this journey, you will need a few pieces of equipment. I assume you already own the most important item, a Commodore 64 or Commodore 128 computer. Since most of the utility packages used in Assembly language programming are disk-based, you will also need a Commodore-compatible disk drive.

A Commodore 1520 or 1525 printer, or any other type of line printer compatible with your Commodore, will also come in handy. It does not have to be a letter-quality printer, but it should be capable of printing out readable listings of Assembly language programs.

You should also put a few standard reference books in your library, including two extremely important books published by Commodore. One of these books is the user's guide that came with your computer. The other important book is the *Commodore 64 Programmer's Reference Guide,* available both from Commodore and from Howard W. Sams & Co., Inc. The *Commodore 64 Programmer's Reference Guide* is an essential book for the Commodore Assembly language programmer. Other useful books are listed in the bibliography.

Finally, you need an assembler, a software package you will use to write the Assembly language programs in this book. Unless you own an assembler, you cannot write programs in Assembly language.

CHOOSING AN ASSEMBLER

There are many assemblers on the market, only a few of which are designed to be used with Commodore computers; only three of these were used to write the programs in this book. If you want to use this book to learn Assembly language, it probably would not be a bad idea to own at least one of the three assemblers that were used to write it. They are:

- The Commodore 64 Macro Assembler Development System
- The Merlin 64 assembler
- The Panther C64 assembler

These three assemblers are all quite different, each with its own advantages and disadvantages. The Commodore assembler is extremely powerful but quite difficult to use, especially for the beginner. The Panther assembler is much easier to use than the Commodore assembler, but it has a few quirks, and it has too many limitations to be used much in serious programming. The Merlin 64 is not quite as powerful as the Commodore assembler, but it is considerably more sophisticated than the Panther C64.

In trying to decide which of these three assemblers to use in writing this book, I never managed to make a really clear choice among the three. Mostly, I used the Goldilocks method: The Commodore was too big, the Panther was too little, but the Merlin was just right. So I wrote many though not all of the programs in this book with the Merlin 64. So, if I had to make a choice among these three assemblers, I would pick the Merlin 64. However, fairly complete instructions for using the Commodore and Panther assemblers are included in this book—and, with minor modifications, most of the programs in this book can be assembled and run using just about any other assembler that is compatible with the Commodore 64 or Commodore 128.

Now here are brief descriptions of the Commodore, the Merlin, and the Panther C64 assemblers.

THE COMMODORE 64 ASSEMBLER

One of the best is the Commodore 64 Macro Assembler Development System, manufactured by Commodore. This system is quite difficult to use, especially for beginners. Among the problems is the documentation that comes with the Commodore assembler: it is a cheaply reproduced sheaf of typed notes that raises more questions than it answers for the beginning-level Assembly language programmer. Despite this shortcoming, however, the Commodore assembler/editor system is a very sophisticated package with many advanced features, and once you gain a certain amount of experience, you should definitely acquire the Commodore 64 assembler. Commodore's own programmers use the system to develop programs, and many of the programs in this book were written on it. So, if you are really serious about learning Commodore Assembly language, keep the Commodore 64 Macro Assembler in mind.

The Commodore 64 assembler has a conventional, no-nonsense kind of design, so if you are familiar with other standard microcomputer assemblers, this one will not take much getting used to. Since it has been used by many professional programmers over a long period of time, it has been thoroughly debugged and is a solidly engineered, professionally designed program-development system. It was created specifically for Commodore computers, so it is perfectly tailored for Commodore machines. The Commodore 64 assembler was written by Commodore experts, so, unlike the two other assemblers that were used in the preparation of this book, it leaves free blocks of memory precisely where an experienced Commodore programmer would expect them to be. In short, the Commodore 64 Macro Assembler Development System is a fine assembler, with many features that one would never expect to find in such a low-priced package and with some capabilities that are difficult to find at any price.

Because of its powerful capabilities and its versatile design, the Commodore assembler can be used to write long, complex programs that can be stored almost anywhere in a Commodore computer's memory. The assembler is also fully equipped to handle macros—chunks of Assembly language code designed to carry out repetitive operations in a program. It comes with a DOS wedge, which is a special program that greatly simplifies the handling of Commodore 64 files. It is equipped with two machine-language monitors—one that can be loaded into a lower block and one that can be loaded into a higher block of RAM. Twin monitors are provided, according to Commodore, so that you can always use one without overwriting other utilities or user-written programs. Two machine-language loaders are also included, for the same reason.

As previously mentioned, though, the Commodore assembler does have one serious shortcoming. User-friendliness is not its strong suit: in fact, it can be downright user-hostile. It was obviously designed for professional programmers, not for beginners. Also, it was definitely not created for use with a computer system with just one disk drive. It consists of a number of different programs, including an editor, an assembler, a monitor, and a loader, each of which must be loaded and used separately, independently of all the rest. That involves a lot of disk-switching if your computer has just one disk drive—so if you decide to buy a Commodore assembler, it might also be a good idea to invest in a second disk drive.

Despite the annoying and sometimes frustrating complexity of the Commodore 64 assembler, however, it is probably the best assembler on the market for the computer for which it was created. If you are willing to put up with some hassles in exchange for owning one of the most powerful 6502-type assemblers, then I strongly suggest that you buy a Commodore 64 assembler.

THE MERLIN 64

The Merlin 64 is also an excellent editor/assembler package. Merlin is an imaginatively designed assembler, with a host of advanced features that are somewhat unconventional but quite useful—and surprisingly easy to learn and use. The Merlin instruction manual looks better than the one that comes with the Commodore assembler, but is poorly organized and, in places, very difficult to understand. On the positive note, the Merlin package includes an extremely sophisticated machine-language disassembler, called the Sourceror, plus a good-sized library of Assembly language routines, all stored on the program's master disk and ready to be incorporated into user-written programs.

Merlin can handle macros, of course, and it has linking features that allow the user to write and assemble source-code listings that would ordinarily be too long to fit into a Commodore 64's memory all at once. Merlin can be used in 80-column format on a computer equipped with a high-resolution monitor and an 80-column card.

The Merlin 64 is an adaptation of an assembler originally designed for Apple computers, and, perhaps as a result, it is less than ideally suited for Commodore computers. For example, it consumes quite a few blocks of memory space that really should be left free in a Commodore machine: while it is easy enough for an experienced programmer to reclaim the memory space that Merlin so thoughtlessly lays claim to, that would not be necessary if the assembler's memory layout had been better designed.

The good news about the Merlin assembler is that it is much, much easier to use than the Commodore assembler. When you boot the Merlin disk, all its important utilities are loaded at once, so repeated disk-switching is not necessary. Merlin is menu-driven—a nice touch for an assembler—and its file-management system is superbly designed and easy to use. Merlin is equipped with a DOS wedge, just as the Commodore assembler is—but Merlin's wedge loads automatically when you boot the master disk, while the Commodore wedge does not. In short, Merlin is much more user-friendly than the powerful but more complicated Commodore 64 macro assembler system.

So if you are new to Assembly language, you might want to start out with the Merlin assembler, graduating to the Commodore system later. You may even decide that you like Merlin so much that you want to stick with it for good. I own both the Merlin and the Commodore systems, and I enjoy working with both of them. I like the Commodore assembler, since I have mastered its idiosyncrasies and grown used to its standard, conventional-style approach to Assembly language programming, but I also like the less conventional Merlin assembler, and Merlin is my assembler of choice for fast, uncomplicated Assembly language programming.

THE PANTHER C64 ASSEMBLER

Of the three assemblers used in writing this book, the Panther C64 is the easiest to operate—but also the most limited. The Panther is a very a popular assembler, partly because of its simplicity. In fact, many Commodore 64 owners say it is their favorite assembler/editor system. However, I cannot agree with this. It has no macro capability, and it consumes more memory space than either the Commodore or the Merlin 64. Furthermore, I find its memory layout even less convenient than that of the Merlin. Some of the RAM the Panther system occupies is situated in spots where I like to put my own code, including one big block of memory ($C000–$CFFF) which the designers of the Commodore 64 purposely set aside for user-written Assembly language programs. The Merlin 64 also uses this block of memory, but the Panther—unlike Merlin—does not provide any easy way to reclaim it.

On the positive side, the Panther is even easier to use than the Merlin, everything it can do is directly accessible from the editor, and the editor's beautifully engineered error-checking system makes it almost impossible to write a bad line of code. The instruction manual that comes with the assembler, while rather uneven in quality, is much better than those provided with the Commodore and Merlin assemblers. It includes a very good tutorial section—in fact, the tutorial material in the Panther instruction book is so good that you might consider buying the assembler just to read the instruction manual!

The Panther assembler comes on a single disk. When you boot the disk, every program on it is automatically stored in your computer's memory. So the Panther C64 is very easy to get up and running. You can boot it once, replace the master disk with a data disk, and program all day long without ever having to switch disks again.

Once you have all your gear together, you will be ready to start programming in Assembly language, and we can move on to the first chapter.

Principles and Techniques of Assembly Language Programming

1 Introducing Assembly Language

What Assembly Language Is and What It's For

If you have done much programming in Commodore BASIC, you have probably wished at one time or another that you knew Assembly language.

You have opened the right book. 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 good working knowledge of machine language, your computer's native tongue. It will show you how to create programs that would be impossible to write in BASIC and other less advanced languages. Furthermore, it will do all of that in down-to-earth language that any BASIC programmer will understand.

You are about to discover that programming in Assembly language is not nearly as difficult as you may have thought it would be.

WHAT A LITTLE ASSEMBLY LANGUAGE CAN DO

If you know even a little BASIC, then you can learn to program in Assembly language. You will soon be able to:

- Write programs that run up to 1000 times faster than programs written in BASIC.
- Design high-resolution arcade games, featuring animation, fine scrolling and joystick action, that would be impossible to program in BASIC.
- Custom design your own screen displays, mixing text and graphics in any way you like on the screen.
- Create your own customized character sets, and display them in almost any size and color on your video screen.
- Intermix BASIC and Assembly language in the same program, combining the simplicity of BASIC with the speed and versatility of Assembly language.

- Use interrupts, raster interrupts, multi-voice sound-generation techniques, and many other advanced techniques that are often used by professional programmers.

Even more important, as you learn Assembly language, you will also be learning what makes computers tick. That will make you a better programmer in *any* language.

COMPUTER LINGUISTICS

To start at the beginning, programming languages can be divided into three major categories: high-level languages, machine language, and Assembly language.

There are many *high-level languages:* BASIC, Pascal, COBOL, FORTH, FORTRAN, C, PL/1, APL, SNOBOL, LISP, Ada, ALGOL, and hundreds more. High-level languages are not called that because they are particularly esoteric or profound. The term just means that these languages are several levels removed from machine language, the only language that a computer can really "understand."

At the other extreme, *machine language* is made up of nothing but numbers. It is obviously not the most programmer-friendly language you will ever encounter, but it is, so to speak, your computer's native tongue.

Assembly language, as you will soon see, is very closely related to machine language. In fact, Assembly language is not really a full-fledged programming language at all; it is actually just a notation system designed to make machine-language programs a little easier to write and understand.

Now we will take a closer look at each of these language families, beginning with high-level languages.

HIGH-LEVEL LANGUAGES

All popular high-level languages have one feature in common. They all bear at least a passing resemblance to English. BASIC, for example, is made up almost completely of instructions derived from English words, such as PRINT, LIST, LOAD, SAVE, GOTO, RETURN, and so on. Most other high-level languages also have instruction sets based largely upon natural-language (that is, *human* language) words and phrases.

Computers cannot work with English; the only language they can really "understand" is machine language. Therefore, when you write a program in BASIC or in any other high-level language, it has to be translated into machine language before the computer can understand it. So programmers use special kinds of software packages, called *interpreters* and *compilers,* to help them translate the programs they have written into machine language.

Interpreters are the easiest kinds of translation aids to use, because they are "transparent" to the user. That is, when you write a program with an interpreter, you are usually not even aware of the fact that it is there. The BASIC that is built into your Commodore is an interpreter, and if you have ever used it, you have seen how transparent a good BASIC interpreter can be. In Commodore BASIC, your computer's built-in interpreter translates into machine language every line of code as the line is executed. It does the job so smoothly that you have probably never even noticed that any BASIC-to-machine language translating was going on.

A compiler is a little more difficult to use than an interpreter. A compiler does not usually translate a program into machine language line by line. Instead, it generally takes a complete program, or at least a reasonably large chunk of source code, and translates the whole thing into machine language—all at once. Then you can store on disk the machine code that has been generated by the compiler. From that point on, the compiled program can be run like any other machine-language program. So, once a program has been compiled, it never has to be compiled again, unless you change it. From then on, it can be run at any time, without further need for a compiler.

DIFFERENCES AT RUN TIME

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. Compilers are not that interactive. Most compilers cannot check a program for errors until the program is compiled.

But compilers 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 is run. But a compiler has to do its job just once, and never has to be used when a program is actually running. So programs written with interpreters usually run more slowly than programs written using compilers.

One reason that BASIC is a slow-running language is that it is almost always translated into machine language with an interpreter. A few other programming languages, such as LOGO and PILOT, are also interpreted languages. COBOL, Pascal, and most other high-level languages are usually compiled. Recently, however, varieties of BASIC have been introduced that are designed to be used with compilers to increase running speed.

Is Assembly an interpreted or a compiled language? It is neither. To convert Assembly language programs into machine language, programmers use a software package called an *assembler.* We will return to the subject of interpreters later on in this chapter.

MACHINE LANGUAGE

Machine language, as mentioned, is made up of nothing but numbers. In its purest form, in fact, machine language is composed of *binary numbers,* numbers written as strings of ones and zeroes. Here is a listing of a machine-language program written in binary numbers. This program, for reasons that you will discover very shortly, is called HI.TEST. Its binary-code version is called HI.TEST.BIN.

```
HI.TEST.BIN
(THE HI.TEST PROGRAM, BINARY-CODE VERSION)

10101001
01001000
00100000
11010010
11111111
10101001
01001001
00100000
11010010
11111111
01100000
```

After seeing these numbers, you may feel that when you have seen one binary number, you have seen them all. Binary numbers look so much alike that it is very difficult to distinguish one from another, even for those familiar with machine language. So machine-code listings are rarely written in binary numbers. Instead, they are usually written in a closely related notation called the *hexadecimal system*.

The hexadecimal system is based on the value 16, unlike our familiar decimal system, which is of course based on 10. In hexadecimal notation, the arabic numbers 0 through 9 represent the same values as in decimal notation. In addition, however, the letters A through F are used to represent the decimal values 11 through 16. In later chapters, you will learn more about the hexadecimal system and why it is used in Assembly language programs. Just so you will know what hexadecimal numbers look like, here is the HI.TEST program, written in hexadecimal numbers:

```
HI.TEST.HEX
(THE HI.TEST PROGRAM, HEX-CODE VERSION)

A9 48
20 D2 FF
A9 49
20 D2 FF
60
```

The hex numbers in this version of the HI.TEST program have the same values as the binary numbers that were used in the binary version of the program. Even without understanding yet what the HI.TEST program means, you can now see quite clearly that the hexadecimal version of the program is at least a little easier to read than the binary version.

Now that we have converted the HI.TEST into hexadecimal numbers, only one more step is needed to translate it into Assembly language.

ASSEMBLY LANGUAGE

Although Assembly language is very closely related to machine language, the relationship is not always obvious at first glance. Assembly language is *not* made up solely of numbers, as machine language is. Instead, it is written using three-letter instructions called *mnemonics*. So, to the casual observer, Assembly language does not look at all 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. *There is a precise one-to-one correlation between the mnemonics used in Assembly language and the numeric instructions used in machine language.*

Because of this close relationship, it is easy to convert a machine-language program into Assembly language and to convert an Assembly language program into machine language. To translate a program from either language to the other, change each Assembly language instruction into the machine-language instruction that means the same thing, or vice versa.

TWIN LISTINGS The following two listings of the HI.TEST program illustrate the close relationship between machine language and Assembly language:

```
HI.TEST.ASM
(OBJECT CODE AND SOURCE CODE COMPARED)

LINE NO.  OBJECT CODE  SOURCE CODE

1         A9 48        LDA    #72
2         20 D2 FF     JSR    $FFD2
3         A9 49        LDA    #73
4         20 D2 FF     JSR    $FFD2
5         60           RTS
```

Look carefully at this pair of listings, and you can see some close similarities. Although the letters and numbers in the two listings are arranged slightly differently, a close examination reveals certain similar patterns in both listings. In the object-code listing, for example, you see that the machine-language instruction A9 is used twice: once in Line 1, and again in Line 3. In the source-code listing, the Assembly language mnemonic LDA is also used twice, on the same lines and in the same positions as the machine-language instruction A9. You might guess, then, that the object-code instruction A9 corresponds to the source-code instruction LDA. As it turns out, that is true.

In the object-code listing, you can see that the machine-code instruction 20 is also used twice. In both cases, it corresponds to the source-code instruction JSR.

Now you have had a first-hand look at how Assembly language compares with machine language. You have now seen that there is some kind of close correlation between the two. That is all I wanted to get across in this example. So now we can leave the topic of machine language for a while and look at the Assembly language version of the HI.TEST program:

```
HI.TEST.SRC
(THE HI.TEST PROGRAM, SOURCE-CODE VERSION)

COLUMN 1   COLUMN 2
LINE NO.   SOURCE CODE

1          LDA   #72
2          JSR   $FFD2
3          LDA   #73
4          JSR   $FFD2
5          RTS
```

We start our examination of this listing by looking at Column 2, which contains the hexadecimal number FFD2 (preceded by the symbol $) and the decimal numbers 72 and 73 (preceded by the symbol #). In Commodore Assembly language, the number 72 is a screen-display code that corresponds to the letter H. The number 73 is a screen-display code for the next letter in the alphabet, I. The hexadecimal number FFD2 (65490 in decimal notation) is the starting address of a machine-language subroutine built into the Commodore 64: a subroutine that prints a character on the screen.

When the symbol # precedes a number in Commodore Assembly language, it means that the number is to be interpreted as a literal number, not as a memory address. In the HI.TEST program, if the numbers 72 and 73 were not preceded by the symbol #, they would be interpreted as addresses in your computer's memory. But, since they have the # prefix, they are interpreted as actual numbers.

The other special symbol in the HI.TEST program, the dollar sign in front of the number FFD2, is the Assembly language prefix for hexadecimal numbers. Sometimes decimal numbers and hex numbers look exactly alike. So, in the HI.TEST program, the $ prefix is used to show that the number $FFD2 is a hexadecimal number.

Note, however, that the # is not used in front of the number $FFD2. In this program, $FFD2 is to be interpreted as a memory address, not as a literal number. In the Commodore 64, as mentioned, $FFD2 is the memory address of a built-in subroutine (called in lines 2 and 4 of HI.TEST.SRC) that prints a character on the screen.

ASSEMBLY LANGUAGE MNEMONICS

Column 1 of the HI.TEST.SRC program contains three Assembly language instructions: the mnemonics LDA, JSR, and RTS. We will now examine each of these mnemonics with the help of a line-by-line analysis of the HI.TEST.SRC program.

```
1  LDA #72
```

When you write a program in Assembly language, what you are actually doing is programming the 6510/8502 chip, your computer's main microprocessor. So, before you can start programming in Assembly language, you will have to know a few important facts about your computer's central processing unit, or *CPU.*

Inside your Commodore's 6510/8502 chip are several *internal registers.* You can store data in these registers, much as you store data in the memory registers in your computer's ROM and RAM. However, the internal registers in the 6510/8502 chip have some special features that ordinary memory registers do not have. The various functions and features of the 6510/8502's internal registers will be covered in detail in later chapters. Before we go any further, though, a few words about one specific 6510/8502 register: a very special register called the *accumulator.*

The accumulator is the busiest register in the 6510/8502 chip. Before any mathematical or logical operation can be performed on a number in 6502/6510/8502 Assembly language, the number first has to be loaded into the accumulator. The Assembly language instruction that is usually used to load a number into the accumulator is *LDA.*

In Line 1 of HI.TEST.SRC, the statement "LDA #72" means "Load the accumulator with the literal number 72." In Commodore Assembly language, as mentioned, the number 72 is a screen code for the letter H. So, Line 1 means "Load the accumulator with the screen code for the letter H."

```
2   JSR $FFD2
```

In 6502/6510/8502 Assembly language, the mnemonic JSR means "Jump to subroutine." This instruction is used much as GOSUB 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 starts at whatever memory address follows the JSR instruction.

In Assembly language, JSR is usually used along with another mnemonic, RTS. RTS 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 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, it is usually expected to end 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.

So, in Line 2 of the HI.TEST.SRC program, the statement "JSR $FFD2" means "Jump to the subroutine that begins at Memory Address $FFD2." This subroutine finds whatever value is stored in the accumulator and automatically prints on the screen the character that corresponds to that value. Then it returns control to whatever program is in progress, in this case, to the HI.TEST program.

A number of handy I/O routines that work much like this one are built into the Commodore 64. And we will be using quite a few of them in this book.

```
3  LDA #73
```

In Commodore Assembly language (and Commodore BASIC, too), the number 73 is a screen code for the letter I. So, in the HI.TEST.SRC program, the statement "LDA #73" means "Load the accumulator with the screen code for the letter I."

```
4  JSR $FFD2
```

This statement is identical to the statement in Line 2: "Jump to the subroutine that starts at Memory Address $FFD2." This time, however, since the accumulator has been loaded with the value 73, the subroutine that starts at $FFD2 will cause the letter I to be printed on the screen.

```
5  RTS
```

RTS, you recall, is an Assembly language mnemonic that means "Return from subroutine." When RTS is used to terminate a subroutine, it usually causes a program to jump back to where it left off before the subroutine was called. In this case, however, RTS is used in a slightly different way: to terminate a whole program, rather than just a subroutine. When RTS is used in this fashion, it usually returns control of the computer to whatever program or system was in control before the program began. So, if you were to call the HI.TEST program from BASIC, the RTS instruction in Line 5 would transfer control to your computer's BASIC interpreter.

RUNNING THE HI.TEST PROGRAM

Before an Assembly language program can be executed, it is necessary to assemble it into machine language. It is possible to assemble a program by hand; in fact, before automatic assemblers came along, that is how all programs were assembled. Of course, the easiest way to assemble a program is with an assembler.

HOW AN ASSEMBLER WORKS

An assembler has a much more straightforward job than either an interpreter or a compiler. Interpreters and compilers have to go through all kinds of manipulations to translate programs written in high-level languages into machine language. An assembler merely converts each mnemonic in an Assembly language program into a corresponding machine-language instruction. The object code produced by this process can then be stored on a disk. Once that is done, the program's original source code can be put away for safekeeping and need never be assembled into machine language again.

Does that mean that an assembler works just like a compiler? Not exactly. The main difference is that an assembler translates source code word for word into object code; the compiler's task is much more complex. As you have seen throughout this chapter, the instructions used in Assembly language perform much simpler—that is, more fundamental—functions than the instructions that are used in most high-level languages. Consequently, Assembly language programs tend to be much wordier than programs written in high-level languages. However, the Assembly instruction set is also quite versatile, since the mnemonics can be combined with each other in an almost endless variety of ways.

Assembly language is also very memory-efficient. Since Assembly language programs are assembled into machine language one instruction at a time, and not translated into chains of instructions by interpreters and compilers, the machine code produced by an Assembly language program is less repetitious than the code produced by interpreters and compilers. This is 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 strung together, one after another, like clothes hanging on a line. If the same instruction is repeated over and over in a program written in a high-level language, the interpreter or compiler that converts the program into machine code typically repeats the same sequence of code over and over again, usually wasting both memory and processing time. In contrast, a good Assembly language programmer usually writes an important block of code just once during the course of a program, then uses it as a subroutine from that point on, conserving memory and cutting down on processing time.

WHAT EVERY GOOD PROGRAMMER SHOULD KNOW

Unfortunately, a certain price must be paid for all this speed, versatility, and memory efficiency. That price is usually exacted from the programmer. Because it is not designed to do many things automatically, Assembly language imposes quite a few more demands on the programmer than most high-level languages do. For example, before you write an Assembly language program, you have to decide exactly where you want the program to be stored in the computer's memory. If you try to store it in the wrong place, it may overwrite other important programs, such as your computer's operating system, screen memory, or disk operating system. The result of this could be complete disaster.

Before you can become a good Assembly language programmer, you also need a good understanding of how your computer works. Later on in this book, when you start writing Assembly language programs, you will actually be programming your computer's central microprocessor: the 6510/8502 chip that is built into your Commodore. So, before we run the HI.TEST program and end this chapter, let's take a quick look at how the 6510/8502 chip works and how it works with the rest of your computer system.

INSIDE YOUR COMMODORE

Every microcomputer can be divided into three parts:

1. A *central processing unit* (CPU). As its name implies, this is the central component in a computer system, in which all computing functions take place. In a microcomputer, which is what your Commodore is, all of the functions of a central processing unit are contained in a microprocessor unit (MPU). Your Commodore computer's MPU (as well as its CPU) is a *very large-scale integrated circuit* (VLSI), a 6510 chip if you own a Commodore 64 and an 8502 chip if you own a Commodore 128.
2. *Memory,* which can be further divided into RAM (random-access memory) and ROM (read-only memory).
3. *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 Commodore can be connected to, or, to use one unavoidable jargon term, can be interfaced with, include telephone modems, graphics tablets, cassette data recorders, and disk drives.

Figure 1-1 illustrates the architecture of one fairly typical microcomputer—namely, your Commodore.

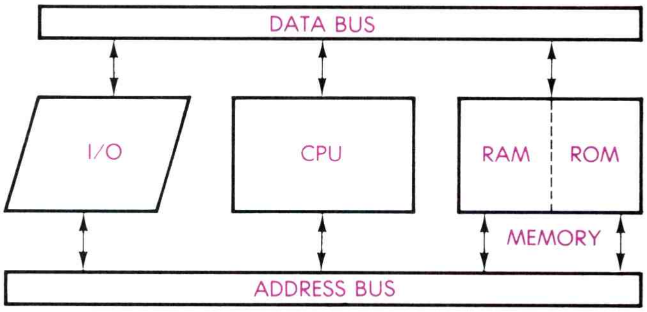

Figure 1-1 Block Diagram of a Microcomputer

Now let's examine each of the three major ingredients of your Commodore system. We will start with your computer's 6510/8502 microprocessor: its CPU.

THE 6510/8502 FAMILY

The 6510/8502 microprocessor is an improved and updated version of an earlier chip, the 6502, developed by MOS Technology, Inc. Several companies, including Commodore, are now licensed to manufacture the 6502 and other chips based on the 6502, such as the 6510 chip used in the Commodore 64 and the 8502 chip used in the Commodore 128. The 6502 and chips based on the 6502 are used not only in Commodore computers, but also in personal computers manufactured by Apple, Atari, Commodore, Ohio Scientific, and a number of other companies.

There is only one significant difference between the original, no-frills 6502 chip and the 6510 chip used in the C64. This difference is that the 6510/8502 chip has some special I/O capabilities that the 6502 lacks; these will be covered in a later chapter, so there is no need to discuss them now. For the moment, it is sufficient to point out that, from the point of view of instruction sets, there is no difference between the 6502 chip and the 6510/8502 chip used in your Commodore. Both chips are designed to be programmed in an Assembly dialect commonly known as 6502 Assembly language. So, once you learn how to write programs in 6502/6510/8502 Assembly language, you will be able to program many different kinds of personal computers, not only those made by Commodore.

Even more important, the *principles* of Commodore Assembly language programming are universal: they are the same principles *all* Assembly language programs use, no matter what computers they are written for. Once you learn 6510/8502 Assembly language, therefore, it will be easy to 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, the IBM PC, and many more.

Now let's take a look at your Commodore's memory.

ROM

The big difference between RAM and ROM is that RAM can be erased and ROM cannot be. Every time you turn your computer off, everything stored in RAM is immediately wiped out. But when you turn your computer on again, everything that was in ROM is still there.

ROM cannot be wiped out because it is permanently etched into a bank of memory chips inside your Commodore. So it is as permanent a part of your computer as the keyboard. In computer jargon, ROM is *nonvolatile;* RAM is *volatile*.

The most important part of your computer's ROM is the block of memory that holds its *operating system* (OS). The operating system is really just a long machine-language program, but what a program! Thanks to the machine-language routines in its ROM, your computer can generate text characters, display colors, accept keyboard inputs, and operate I/O devices, such as printers, video screens, and disk drives. Your computer's BASIC interpreter also resides in ROM. As you will see later in this book, there are ways to access and use many of these routines that are built into your computer's operating system from your own Assembly language programs.

RAM

Your Commodore's ROM package is the result of a lot of work by a lot of Assembly language programmers. RAM, on the other hand, can be filled up by anybody. Most of your computer's available memory space is dedicated to RAM.

Most of the RAM in your Commodore is available for use by user-written programs.

When you turn your computer on, the block of memory inside it that is reserved for RAM is as empty as a blank sheet of paper. Every time you write a program or load a program from a disk, that program is always stored somewhere in RAM.

Turn your computer off, and everything you have stored in RAM will suddenly disappear. That is why the Commodore can be used with cassette data recorders and disk drives. After you have written a program, you have to store it on some kind of mass-storage medium, such as a disk or a cassette, if you do not want it to be wiped out the next time the power goes off and erases your RAM.

Your computer's RAM, or main memory, can be visualized as a huge grid made up of thousands of compartments, or cells, something like tiers upon tiers of post-office boxes along a wall. Each cell in this vast memory matrix is called a *memory location,* or a *memory register,* and each memory register, like each box in a post office, has an individual and unique *memory address.*

The analogy between computers and post-office boxes does not end there. A program, like an expert postal worker putting mail in post-office boxes, can get to any one location in its memory about as quickly as it can get to any other. In other words, it can access any location in its memory at *random,* without having to start at the beginning each time. That is why user-addressable memory in a computer is known as *random-access memory.*

RUNNING A MACHINE-LANGUAGE PROGRAM

There are more than 64,000 memory registers in your Commodore. A machine-language program is just a series of numbers that fills a given block of these memory registers. When you load a machine-language program into your computer, it fills a block of consecutively numbered memory registers with certain values. When you run a machine-language program, you have to tell your computer where this block of values begins.

THE ORIGIN DIRECTIVE

In 6502/6510/8502 Assembly language, the most common way for telling an assembler where a program begins is with an *origin directive,* or *origin pseudo-op.* In Assembly language programs written on the Merlin 64 assembler, an origin directive looks like this:

```
ORG $8000
```

The Commodore 64 Macro Assembler system uses a slightly different kind of origin directive. Here is the origin directive for a Commodore 64 assembler:

```
*=$8000
```

Some assemblers, such as the Commodore 64, require origin lines, and some, such as the Merlin 64, do not. If you do not use an origin line with the Merlin assembler, your assembler will declare a default origin setting of $8000. However, even if you use a Merlin 64, get into the habit of writing your own origin directives, since most assemblers require them.

This is what the HI.TEST.SRC program looks like, written with a Merlin 64 assembler, using an origin line specifying a starting address of $8000:

```
HI.TEST.SRC
(THE HI.TEST.SRC PROGRAM, WITH AN ORIGIN DIRECTIVE)

ORG $8000
LDA #72
JSR $FFD2
LDA #73
JSR $FFD2
RTS
```

POINTS TO REMEMBER

Later, when you start writing Assembly language programs, you will learn exactly how the origin directive works. For now, remember two facts about origin lines:

1. Every Assembly language program should start with an origin directive.
2. When an Assembly language program is converted into machine language, its origin directive tells your computer the memory address where your machine-language program will begin.

When you load a machine-language program into RAM and tell the computer where to find it, the computer checks the memory register specified in the program origin directive when the Assembly program was originally written. When your computer has gone to the address specified in its origin directive, it fetches the number that has been stored in that memory register.

Now, matters get complicated, because numbers stored in a computer's memory registers can be interpreted in various ways. Although each memory location in your Commodore's memory can hold only one number, that number can represent at least three different things:

1. The stored number itself.
2. A code representing a typed character.
3. A machine-language instruction.

Since a number in a memory register can have any of these meanings, the computer needs a program to figure out how to process each number (or, more commonly, the various sequences of numbers) that it encounters as it processes a machine-language program. That, in a nutshell, is what machine-language programming is all about. When you write a machine-language program, you are really just filling the computer's memory with numbers, then telling your computer what all of those numbers mean. This is actually what you will be learning about for the rest of this book.

PROCESSING EXECUTABLE CODE

When your computer goes to the memory location identified as the starting address of a program, it is supposed to find the beginning of a block of executable code, that is, the beginning of a machine-language program. If it does find a program, it carries out the first instruction in the program, then moves on to the next consecutive address in memory. It then carries out that instruction, moves on to the next memory address, carries out the instruction it finds there, and so on.

The computer keeps on doing this, carrying out an instruction and then moving on to the next one, until it either reaches the end of the program or encounters an instruction telling it to jump to another address.

LIVING WITHOUT AN ASSEMBLER

Now we are ready to run the HI.TEST program we have been working with in this chapter. When we have done that, we can move on to the next chapter.

Before we run HI.TEST, however, we must resolve one small problem. Since we have not yet covered the use of assemblers, it is difficult to show you now how to assemble and execute the HI.TEST program, using your Merlin, Commodore, or Panther assembler. To get around this problem, here is a little trick you can use when you want to create and run a machine-language program without having to use an assembler. You just invoke a series of POKE statements from within a BASIC program.

To run a machine-language program from BASIC, the instructions in the program have to be written in decimal numbers. Then you can use RUN and DATA statements to poke each instruction in the program into a series of free memory registers. Finally, you can use the SYS command in Commodore BASIC to execute your machine-language program.

We will do that right now. Here is the HI.TEST program, converted into decimal numbers, translated into BASIC and ready to be run as a BASIC program:

```
HI.TEST.BAS
(THE HI.TEST PROGRAM, BASIC VERSION)

10  REM   *** HI.TEST.BAS ***
20  DATA  169,72,32,210,255,169,73,32,210,255,96
30  FOR L = 49152 TO 49162:READ X:POKE L,X:NEXT L
40  SYS 49152
```

Here is how the HI.TEST.BAS program works. The data in Line 20 corresponds to a series of machine-language instructions. The loop in Line 30 pokes each numeric instruction in Line 20 into a block of RAM that extends from Memory Address 49152 to Memory Address 49162 ($C000 to $C010 in hexadecimal notation). Finally, the SYS command in Line 40 executes the machine-language program.

You can type and run HI.TEST.BAS just as you would any BASIC program. Almost any machine-language program can be converted into BASIC and run in this manner.

Once you have typed and run HI.TEST.BAS, we will be ready to move on to Chapter 2.

2 By the Numbers

The Binary, Hexadecimal, and Decimal Notation Systems

As you may have noticed while reading Chapter 1, three different number systems are used in Assembly language programming. They are:

1. *Decimal numbers,* which are based on the value 10 and are written using the arabic numerals 0 through 9.
2. *Binary numbers,* which are based on the value 2 and are usually written using the arabic numerals 0 and 1. In binary notation, "1" means 1, "10" means decimal 2, "11" means decimal 3, and so on. Binary numbers are often used in Assembly language programs because they are the only kind of numbers computers can actually "understand." The data a computer processes is a stream of on-and-off pulses; on and off can be easily represented by the ones and zeroes used in the binary notation system.
3. *Hexadecimal numbers,* which are based on the value 16 and are customarily written using the arabic numbers 0 through 9 plus the letters A through F. In the hexadecimal notation system, the letters A through F are used as single-unit symbols for the values 10 through 15. Hexadecimal numbers can be translated easily into binary numbers.

When a hexadecimal number appears in a program, the prefix $ is used to indicate that it is a hexadecimal number. When a binary number appears in a Commodore Assembly language program, the prefix % is used to distinguish it from a decimal or 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 assumed to be a decimal number.

Here is an example of how prefixes are used to distinguish among binary, hexadecimal and decimal numbers in Assembly language programs:

1101—the decimal number 1,101

%1101—the binary number 1101 (decimal 13)

$1101—the hexadecimal number 1101 (decimal 4,353)

Decimal numbers are familiar, so not much will be said about them in this chapter. This chapter will be devoted to explanations of the binary and hexadecimal number systems. Let's start with the binary system.

USING BINARY NUMBERS

When data is loaded into a computer, it is usually transmitted to the computer in the form of on-and-off pulses. Inside a computer, these on-and-off pulses cause the current flowing through various electrical lines to fluctuate back and forth between *low* and *high* levels. When the electrical current flowing through a line falls below a certain predetermined level, the switch is considered off, and its state is represented as *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 *1*.

In the binary notation system, the ones and zeroes that make up binary numbers are known as *bits*. A series of four bits is called a *nibble*, a series of eight bits is called a *byte*, and a series of 16 bits is called a *word*.

Now let's see how binary numbers are actually used in Assembly language programming.

PENGUIN MATH

One way to explain the principles of the binary system is with what I call *Penguin Math*. Penguin Math is the number system that penguins might use if they could use numbers.

To get an idea of how Penguin Math works, just imagine that you are a penguin. You do not have 10 fingers on each hand, so there is no way you can count on your fingers. Instead, you have just two flippers, and if you want to count on something, you have to count on them.

Now suppose that you are a very bright penguin and you somehow do learn to count on your flippers. You can't count to 10 on your flippers, as people can when they count on their fingers. But you can count to 2.

Now suppose that you are the smartest penguin on Penguin Island. One day you manage to figure out how to use your flippers to count past 2. Instead of counting on your flippers in exactly the same way that humans count on their fingers, you decide to equate a raised right flipper to 1, and a raised left flipper to 2. Then you let two raised flippers represent the value 3.

Now suppose you are a genius among penguins and devise a new mathematical system to express in writing what you have done.

You could use a 0 to represent an unraised flipper, and a 1 to represent a raised one. Then you could scratch these equations in the ice:

```
FLIPPER COUNTING, VERSION 2

%00 = 0
%01 = 1
%10 = 2
%11 = 3
```

FEET, TOO

Those are, of course, binary numbers. They clearly show that if you were a really smart penguin, you could use two flippers to express values—0 through 3. That is a clear improvement over the two values that can represented with more traditional flipper-counting methods.

Since our imagination has taken us this far, let's now suppose that you (still as a penguin) want to learn to count past 3. While pondering this problem, you look down at your feet—and notice two more aids to counting down there. *Voilà*—bigger numbers.

By using both flippers and both feet at the same time, you count as follows:

```
FOUR-BIT PENGUIN MATH

%0000 = 0
%0001 = 1
%0010 = 2
%0011 = 3
%0100 = 4
%0101 = 5
%0110 = 6
%0111 = 7
%1000 = 8
%1001 = 9
```

and so on.

If you continued counting like this, you would eventually discover that you could express 16 values—0 through decimal 15—using four-digit (or four-bit) binary numbers.

ONE MORE LESSON

Now we are ready for one last lesson in Penguin Math. Imagine that you, as a penguin, have gotten married to another penguin. You and your spouse have a total of eight flippers between you.

If your spouse decided to cooperate with you in counting with flippers, the two of you could now use a set of numbers that looked like these:

```
0000 0001 = 1
0000 0010 = 2
0000 0011 = 3
0000 0100 = 4
0000 0101 = 5
```

and so on.

If you and your spouse kept on counting in this fashion—using 8-bit Penguin Math—you would eventually discover that by using eight flippers, you could could count from 0 to 255, for a total of 256 values. The point of all this is that it is possible to express 256 values—from 0 through 255—using 8-bit binary numbers.

BITS, BYTES, AND NIBBLES

As pointed out at the beginning of this chapter, when ones and zeroes are used to express binary numbers, they are known as bits. Now we are going to take a closeup look at a series of 8-bit binary numbers. Examine the numbers in this list closely, and you will see that every binary number that ends in zero is twice as large as the previous round number:

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

Here are two more numbers that are also noteworthy, but for completely different reasons:

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

The number %11111111, or 255, is worth remembering because it is the largest 8-bit binary number. The number %11111111 11111111, or 65,535, is the largest 16-bit binary number. (The space in the middle of the number %11111111 11111111 is there just to make the number easier to read. Spaces are often inserted in the middle of 8-bit numbers, too, 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 notation system for representing computer data. But, as we saw in Chapter 1, binary numbers have one serious shortcoming: they are extremely difficult to read. So the hexadecimal, rather than the binary, system is most often used in Assembly language programming.

Just as binary numbers are based on the value 2, hexadecimal numbers are based on the value 16. If people had eight fingers on each hand, we would probably all count using hexadecimal numbers.

Hexadecimal numbers are often used in Assembly language programming because they 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. You will see how this all works later on in this chapter.

WHAT HEX NUMBERS LOOK LIKE

The hexadecimal notation system uses not only the digits 0 through 9 but also the letters A through F. Table 2-1 shows what the numbers 1 through 16 look like in the hexadecimal notation system.

Table 2-1 DECIMAL/HEXADECIMAL CONVERSION

DECIMAL	HEXADECIMAL	DECIMAL	HEXADECIMAL
1	1	9	9
2	2	10	A
3	3	11	B
4	4	12	C
5	5	13	D
6	6	14	E
7	7	15	F
8	8	16	10

Look at the last two pairs of numbers in this table, and you will see that the hexadecimal number $F corresponds to the decimal number 15, and that the hex number $10 corresponds to the decimal number 16.

You can also see that odd-looking letter-and-number combinations like FC1C, 5DA4, and even ABCD are perfectly good numbers in the hexadecimal system.

Now let's look at the relationship between binary and hexadecimal numbers.

COMPARING BINARY AND HEXADECIMAL NUMBERS

Binary numbers, as we have seen, have a base of 2. Decimal numbers have a base of 10. Hexadecimal numbers have a base of 16, or 2 to the fourth power.

Table 2-2 is a chart comparing decimal, hexadecimal, and binary numbers.

Table 2-2 DECIMAL/HEXADECIMAL/BINARY CONVERSION

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

As you can see from this table, the decimal number 16 is written as 10 in hex and 00010000 in binary, and thus is a round number in both the binary system and the hexadecimal system. And the hexadecimal digit F, which comes just before hex 10 (or 16 in decimal), is written as 00001111 in binary.

As you become more familiar with the binary and hexadecimal systems, you will begin to notice many other similarities between these two numeric systems. 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 as 11111111 11111111 in binary and FFFFFFFF in hex.

GETTING TO THE POINT

The point of all this is that it is much easier to convert back and forth between binary and hexadecimal numbers than it is to convert between binary and decimal numbers. This is especially true when you are dealing with 16-bit numbers.

Table 2-3 shows how convenient it is to convert back and forth between the binary and hexadecimal systems. Notice how much more difficult it is to detect the relationship between a decimal number and its equivalents in the other two systems.

Table 2-3 HEXADECIMAL/DECIMAL RELATIONSHIPS

binary	1111	1100	0010	1110	1011	1000	0001	1100
hexadecimal	F	C	2	E	B	8	1	C
decimal	252		46		184		28	

As you can see from this table, an eight-bit number written in binary notation can always be equated to two hexadecimal digits. But there is no clear relationship between the length of a binary number and the length of the same number written in decimal notation.

This same principle can be extended to longer numbers. For example:

```
1111 1100 0001 1100 binary
F    C    1    C    hexadecimal
64540               decimal
```

CONVERTING FROM ONE SYSTEM TO ANOTHER

Since hexadecimal numbers, decimal numbers, and binary numbers are all used extensively in Assembly language programming, it would obviously be handy to have a tool to convert numbers back and forth among these three numeric systems. Fortunately, a number of such tools are available:

- Texas Instruments makes an extremely useful calculator called the *Programmer,* which can perform decimal/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 some similar calculator. The TI *Programmer* does not cost much, and it is well worth the money.
- Many books on Assembly language contain charts that you can consult when you want to convert numbers from one notation system to another. You will find a few such charts in this chapter, and you will also find something much better: a series of BASIC programs that will automatically perform decimal/hexadecimal, decimal/binary, and binary/hexadecimal conversions.

Let's start with a program that converts binary numbers to hexadecimal numbers.

BINARY/DECIMAL CONVERSIONS

It is not very difficult to convert a binary number to a decimal number. In a binary number, the bit farthest to the right represents 2 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 rightmost digit. The rightmost bit—often referred to as Bit 0—represents 2 to the 0th power, or the number 1. The leftmost bit—often called Bit 7—is equal to 2 to the 7th power, or 128.

Table 2-4 illustrates what each bit in an 8-bit binary number means.

Table 2-4 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

This table provides an easy method of converting any 8-bit binary number into its decimal equivalent. Instead of writing the number down from left to right, write it instead in a column, with Bit 0 at the top and Bit 7 at the bottom. Multiply each bit in the binary number by the decimal number it represents. Then add the results of these multiplications. The total is the decimal value of the binary number.

Suppose, for example, that you wanted to convert the binary number 00101001 into a decimal number. Table 2-5 shows how to do it.

Table 2-5 BINARY TO DECIMAL CONVERSION

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

If the calculation in this example is correct, then the binary number 00101001 should be equivalent to the decimal number 41. Look up either 00101001 or 41 on any binary to decimal or decimal to binary conversion chart, and you will see that the calculation was accurate. This conversion technique will work with any other binary number.

Now we will go in the other direction, and convert a decimal number to a binary number. First, divide the number by 2. Then write down both the quotient and the remainder. Since you are dividing by 2, the remainder will be either a 1 or 0. So what you write down will be the quotient followed by either a 1 or a 0.

Next, divide that quotient by 2, and write down the new quotient and remainder, as before. Write the 0 or 1 remainder right underneath the first remainder.

When there are no more numbers left to divide, write down all the remainders, starting at the bottom. This, of course, is a binary number—a number made up of ones and zeroes. That number is the binary equivalent of the decimal number you started out with.

This conversion technique is illustrated in Table 2-6.

Table 2-6 DECIMAL TO BINARY CONVERSION

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

To complete the decimal to binary conversion presented in this example, simply copy the binary digits in the righthand column, writing them down horizontally from right to left, with the top digit on the right. You can then see that the binary equivalent of the decimal (not hexadecimal) number 117 is 01110101. If you have a decimal to binary conversion chart, you can use it to confirm the accuracy of this calculation.

DOING IT THE EASY WAY

Even though it is not difficult to convert binary numbers to decimal and vice versa, doing it by hand is time-consuming—and when you program in Assembly language, you have to do a lot of binary to decimal and decimal to binary converting. So there are a lot of BASIC programs around for converting numbers back and forth between the binary and decimal notation systems. Here is a Commodore BASIC program for converting binary numbers to decimal numbers:

A BINARY-TO-DECIMAL CONVERSION PROGRAM

```
10 REM *** BINDEC.BAS ***
20 DIM BIT(8),BIT$(8)
30 PRINT CHR$(147):REM CLEAR SCREEN
40 PRINT:PRINT "        BINARY-DECIMAL CONVERSION"
50 PRINT:PRINT "ENTER AN 8-BIT BINARY
    NUMBER:":PRINT:INPUT A$
60 IF LEN(A$)<>8 THEN 50
70 FOR L=8 TO 1 STEP -1
80 BIT$(L)=MID$(A$,L,1)
90 IF BIT$(L)<>"0" AND BIT$(L)<>"1" THEN 50
100 NEXT L
```

```
110 FOR L=1 TO 8
120 BIT(L)=VAL(BIT$(L))
130 NEXT L
140 ANS=0
150 M=256
160 FOR L=1 TO 8
170 M=M/2:ANS=ANS+BIT(L)*M
180 NEXT L
190 PRINT "DECIMAL:";ANS
200 GOTO 50
```

This program converts decimal numbers to binary numbers:

```
A DECIMAL-TO-BINARY CONVERSION PROGRAM

10 REM *** DECBIN.BAS ***
20 DIM BT$(8):PRINT CHR$(147):REM CLEAR SCREEN
30 PRINT:PRINT "     DECIMAL-BINARY CONVERSION"
40 PRINT:PRINT "ENTER A POSITIVE INTEGER (1 TO 255):"
50 BN$="":PRINT:INPUT A$
60 IF VAL(A$)<1 OR VAL(A$)>255 THEN 40
70 NR=VAL(A$)
80 FOR L=8 TO 1 STEP -1
90 Q=NR/2
100 R=Q-INT(Q)
110 IF R=0 THEN BT$(L)="0":GOTO 130
120 BT$(L)="1"
130 NR=INT(Q)
140 NEXT L
150 PRINT "BINARY: ";
160 FOR L=1 TO 8:PRINT BT$(L);:NEXT L:PRINT
170 GOTO 40
```

BINARY/HEX CONVERSIONS

It is easy to convert binary numbers to their hexadecimal equivalents, using Table 2-7.

Table 2-7 HEXADECIMAL TO BINARY CONVERSION

HEXADECIMAL	BINARY
0	0000
1	0001
2	0010
3	0011
4	0100

Table 2-7 continued

HEXADECIMAL	BINARY
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

The above table shows how to convert a multiple-digit hex number to binary notation. Merely string the hex digits together and convert each one separately. 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, just use the table in reverse, being careful to group the binary number in groups of four digits, starting at the right. The binary number 1101 0110 1110 0101, for example, is equivalent to the hexadecimal number D6E5.

DECIMAL/HEXADECIMAL CONVERSION

It is almost as easy to convert decimal numbers to hexadecimal as to translate binary numbers to decimal. First, divide the decimal integer that you want to convert by 16. Then write down the remainder, like this:

64540/16 = 4033 with a remainder of 12

Then divide the new quotient by 16, and write down the result of that calculation:

4033/16 = 252 with a remainder of 1

Now keep repeating this process until you have a quotient of zero. Here is 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 have finished this series of calculations, you must convert any remainder that is greater than 9 into its hexadecimal equivalent. In the above example, 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 corresponds to the letter C in hexadecimal notation, and the decimal number 15 corresponds to the letter F. So the remainders in the above problem, converted into hex, are:

C
1
C
F

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

This conversion process will work with any decimal integer. But there is an easier way to to convert a decimal number to a hexadecimal number. Let your computer do it for you, with this BASIC program:

```
A DEC-HEX AND HEX-DEC CONVERSION PROGRAM

10 REM *** DECHEX.BAS ***
20 DIM HEX$(8)
30 PRINT CHR$(147):REM CLEAR SCREEN
40 PRINT:PRINT "WHAT TYPE OF CONVERSION DO YOU WANT?"
50 PRINT:PRINT "   (A) DECIMAL TO HEXADECIMAL"
60 PRINT "   (B) HEXADECIMAL TO DECIMAL"
70 PRINT:PRINT "TYPE 'A' OR 'B'":PRINT:INPUT A$
80 IF A$="B" THEN 270
90 IF A$<>"A" THEN 40
100 PRINT CHR$(147):PRINT:PRINT "THIS PROGRAM WILL
    TRANSLATE DECIMAL"
110 PRINT "NUMBERS FROM 0 TO 99999999":PRINT "INTO
    HEXADECIMAL NUMBERS"
120 PRINT:PRINT "TYPE DECIMAL NUMBER (OR 'HEX' FOR
    HEX)":INPUT A$
130 FOR L=1 TO 8:HEX$(L)="":NEXT L
140 IF A$="HEX" THEN 270
150 FOR L=1 TO 8:T$=RIGHT$(A$,L)
155 IF ASC(T$)<48 OR ASC(T$)>57 THEN 120
156 NEXT L
160 IF LEN(A$)<1 OR LEN(A$)>8 THEN 120
170 N=VAL(A$)
180 I=8
190 TMP=N:N=INT(N/16)
200 TMP=TMP-N*16
```

```
210 IF TMP<10 THEN HEX$(I)=RIGHT$(STR$(TMP),1):GOTO
    230
220 HEX$(I)=CHR$(TMP-10+ASC("A"))
230 IF N<>0 THEN I=I-1:GOTO 190
240 PRINT "HEX: ";
250 FOR L=1 TO 8:PRINT HEX$(L);:NEXT L:PRINT
260 GOTO 120
270 PRINT CHR$(147):PRINT:PRINT "THIS PROGRAM WILL
    CONVERT HEXADECIMAL"
280 PRINT "NUMBERS FROM 0 TO FFFFFFFF":PRINT "INTO
    DECIMAL NUMBERS"
290 PRINT:PRINT "TYPE HEX NUMBER (OR 'DCX' FOR
    DECIMAL):":INPUT A$
300 IF A$="DCX" THEN 100
310 IF LEN(A$)>8 THEN 290
320 N=0
330 FOR L=1 TO LEN(A$)
340 HEX$(L)=MID$(A$,L,1)
350 IF HEX$(L)<="9" THEN N=N*16+VAL(HEX$(L)):GOTO 390
360 IF HEX$(L)<"A" THEN 290
370 IF HEX$(L)>"F" THEN 290
380 N=N*16+ASC(HEX$(L))-ASC("A")+10
390 NEXT L
400 PRINT "DEC: ";N:PRINT
410 GOTO 290
420 END
```

USING 16-BIT NUMBERS IN PEEK AND POKE COMMANDS

Before we move on to the next chapter, there is one more topic to cover: how to use 16-bit numbers in PEEK and POKE commands.

As you may have heard, your Commodore belongs to a class of computers called 8-bit computers. The characteristics of 8-bit computers are covered in detail later on in this book; for now, we need remember only that the memory registers in an 8-bit computer are capable of holding numbers no more than 8 bits long; that is, numbers ranging from 0 through 255. Therefore, the only way to store a 16-bit number in an 8-bit computer is to break it down into two 8-bit numbers, storing those two numbers in two consecutive memory registers.

Now here is an important fact about the 6502 microprocessor and its successors, including the 6510/8502 chip used in your Commodore. These chips handle 16-bit numbers differently than you might expect: the low-order byte first and the high-order byte second. For example, if the hexadecimal number $FC1C were stored in hexadecimal Memory Addresses $8000 and $8001, the byte $1C would be stored in Memory Register $8000, and the byte $FC would be stored in Memory Register $8001.

STORING A 16-BIT NUMBER IN RAM

Now let's suppose that you actually do want to store a 16-bit number in Memory Registers $8000 and $8001, using a BASIC POKE command. Since BASIC programs are written using decimal numbers, the first thing you have to do is convert the addresses you wanted to use—in this case, $8000 and $8001—into decimal numbers. Making that conversion, with the hexadecimal-to-decimal conversion program in this chapter, you find that the decimal equivalents of $8000 and $8001 are 32768 and 32769.

Once you have calculated the decimal equivalents of the addresses you want to POKE values into, you can use a BASIC program to do the actual poking. Following is one such program:

```
STORING A 16-BIT NUMBER IN TWO 8-BIT MEMORY REGISTERS

10 AL=32768:AH=32769:REM LOW AND HIGH ADDRESSES
20 PRINT "TYPE A POSITIVE INTEGER"
30 PRINT "RANGING FROM 0 TO 65,535"
40 INPUT X
50 HIBYTE=INT(X/256)
60 LOBYTE=X-HIBYTE*256
70 POKE AL,LOBYTE
80 POKE AH,HIBYTE
90 END
```

RETRIEVING A 16-BIT NUMBER FROM RAM

Now suppose that you want to retrieve a 16-bit number from RAM using a PEEK command. You can do that with a BASIC program like the following:

```
RETRIEVING A 16-BIT NUMBER FROM 2 8-BIT MEMORY
REGISTERS

10 AL=32768:AH=32769
20 X=PEEK(AH)*256+PEEK(AL)
30 PRINT X
```

3 In the Chips

Inside Your Commodore

In this chapter, we are going to get "under the hood" of your Commodore and take a look at how it works. Then you will be able to find your way around inside your computer and be ready, at last, to start doing some Assembly language programming.

As you remember, every computer has three main parts: a *central processing unit* (CPU), *memory* (divided into *RAM* and *ROM*), and some *input* and *output (I/O)* devices (such as keyboards, video monitors, cassette recorders, and disk drives).

In a microcomputer, all of the functions of the CPU are contained in a microprocessor unit (sometimes abbreviated MPU). Your Commodore's MPU is a very-large-scale integrated circuit (VLSI) called a 6510 if you own a C64 or an 8502 if you have a C128. The 6510/8502 chip is almost identical to a more widely used chip called the 6502, and it is designed to be programmed in standard 6502 Assembly language.

The 6510/8502 microprocessor contains seven main parts: an *arithmetic logic unit* (ALU) and five *addressable registers.*

Your computer also contains a set of transmission lines called *buses.* They move data back and forth between the registers in the 6510/8502 chip and the memory registers.

There are two kinds of buses in your Commodore 64 or Commodore 128: an 8-bit *data bus* and a 16-bit *address bus.* The data bus, as its name implies, is used mainly for passing data back and forth between the 6510/8502 chip and the computer's memory registers. The address bus, as you might also guess, is used to keep track of the addresses of the various memory registers used in a program.

THE ARITHMETIC LOGIC UNIT

The most important component in your computer is its 6510/8502 chip. The most important part of the 6510/8502 chip is its ALU. When your computer performs a calculation or a logical operation, the ALU is the part that does the work.

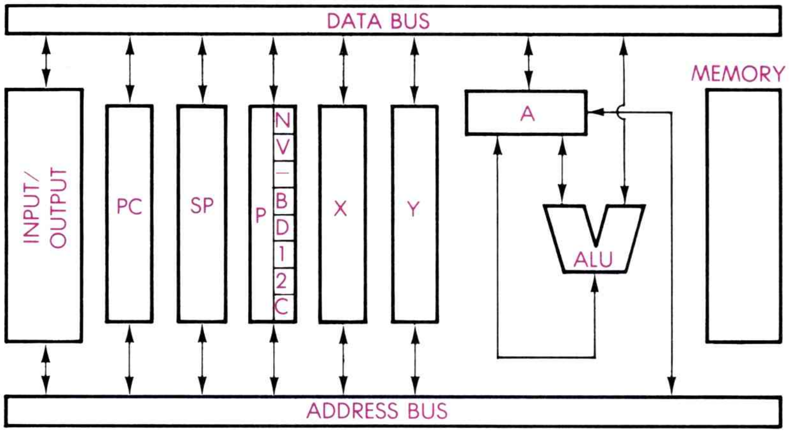

Figure 3-1 Block Diagram of the 6510 Microprocessor

The ALU can actually perform only two kinds of calculations: addition and subtraction. Division and multiplication are handled by the ALU in the form of sequences of addition and subtraction operations.

The ALU can also compare values, but as far as the 6510/8502 chip is concerned, that too is an arithmetical operation. When the 6510/8502 chip compares two values, what it actually does is subtract one value from the other. Then, by merely checking the results of this subtraction operation, it can determine whether the subtracted value is more than, less than, or equal to the value that it was subtracted from.

The 6510/8502 chip's ALU has two inputs and one output. When two numbers are to be added, subtracted, or compared, they are fed separately into the ALU through each of its inputs. The ALU then carries out the requested calculation and puts the answer on a data bus so that it can be transported to another register.

THE ALU HOPPER

In diagrams of the 6502 chip and of its numerous descendents, including the 6510/8502, the ALU is often represented as a V-shaped hopper. The ALU has two inputs, which are traditionally illustrated as the two arms of the the hopper, and one output, traditionally represented as the bottom of the V.

When two numbers are to be added, subtracted or compared, the following happens. First, a number is stored in the 6510/8502'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 its other input. When all that is done, the ALU carries out the requested calculation, and the result of the calculation finally 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.

Here is a short Assembly language program that shows how this process works:

```
EXAMPLE OF AN ALU OPERATION

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

The first instruction in this listing, LDA, means "load the accumulator." In this case, the accumulator is loaded with the number 2. The # in front of the number 2 means that the 2 in the instruction is to be interpreted as a literal number. If there was no #, the 2 would be interpreted as the address of a memory register.

The second instruction in the listing, ADC, means "add with carry." In 6502 arithmetic, as in ordinary pencil-and-paper arithmetic, the addition of two numbers often results in a carry. If there was a carry, the ADC instruction would be able to handle it, and in a later chapter you will find out how. However, in this particular addition problem, there is no number to be carried, so all the ADC instruction does is add 2 and 3.

As soon as ADC #3 appears in this program, the 2 that has been loaded into the accumulator is deposited into one of the ALU's inputs. The instruction ADC #3 is placed in the ALU's other input. The ALU then carries out this instruction: it adds 2 and 3, placing their sum back in the accumulator.

Now we are ready for the third and last instruction in this little program. The numbers 2 and 3 have been added, and their sum is now in the accumulator. 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 $FB—the hexadecimal equivalent of the decimal number 251. So it appears that the number 5 is now going to be stored in Memory Register $FB.

Now take a close look at the hexadecimal number $FB in Line 3. Since there is no # in front of the number $FB, your assembler will not interpret it as a literal number. Instead, $FB will be interpreted as a memory address, which is what a number has to be in Assembly language if it is not designated as a literal number and carries no other identifying labels.

Incidentally, if you did want your assembler to interpret $FB as a literal number, you would have to write it #$FB. When # and $ both appear before a number, it is interpreted as a literal hexadecimal number.

If the third line of the program were STA #$FB, however, that would be a syntax error. That is because the instruction STA (store the contents of the accumulator in . . .) has to be followed by a value that can be interpreted as a memory address—not by a literal number.

THE 6510/8502'S OTHER REGISTERS

Besides its accumulator, the 6510/8502 processor has five other internal registers. They are the *X Register,* the *Y Register,* the *Program Counter,* the *Stack*

Pointer, and the *Processor Status Register.* Brief descriptions of the functions of each of these five registers follow.

- The X Register (abbreviated *X*) is an 8-bit register that is often used for temporary storage of data during a program. The X register has a special feature, too; it can be incremented and decremented with a pair of special Assembly language instructions (INX and DEX), and it is therefore often used as an index register, or counter, during loops and read/data-type instructions in programs.
- The Y Register (abbreviated *Y*) is also an 8-bit register, and it can also be incremented and decremented with a pair of special instructions (INY and DEY). The Y register, like the X register, is used both for data storage and as a counter.
- The Program Counter (abbreviated *PC*) is actually a pair of 8-bit registers that are usually used together as one 16-bit register. The two 8-bit registers that make up the program counter are occasionally referred to as the *Program Counter—Low (PCL)* register and the *Program Counter—High (PCH)* register. The program counter always contains the 16-bit memory address of the next instruction to be executed by the 6510/8502 processor. When that instruction has been carried out, the address of the next instruction is loaded into the program counter.
- The Stack Pointer (which can be abbreviated either *S* or *SP*) is an 8-bit register that always contains the address of the next available memory address in a block of RAM called the *hardware stack,* or simply stack—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 6510/8502 chip uses the stack as a temporary storage location for return addresses. You can use the stack for other purposes in Assembly language programs, too.
- The Processor Status Register (often called simply the *Status Register,* but abbreviated *P*), is an 8-bit register that keeps track of the results of operations performed by the 6510/8502. It is a very important register, so we will take a closer look at it before we move on.

THE PROCESSOR STATUS REGISTER

The processor status register is built differently from the other registers in the 6510/8502, and it is used differently, too. It is not designed for storing or processing ordinary 8-bit numbers, as the 6510/8502's other registers are. Instead, its bits are used as flags that keep track of several kinds of important information.

Four of the status register's eight bits are called status flags. These four flags and their abbreviations are:

1. The *carry flag* (C)

2. The *overflow flag* (V)
3. The *negative flag* (N)
4. The *zero flag* (Z)

These four flags are used to keep track of the results of operations carried out by the other registers inside the 6510/8502 processor.

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

1. The *interrupt disable flag* (I)
2. The *break flag* (B)
3. 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 with 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 said to be set. If it has the binary value 0, then it is said to be clear.

The bits in the 6510/8502 status register—like the bits in all 8-bit registers—are customarily numbered from 0 to 7. By convention, the rightmost bit in an 8-bit register is referred to as Bit 0, and the leftmost bit is referred to as Bit 7. The positions of each bit in the 6510/8502 status register are:

BITS	7	6	5	4	3	2	1	0
FLAGS	N	V	-	B	D	I	Z	C

BIT 0: THE CARRY FLAG (C) It is not easy to do 16-bit arithmetic with an 8-bit chip like the 6510/8502. When the 6510/8502 chip is required to perform an addition operation on a number greater than 255 or if the result of a calculation might be greater than 255, then a program is needed to break each number down into 8-bit segments for processing and then patch all the numbers back together again.

This kind of mathematical cutting and pasting, as you can imagine, involves a lot of carrying (if addition problems are being performed) and borrowing (when the 6510/8502 is performing subtraction). The carry flag of the 6510/8502 P register is the flag that keeps up with all this carrying and borrowing.

If an addition operation results in a carry, the carry flag is automatically set. If a subtraction operation requires a borrow, the carry flag is automatically cleared. The carry flag is also set and cleared by many other kinds of 6510/8502 operations. It is therefore good programming practice to *clear* the carry flag any time addition is to be performed and to *set* it before subtraction. If you do not clear the carry flag before every addition and set it before every subtraction, your calculations may be thrown off by the left-over results of previous calculations.

The Assembly language instruction to clear the P register's carry bit is CLC, which stands for "clear carry," and the instruction to set the carry bit is SEC, which stands for "set carry."

BIT 1: THE ZERO FLAG (Z) When the result of an arithmetic or logical operation is zero, the status register's zero flag is automatically set. Addition, subtraction, and logical operations can all result in changes in the status of the zero flag. If a memory location or an index register is decremented to zero, that also causes the zero flag to be set.

When you write routines that make use of the zero flag, it is important to remember one 6502 convention that seems odd until you get used to it. 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). This convention is easy to forget—and can trip you up if you are not careful.

There are no Assembly language instructions to clear or set the zero flag. It is strictly a "read" bit, so instructions to write to it are not provided.

BIT 2: THE INTERRUPT DISABLE FLAG (I) Many Commodore programs contain *interrupts,* instructions that halt operations temporarily so that other operations can take place. Some interrupts are called *maskable interrupts,* because you can be prevent them from taking place by including special "masking" instructions in a program (there will be more about that in a later chapter). Other interrupts are called *nonmaskable,* because you cannot stop them from taking place, no matter what you do. Nonmaskable interrupts usually perform functions that are vital to the operation of a computer, such as writing to the screen every $\frac{1}{60}$ second and performing other time-critical chores.

To disable a *maskable* interrupt, clear the P register's interrupt disable flag. When this flag is set, maskable interrupts are not permitted. When it is clear, they are.

The Assembly language instruction to clear the interrupt flag is CLI. The instruction to set the interrupt flag is SEI.

BIT 3: THE DECIMAL MODE FLAG (D) The 6510/8502 processor normally operates in what is called binary mode, using standard binary numbers. But the chip can also operate in what is known as binary-coded decimal, or BCD, mode. To put your computer into BCD mode, set the decimal flag of the 6510/8502 status register.

When the 6510/8502 chip is put in BCD mode, it uses only the 10 standard decimal digits; 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 requires three bytes to express decimal 255 as a BCD number. In your computer's memory, the BCD number 255 is stored this way:

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

That is quite different, of course, from the way that decimal number 255 is expressed in conventional binary (non-BCD) notation. In binary arithmetic, the kind you will use most often in your Assembly language programs, the decimal number 255 is expressed as a hexadecimal number, like this:

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 conventional binary numbers. (It is possible to "pack" BCD digits into half that amount of space, but that requires special procedures and additional processing time, as you will see in a later chapter.) Another disadvantage of BCD arithmetic is that it is slower than binary arithmetic. But its results, unlike those of plain binary arithmetic, are always 100% accurate. So it is often used in programs and routines in which accuracy is more important than speed or memory efficiency.

Another advantage of BCD numbers is that they are easier to convert into decimal numbers than standard binary numbers. So BCD numbers are sometimes used in programs that call for the instant display of numbers on a video monitor.

BCD numbers will be discussed in more detail in a later chapter. For now, it is sufficient to remember that when the status register's decimal mode flag is set, the 6510/8502 chip performs all its arithmetic using BCD numbers. You probably will not be using much BCD arithmetic in your Assembly language programs—at least not for a while—so you will usually want to 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 set by the Assembly language instruction BRK. Program designers often use the break instruction while they are debugging Assembly language programs. When the 6510/8502 break flag is set and the BRK instruction is invoked, certain error-flagging operations take place, and control of the computer returns to the programmer. Once the debugging of a program is complete, any BRK instructions placed in the program for use during debugging are usually removed.

BIT 5: UNUSED BIT For some reason, the programmers who designed the 6510/8502 status register left one bit unused. This is the one.

BIT 6: THE OVERFLOW FLAG (V) The overflow flag is used to detect an overflow from Bit 6 to Bit 7 in a binary number. The overflow flag is used primarily in addition and subtraction problems involving signed numbers. When the 6510/8502 microprocessor performs calculations on signed numbers, each number is expressed as a 7-bit value, with its leftmost bit used to designate its sign. When Bit 7 is used in this way, an overflow from Bit 6 to Bit 7 can make the result of a calculation incorrect. So, after a calculation involving signed numbers has been performed, the V flag is tested to see whether such an overflow has occurred. Then, if so, corrective action can be taken. (More information on how the V flag is used in signed-number operations work is provided later on in this book, in a chapter devoted to 6510/8502 arithmetic.)

The Assembly language instruction that clears the overflow flag is CLV. The V flag is a read-only bit, so there is no specific instruction to set it.

BIT 7: THE NEGATIVE FLAG The negative flag is set when the result of an operation is negative and cleared when the result is zero. The negative flag is often used in operations involving signed numbers and to detect whether a counter being used in a loop has been decremented past zero. It has other uses that will be discussed in later chapters. There are no instructions to set or clear the negative flag; it is strictly read-only.

4 Writing an Assembly Language Program

Three Popular Commodore Assemblers

This chapter is the payoff to everything you have learned so far in this book. Very soon, you are going to write your first Assembly language program, a program you can type, assemble, and execute without using any BASIC commands.

The program was created using the Merlin 64 assembler, but you can also type, assemble and run it using two other assemblers: the Commodore 64 Macro Assembler Development System and the Panther C64.

This chapter is divided into three parts. Part 1 tells how to type and assemble a program on the Merlin assembler. Part 2 explains how to enter the program using the Commodore 64 assembler. Part 3 is about typing and assembling the program using the Panther C64 assembler.

No matter which assembler you are using, I suggest that you read Part 1, since it is the only section that contains a full line-by-line explanation of how the program works.

THE MERLIN 64 ASSEMBLER

As you may have guessed by now, the word *assembler* can have a couple of different meanings, depending upon the context. When programmers speak of an "assembler," they are sometimes talking about just one part of a software-development package, the part that does the actual work of converting Assembly language into machine language. But the word can also be used to refer to a complete Assembly language programming package, such as the Commodore 64 Macro Assembler Development System or the Merlin 64 assembler/editor program. Software packages such as these usually include more than just an assembler. Other types of programs that are often contained in "assembler" software packages include editors, monitors, loaders, and debugging utilities.

The Merlin 64 assembler/editor system comes on a single diskette, which contains a number of programs. These programs are divided into five modules, which are:

- an Executive module
- an Editor module
- an Assembler module
- a Monitor module
- a Symbol Table Generator module

When you use the Merlin 64 assembler, the modules that you encounter most often are the Executive, Editor, Assembler, and Monitor modules. You seldom have to worry about the Symbol Generator module; its job is to compile tables of constants and variables, and it does its work automatically and quite transparently, usually without any assistance from the programmer.

The Merlin 64 assembler/editor can be loaded into the Commodore's memory just like any other disk-based program. Just turn on the disk drive, the monitor, and the computer, and slip the Merlin 64 disk into the disk drive. Then type:

```
LOAD "MERLIN",8
```

followed by a carriage return. Your computer should respond with the message:

```
LOADING
```

and then the prompt:

```
READY.
```

You can then type the command:

```
RUN
```

The next thing you see is a preliminary title screen. Then, after a few moments of disk-spinning, Merlin's master menu appears on your video display.

MERLIN'S MENU

Merlin is a menu-driven assembler: when you boot the Merlin 64 disk, the first thing you see, after the title screen, is the master menu used to select the program's function. When you load Merlin and see this master menu, you know the assembler is in its Executive mode; the Merlin 64 module that controls this mode is called the Executive module.

When Merlin is in its Executive mode, the menu on the screen looks like this:

```
C :Catalog
L :Load source
S :Save source
```

```
R :Read text file
W :Write text file
D :Drive change
E :Enter ED/ASM
O :Save object code
G :Run program
X :Disk command
Q :Quit
```

All these options are explained in detail in the Merlin 64 assembler instruction book, so we will not present a long explanation of each menu option in this chapter. For now, it is sufficient to note that Merlin can do quite a few things in Executive mode, from loading and saving source code and object code to listing the contents of a disk (using the menu's C command). You can read and write text files with 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.

THE PROGRAM

To put Merlin into its Editor mode, the mode that you will be using to write Assembly language programs, select choice E from the assembler's Executive menu. At the bottom of your monitor screen, just below the menu, you will see a % followed by a flashing cursor. You will always see the % prompt when Merlin is in its Executive mode. When the assembler is in Editor mode, the prompt changes to a colon, and in monitor mode, the prompt is a dollar sign.

When you have located the % prompt, type the letter:

```
E
```

for "Enter Editor/Assembler mode." As soon as you have typed an E and pressed RETURN, Merlin's Editor module goes into action. To let you know that it is in its edit mode, Merlin will print a : prompt on your screen. When you see that prompt, type:

```
A
```

for "Append." You will then 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 1 and progressing in increments of 1. So when the number 1 appears on your screen, it means that Merlin is ready to accept the first typed line of a source-code program. Let's start typing an Assembly language program right now.

If your assembler is working properly, you will notice that the 1 on your screen is followed by a space and a flashing cursor. Now, beginning at the spot where the cursor is, type an asterisk, *without* any additional spaces in front of it, and press RETURN. Line 1 of your program should now look like this:

```
1 *
```

Merlin then advances automatically to Line 2. Following the number 2, again without any extra spacing, type:

```
*ADDNRS
```

and press your RETURN key. Then, when Merlin advances to Line 3, type another asterisk.

This is what you should see on your screen now:

```
1 *
2 * ADDNRS
3 *
4
```

At Line 4, just press RETURN, and you will see Merlin's : prompt again. Then you can type A (for APPEND) again and continue writing your program, or, if you prefer, you can type some other command.

One command that can be used after the : prompt is L for List. If you type an L command, your program will be listed, in its entirety, on your computer screen.

Another command that can follow the : prompt is D for Delete. To use the Delete command, type the letter D followed by the number of the line (or lines, separated by a comma) you want to delete. Suppose, for example, that you wanted to delete Lines 2 and 3 in the above listing. You could do that by simply typing:

```
D2,3
```

after the : prompt. Try it. Then restore the lines you have deleted by using the A command.

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. Suppose, for example, that you wanted to insert another asterisk at Line 2 in the above program. You could simply press RETURN to get a colon prompt, and then type:

```
I2
```

Try that, and you will see Merlin respond with the number:

```
2
```

Now type an asterisk, followed by two RETURNs. Merlin will display its : prompt again, and you can then type L for list. Then Merlin will list your program, and you see that another line containing an asterisk has indeed been inserted into your program, at Line 2.

You may also notice that Merlin has automatically renumbered each line after the line that you have inserted. That is a convenient characteristic of the Merlin editor, but it requires some getting used to. With Merlin, you never have to worry about renumbering lines, so that you can make insertions or clean up the line numbers in a program. But you also have to remember that the line numbers in your program can change, whether you like it or not, as lines are inserted and deleted.

Speaking of deleting lines, you can now delete that extra asterisk that you have just added to your program. Just press RETURN to get the : prompt, and type D2. Then you can type L for List, which will get you a listing showing you that the program looks like this again:

```
1 *
2 * ADDNRS
3 *
4
```

In addition to the A, I, and D commands, there is also an R command, to replace a line. Merlin also has commands that you can use to copy lines, to move lines, to find and replace strings, and to perform many other useful functions. You can find full details on how to use all these functions by reading the Merlin 64 instruction manual.

However, you do not have to do that now. For the moment, just continue typing until you have entered the following program into your computer:

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

You have probably noticed that Merlin tabulates columns automatically, dividing a program into easy-to-read fields. Merlin automatically generates a space after each line number, so you do not have to type one. If you do type a space, you wind up in the third column, where all the three-letter abbreviations appear. Later on in this chapter, I will discuss the spacing of program listings at greater length, and you can find still more on the subject in the instruction manual that comes with the Merlin 64 assembler.

LISTING YOUR PROGRAM

When you have reached Line 12 in your ADDNRS program, just press your RETURN key. Then you can type either L or List, to list the complete program on your computer screen.

Sharp-eyed readers may have noticed that this is the same program that was presented in Chapter 1, a program that adds 2 and 2 and stores their sum in Memory Address $02A7 (that is, decimal 679). As you can see by looking at the program, that happens in Line 9.

Some of the instructions in this program may look familiar by now; we have touched on most of them in the preceding chapters. However, the program also includes one or two items that you have not encountered until now. These new features include the asterisks in the first section of the program and the END directive in Line 11.

Here is an "exploded" listing of the program to give you a clearer understanding of how it is written. The listing you saw in the first example is divided into five fields, or columns, each with a heading that describes the kind of information it contains. As you examine this listing, please understand that no one actually writes programs in this way. It is presented only to give you a clear picture of the organization of an Assembly language program.

```
AN 8-BIT ADDITION PROGRAM
(Listing No. 2)

LINE              OP
NO.    LABEL      CODE  OPERAND   REMARKS

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

As you can see from this listing, Assembly language programs using the Merlin 64 system can be divided into five fields: the line-number field, the label field, the op-code field, the operand field, and the remarks field.

LINE NUMBERS

Not every assembler uses *line numbers* exactly as Merlin does. In fact, some assemblers do not use line numbers at all. In Assembly language programs, line numbers are *never* used as reference points; routines and subroutines are referred to by their labels, not by their line numbers. When line numbers appear in an Assembly language program, they are used only as a convenience to the programmer, not because the program really requires them. So keeping line numbers all neat and tidy is not nearly as important in an Assembly language program as it is in a BASIC program.

LABELS

Labels always occupy the second field in Assembly language statements written using the Merlin 64 assembler. Labels are very important in 6502/6510/8502 Assembly language, since they are used instead of line numbers to address routines and subroutines. In the ADDNRS program, the abbreviation ADDNRS in Line 2 is a label and thus appears in the second field of the source-code listing.

Because the ADDNRS program is identified with a label, it could be used as either a secondary routine or a subroutine in a longer program. If it were used as a subroutine, it could be accessed using the instruction JSR ADDNRS, which is equivalent to GOSUB in BASIC, or the instruction JMP ADDNRS, which works like BASIC's GOTO instruction.

If ADDNRS 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 were used as a *secondary routine,* the RTS instruction would end both the secondary routine and the main program. We will discuss the JSR and JMP instructions at greater length in later chapters.

A label can be as short as one character and as long as the length of a statement permits. Most programmers use labels three to six characters long.

OP-CODE MNEMONICS

An *operation-code* (or *op-code*) mnemonic is just an Assembly language instruction. The 56 op-code mnemonics in the 6502/6510/8502 instruction set are the only ones that can be used in Commodore 64 Assembly language instructions.

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 Commodore 64 assembler, 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 it 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,* which are words and symbols that are entered into a program like mnemonics but are not actually members of the 6510/8502 instruction set. ORG in Line 4 of the ADDNRS program is a directive, and END in line 11 is a pseudo-op. The ORG directive tells your computer where an Assembly language program is to be stored in memory after it is assembled. END tells the assembler where to stop assembling, to end an Assembly language program.

OPERANDS

The *operand* field in a Merlin 64 assembler program starts one space (or a tab) after an op-code mnemonic. Operands and op-code mnemonics make up complete instructions. Some mnemonics, such as CLC, CLC and RTS, do not require operands. Others, such as LDA, STA and ADC, do. Much more information about operands is provided later on.

COMMENTS

Comments in Assembly language programs are like remarks in BASIC programs: although they do not affect a program in any way, they explain programming procedures and provide eye-saving space in program listings.

There are two ways to include a comment in source-code listings written on the Merlin 64 assembler. One is to put it in the label field of a listing, preceded by an asterisk. The other is to precede the comment with a semicolon and put it in the remarks field that follows the operand field.

EXAMINING THE PROGRAM

Now that we have looked at our sample program field by field, let's examine it line by line.

LINES 1 THROUGH 3

(Comments) Lines 1 through 3 are comments. Line 2 explains what the program does, and lines 1 and 3 set off the explanatory line by printing asterisks followed by white space.

It is good programming practice—in Assembly language, just as in most other programming languages—to use remarks liberally. Comments have been used quite liberally in the programs in this book.

LINE 4
ORG $8000

This is the *origin* line of our sample program. Every Assembly language program must start with an origin line. As you may remember from Chapter 1, when a computer runs a machine-language program, it first goes to a predetermined memory location and checks the value stored at that address. So when you write an Assembly language program, you have to tell your computer where to start looking for the program in its memory.

The origin directive in an Assembly language program sets a counter that your assembler uses to keep track of the bytes in the program. This counter is called, logically enough, the program counter. Your assembler's program counter, like the program counter in your computer's 6510/8502 processor, always contains the address of the next instruction to be used in a machine-language program.

When your assembler encounters an origin directive, it sets its program counter to the address that follows the abbreviation ORG. The first instruction in the program is then loaded into that address, and the rest of the program follows.

The origin directive looks like a simple line to write, but deciding what to put in it can be quite difficult, especially for the beginning programmer. There are many blocks of memory in your computer that you cannot use for Assembly language programs because they are reserved for other uses (to hold your computer's operating system, disk operating system, and BASIC interpreter, for example). Even the programs in your Merlin 64 assembler/editor package take up blocks of memory that the Assembly language programmer must avoid using.

Deciding where to store a program in a computer's memory is such a tricky job, with so many variables to be taken into account, that a full chapter on memory management is presented later on in this book. For now, it is sufficient to know that it is usually safe to start Assembly language programs somewhere around memory location $8000 and that there is also an 88-byte block of free RAM from Location $02A7 (decimal 679) to Location $2FF (decimal 767), which makes a handy storage spot for short lists of data.

Now you know why Line 4 in the ADDNRS program tells sets your assembler's program counter at $8000 and why Line 9 stores the sum of 2 and 2 in Memory Address $02A7.

LINE 5
ADDNRS CLD

(Label: ADDNRS)
(Mnemonic: "Clear Decimal Mode") *ADDNRS:* The label field in this line has been used to name this routine ADDNRS, so, if we ever decide to use the routine as part of a larger program, it will have a name. We can access it by its label, if we wish, instead of by its memory location. It is usually good programming practice to give labels to important routines and subroutines. A label not only makes a routine easier to locate and use, it also serves as a reminder of what the routine does (or, until your program is debugged, what it is supposed to do).

CLD: We are using plain binary numbers in this program, not binary-coded decimal numbers. So in this line we will clear the decimal mode flag of the 6510/8502 processor status register. It is not necessary to clear the decimal flag before every arithmetical operation in a program, but it is a good idea to clear it before the first addition or subtraction operation in a program, since it just may have been set during a previous program.

LINE 6
CLC

("Clear Carry") The status register's carry flag is affected by so many kinds of operations that it is considered good programming practice to clear it before every addition operation and set it before every subtraction operation. It takes very little time and just one byte of RAM; compared to the time and energy debugging can cost, that is a bargain.

LINE 7
LDA #2

("Load Accumulator with the Number 2") This is a very straightforward instruction. The first step in an addition operation is always to load the accumulator with one of the numbers that is to be added. The # in front of the number 2 means that it is a literal number, not an address. If the instruction were LDA 2, then the accumulator would be loaded with the contents of Memory Address 0002, not the number 2.

LINE 8
ADC #2

("Add the Number 2 to Contents of Accumulator, with Carry") This is also a straightforward instruction ADC #2 means that the literal number 2 is to be added to the number that is in the accumulator, in this case, another 2. As we have mentioned, there is no 6510 Assembly language instruction that means "add without carry," so the only way an addition operation can be performed without a carry is to clear the status register's carry flag and then perform an "add with carry" operation.

LINE 9
STA $02A7

("Store Contents of Accumulator in Memory Address $02A7") This line stores the contents of the accumulator in Memory Address $02A7. Note that the symbol # is not used before the operand ($02A7) in this instruction, since in this case the operand is a memory address, not a literal number.

LINE 10
RTS

("Return from Subroutine") 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. However, if the RTS is used at the end of the main body of a program, as it is here, it has a different function: instead of passing control of the program to a different line, it terminates the whole program, returning control of the computer to the input device that was in control before the program began, usually a cartridge, disk operating system (DOS), keyboard-screen editor, or machine-language monitor.

LINE 11
END

Just as the ORG directive begins an Assembly language program, the END pseudo-op ends it. END tells the assembler to stop assembling, *even if there is more source code after the word END*. The END directive can therefore be a powerful debugging tool. You can put it wherever you want in a program you are debugging, to make the assembler stop assembling at that point (until you remove it). When you have finished debugging your program, you can use the END directive to bring it neatly to an end. Before you can do that, of course, you must remove any leftover END directives that may still be hanging around—that is, if you want your final program assembled completely. When debugging is complete and your program is finished, it should contain only one END directive, where it belongs, at the very end.

PRINTING YOUR PROGRAM

When you have finished typing your source-code listing using the Merlin editor, you can print it out by typing the command:

```
PRTR 4
```

(This example assumes that your printer is installed as Device No. 4. If not, use the appropriate device number.) After you type the command PRTR, followed by the device number of the printer, just type LIST to print out your program.

ASSEMBLING AND SAVING YOUR PROGRAM

To assemble the ADDNRS program using the Merlin assembler, type the command ASM following the : prompt. Merlin then asks you if you want to update your source-code file, with the current date, for example. If you do not want to update your file, type N (for "No"), and Merlin will assemble your source-code program very rapidly.

In just a moment, we will save the ADDNRS program on a disk. First, though, let's take time out to compare the object code, which your assembler has generated with the source code (from which the object code was derived). In the table below, in the column labeled SOURCE CODE, you can see your source-code listing. In the next column, you can see the machine-language version of the program. To the right of that is the meaning of each Assembly language/machine-language instruction.

SOURCE CODE	MACHINE CODE	MEANING
CLD	D8	Clear status register's decimal-mode flag
CLC	70	Clear status register's carry flag
LDA #2	A9 02	Load accumulator with the number 2
ADC #2	69 02	Add 2, with carry
STA $02A7	8D A7 02	Store result in Memory Address $02A7
RTS	60	Return from subroutine

Now we will save both your source-code listing and your object-code listing on a disk. First, type Q (for Quit) after the : prompt to get your assembler back into its Executive (menu) mode. When you have done that, Merlin's main menu reappears. Then save your source code by selecting menu choice S, and save your object code by picking menu choice O.

The Merlin menu also offers a choice W, for "Write text file." When you pick that menu selection, your assembler saves your object-code listing on a disk, but as an ASCII-style text file, not as an executable binary file. You cannot run a text file on your computer, as you can a binary file, but ASCII listings of machine-language programs do have their uses. For example, you can print a text file out on paper or transmit it from computer to computer over a telephone line. Then the recipient can convert the program into a binary file and run it.

When you type an S, O, or W to save a source-code or object-code listing, Merlin asks you what you would like to name your program. When you name your program, you do not have to add any special kind of suffix to indicate whether it is a source-code listing or an object-code program, because Merlin does that automatically. If you are saving a source-code listing, the assembler automatically adds an S suffix to your file name. If you are saving an object-code listing, Merlin appends an O suffix.

THE COMMODORE AND PANTHER ASSEMBLERS

The next section in this chapter is for owners of the Commodore 64 Macro Assembler Development System. If you do not own a Commodore Macro assembler and do not care how it works, you can skip to the final section of this chapter, which is about the Panther C64 assembler. If you do not care about the Panther either, you can move on to Chapter 5.

THE COMMODORE 64 ASSEMBLER

The Commodore 64 Macro Assembler Development System, like the Merlin 64, comes on a single disk but includes a number of individual programs. They are:

- an Assembly language *editor,* called EDITOR64, which is a program used to write Assembly language programs.
- an *assembler,* which is used to convert source code to object code. The assembler in the Commodore Development system is called ASSEMBLER64.
- a *loader.* When an Assembly language program is converted into object code using the Commodore 64 assembler, the listing produced is not true machine language, but ASCII code that must be converted into actual machine code. The Commodore 64 loader produces this code from the pseudo-object code generated by the Commodore assembler.

  Actually, there are two loader programs in the Commodore 64 assembler package. These programs, called LOLOADER64 and HILOADER64, perform identical functions but are placed in different parts of RAM when you load them into your computer's memory. That way, if one of the loaders would overwrite machine code that you have written, you can use the other loader. The LOLOADER program goes into memory beginning at Address $0800, and HILOADER starts at RAM location $C800.

- a machine-language *monitor*. When the Commodore 64 loader generates a machine-language program, it is often necessary to debug the program before it is ready to be run. Then, after the program is debugged, it has to be saved on disk if its creator has any intention of ever running it again. The machine-language monitor program included in the Commodore 64 assembler/editor package can be used both to debug programs and to save them to disk, so that they can be retrieved and run whenever needed. The monitor also has a number of other uses, which will be described in the next chapter.

There are also two monitors in the Commodore 64 assembler package. As with loaders, the only difference between them is where they reside in your computer's memory when they are loaded. One goes into RAM beginning at Memory Address $8000, and the other starts at $C000.

If you do not quite understand what this memory-address business is all about, do not worry about it right now. For now, it is enough to know that since the ADDNRS.COM starts at Memory Address $8000, the monitor that should be used to debug it is MONITOR$C000, which starts at $C000, not MONITOR$8000, which starts at $8000.

In addition to all the programs mentioned above, the Commodore 64 assembler/editor disk also contains two other useful programs called DOS WEDGE 64 and BOOT ALL. Both those programs are discussed later on in this chapter.

THE PROGRAM

First, though, here is a listing of ADDNRS—the same program presented in the first section of this chapter—as it would appear if it were written using the Commodore 64 assembler. Very shortly, you will get an opportunity to type the program, assemble it, and run it, but first let's take a quick overall look at it and see what it does, and how it does what it is supposed to do.

```
AN 8-BIT ADDITION PROGRAM

10 ;
20 ;ADDNRS.COM
30 ;
40  *=$8000
50 ;
60 ADDNRS CLD
70  CLC
80  LDA #2
90  ADC #2
```

```
0100  STA $02A7
0110  RTS
0120  .END
```

As you remember, this program adds two numbers. In this version of the program, the numbers that it is supposed to add (2 and 2) are in Lines 80 and 90.

After the program adds 2 and 2, it stores their sum in Memory Address $02A7 (decimal 679), just as it did in its Merlin 64 version. In this version of the program, this happens in Line 100.

Here is an "exploded" listing to give you a clearer understanding of how it is written. In this example, the listing is divided into five fields. Each field has a heading that describes the kind of information it contains.

As you examine this listing, remember once again that no one actually writes programs in this way. It is presented only to give you a clear picture of the organization of an Assembly language program.

```
AN 8-BIT ADDITION PROGRAM
(Listing No. 2)

LINE            OP
NO.    LABEL    CODE   OPERAND      COMMENTS

 10    ;
 20    ;ADDNRS.COM
 30    ;
 40              *=$8000
 50    ;
 60    ADDNRS   CLD
 70             CLC
 80             LDA    #2
 90             ADC    #2
100             STA    $02A7
110             RTS
120             .END
```

LINE NUMBERS

The Commodore 64 assembler, like most assemblers, uses *line numbers,* but they work more like BASIC line numbers than Merlin numbers do, and the Commodore assembler, unlike Merlin, does not assign them automatically unless you tell it to. When you use the Commodore assembler, you can either write your own line numbers or instruct your assembler to assign them automatically by using a special AUTO command. Instructions for using the AUTO command are given on page 21 of the instruction manual.

As you can see, the line numbers in our sample ADDNRS.COM program progress from 10 to 120 in increments of 10, just like the numbers in a typical BASIC program. They do not have to be written that way, but they usually are. When a program is written using the Commodore 64 assembler, line numbers are typed flush left, just as they generally are in a BASIC program. The line numbers in a Commodore 64 assembler listing occupy the *first field* in the program's source-code listing.

LABELS

Labels always occupy the second field in Commodore 64 assembler programs, just as in programs written using the Merlin 64 assembler. Exactly one space—not two—must be left between a line number and any label that follows it. If you start a label two or more spaces after a line number or if you use your tab key to get to the label field, you may clobber your program.

The length of a label can range from one character to the maximum length of the label field in a tabulated listing. Most programmers use labels three to six characters long.

OP-CODE MNEMONICS AND OPERANDS

Op-code mnemonics and *operands* occupy the third and fourth fields of programs written using the Commodore 64 assembler, just as they do in Merlin 64 source-code listings, but some of the op-code directives used by the Commodore 64 assembler are different from those used in Merlin 64 programs. For example, look at line 40 in the ADDNRS.COM program. Instead of using the abbreviation ORG to identify the origin line of a program, the Commodore assembler uses an asterisk followed by an equal sign. The standard format for the use of these symbols is shown in Line 40 of the ADDNRS.COM program:

```
*=$8000
```

The third and fourth fields of the ADDNRS.COM program, just like the third fields in the ADDNRS.MER presented in section on Merlin, are used for op-code mnemonics and operands. Some of the pseudo-ops used in source-code listings written using the Commodore assembler are slightly different from those used in Merlin listings. One such pseudo-op is the .END directive in Line 120 of the ADDNRS.COM program; it has a period in front of it, which the Merlin END directive lacks.

COMMENTS

There are also differences in the way comments are written using the Commodore and Merlin assemblers. In programs written with the Commodore assembler, comments that begin in Field 2 are preceded by semicolons rather than by asterisks. Comments preceded by semicolons can also appear in what is left of each line following the instruction fields (the op-code and operand fields). If you use the comment field at the end of a line and do not have room there for the comment you want to write, you can continue your comments on the next line by simply typing a space, a semicolon, and the rest of your remarks.

A line-by-line explanation of the ADDNRS program was provided earlier in this chapter, in the section dealing with the Merlin 64 assembler. If you skipped that section because you are using the Commodore 64 assembler, back up and read the line-by-line program analysis right now, since the same explanations apply to the version of the program that was written on the Commodore 64 assembler. Once you have done that, you will be ready to write and run your first Assembly language program.

LOADING THE EDITOR 64 PROGRAM

Sit down at your computer, turn on your disk drive and your video monitor, and slip the Commodore 64 Macro Assembler Development System diskette into your disk drive. When that is done, you can load the Assembly language editor that is on the disk into your computer, or, if you prefer, you can start your editing session by loading a *wedge* program that is also on the disk into your computer's memory. A wedge is a machine-language program that makes life with a Commodore disk drive much simpler: the commands that are used to perform DOS functions are greatly simplified. For example, when the red light on your disk drive starts blinking because of some saving or loading problem, the source of the trouble is easy to track down if you are using the Commodore assembler's wedge program. Type the symbol @, and your wedge will provide you with an on-screen error message telling you exactly what went wrong.

If you want to use the wedge program on your assembler/editor disk, load it by typing LOAD "DOS WEDGE64",8 followed by the command RUN. Then, instead of typing the line LOAD "EDITOR64",8,1 to load your Commodore 64 editor, type the line %:EDITOR64. Put your editor program into operation with the command SYS49152.

If that all sounds complicated, you may be pleased to learn that there is a much simpler way to load both the DOS wedge program and the Commodore 64 editor program. Just insert your assembler/editor disk and type:

```
LOAD "BOOT ALL",8
```

When you have typed that line, press RETURN, wait for the READY prompt, and then type:

```
RUN
```

and those two simple commands will load the DOS WEDGE64 program, the EDITOR64 program, and the HILOADER64 program (more on that one later) into your computer's memory. You will not even have to type SYS49152 to run your EDITOR64 program; BOOT ALL will take care of that automatically. All you have to do is load BOOT ALL and start programming.

Unfortunately, however, there are some circumstances under which you cannot use the BOOT ALL routine. For example, if you want to use the LOLOADER utility instead of the HILOADER routine, as we will be doing later on in this chapter, you cannot use BOOT ALL.

STORING YOUR PROGRAM

When you have the EDITOR64 up and running, put a formatted disk into your disk drive so that you can store the Assembly language programs that you are writing. Many of the programs in this book build on each other, so if you start saving them now, you can save yourself a lot of typing later.

If you have saved any of the BASIC programs in preceding chapters on a disk, use that same disk for your Assembly language programs. If you have not started a program disk yet, then this is a good time to put a blank but formatted program-storage disk in your disk drive.

USING THE COMMODORE EDITOR

When your editor is up and running, you will see a COMMODORE 64 EDITOR title line and a READY prompt on your computer screen. You can then type Listing No. 1 of the ADDNRS.COM program. To save you some page-flipping, here it is again:

```
AN 8-BIT ADDITION PROGRAM
(Listing No. 1)

10 ;
20 ;ADDNRS.COM
30 ;
40  *=$8000
50 ;
60 ADDNRS CLD
70  CLC
80  LDA #2
90  ADC #2
0100  STA $02A7
0110  RTS
0120  .END
```

MORE NOTES ON SPACING

The Commodore assembler is very fussy about spacing, so be careful about the spacing you use when you type the ADDNRS.COM source-code listing. Here are a few helpful tips about typing spaces in Assembly language programs written using the Commodore 64 assembler:

In the lines that contain semicolons, there should be only one space between the line number and the semicolon. In Line 40, however, there should be two spaces between the line number and the asterisk, since * is a directive and directives appear in the op-code field of Commodore 64 assembler programs.

In Line 60, there should be one space between the line number and the ADDNRS label, and one space between ADDNRS and the mnemonic CLD.

In Lines 70 through 110, there should be two spaces between each line number and the op code that follows. In Line 120, there should be two spaces between the line number and the .END directive.

If you make a mistake while typing a line, move back and correct it in the usual Commodore fashion, using the cursor-control keys on your keyboard.

LISTING YOUR PROGRAM

After you have typed your source listing of Program 1 into your computer, type the word LIST, and you should see a screen display that looks like this:

```
READY
LIST

10 ;
20 ;ADDNRS.COM
30 ;
40  *=$8000
50 ;
60  ADDNRS CLD
70  CLC
80  LDA #2
90  ADC #2
100  STA $02A7
110  RTS
120  .END
READY
```

If you have a printer, you can now print your program on paper, in the same way that you would get a printout of a BASIC program. Just type, for example,

```
OPEN 1,4,4
```

The first number must be any value between 1 and 255; the second is the usual device number for a printer attached to a Commodore. If your printer is not Device No. 4, of course, you have to use a different device number. Once your printer's device channel is open, you can type:

```
CMD 1
```

or CMD followed by whatever other optional number you have chosen. That command routes your computer's output to your printer instead of to your video screen. Then you should be able to type the command:

```
LIST
```

and get a hard-copy printout of the ADDNRS.COM source-code listing.

After you have printed the program, do not forget to type:

```
CLOSE 1
```

so that your screen becomes your primary output device again.

Now, if your listing looks all right, you can save your program on disk. Make sure that your formatted program-storage disk is in your disk drive and that your disk is initialized. Then type:

```
PUT "ADDNRS.COM"
```

The top red light on your disk drive should now go on, and the disk you are storing your program on should start to spin. When the light goes off, your source code should be safely recorded on a disk under the file name ADDNRS.COM.

Are you sure that your program has been stored safely? To find out, type:

```
NEW
```

and then type:

```
GET "ADDNRS.COM"
```

Now type:

```
LIST
```

If you have succeeded in saving your program on your disk, you will see it listed on your screen, but you may notice that there has been a change in the program. The line numbers, instead of running from 10 to 120, extend from 1000 through 1110.

Why did that happen? Your Commodore 64 editor just numbers programs that way, and that convention does make a certain amount of sense. If a routine starts with a line numbered 1000, it is easy to put other routines both in front of it and after it, but if you do not want your source code to start with Line 1000, that is easy to change; your Commodore 64 editor has a line-renumbering utility that is very easy to use. For further details, consult the instruction manual.

ASSEMBLING YOUR PROGRAM

Now you know how to write, save, and load an Assembly language source-code listing using the Commodore 64 assembler. So you are now ready to learn how to use the Commodore assembler to assemble a program.

If you are using your DOS wedge program, it is easy to load the assembler that is on the C64 assembler/editor disk. Just remove the user disk from the disk drive, insert the assembler/editor master disk, and type:

```
/ASSEMBLER64
```

(If you are not using your DOS wedge, type LOAD "ASSEMBLER64",8—if your assembler fails to load, you may never know why.)

When you have loaded your assembler, type:

```
RUN
```

When you have typed that command you will see this prompt:

```
OBJECT FILE (CR OR D:NAME):
```

Now remove your assembler/editor disk from your disk drive, replace it with your own program disk, and type the file name:

```
ADDNRS.ASM
```

The assembler will assign that name to the object file it will soon be creating.

Next you are asked whether you want a hard copy, or printout, of your Assembly-code listing. If you do, just press the RETURN key. If not, type:

```
N
```

Your computer now asks whether you want another kind of file called a cross-reference file (so you can refer in other programs to any labels, constants, or other identifiers you have created in this one). You cannot create an object-code file and a cross-reference file during the same pass through the Commodore 64 assembler, and you have no need for such a file at this point in your study of Assembly language anyway. So press the RETURN key.

Finally you are asked to give the name of the source-code program you want assembled. In response to this prompt, type:

```
ADDNRS.COM
```

Your C64 assembler then assembles your program and provides you with a listing—either on your screen or on paper, depending on what you requested—that looks like this:

```
ADDNRS.COM......PAGE 0001

LINE# LOC   CODE         LINE

00001  0000               ;
00002  0000               ;ADDNRS.COM
00003  0000               ;
00004  0000                      *=$8000
00005  8000               ;
00006  8000  D8           ADDNRS CLD
00007  8001  18           CLC
00008  8002  A9 02        LDA #2
```

```
00009  8004  69 02        ADC #2
00010  8006  8D A7 02     STA $02A7
00011  8009  60           RTS
00012  800A               .END


ERRORS = 00000


SYMBOL TABLE

SYMBOL VALUE

 ADDNRS   8000

END OF ASSEMBLY
```

ERRORS

If you have made any typing errors in your program, this is where you are most likely to find out about them. If your assembler finds an error in a line, it flags the mistake and displays an error message. It may not be able to spot every error you make, but when it does catch one, it prints an error message on your screen or your printout. If you do not understand exactly what the message means, you should consult your C64 assembler/editor user's manual.

As the assembler assembles your program, it automatically saves an object-code listing on the disk in your disk drive, using the file name ADDNRS.ASM. If the assembler finds any errors in your program while the program is being assembled and saved, it still saves the assembled program, errors and all.

If there are any errors in your program, reload your EDITOR64 program so you can go back to your original source-code listing and correct them. Then take the following steps, in this order. Erase the incorrectly written program from your disk (a good reason for using your wedge). Reload your assembler, assemble your source code again, and see whether you have made any more errors. If there are any mistakes, then go through each of these steps again until all the errors your assembler has spotted are removed from the program.

When you have assembled your ADDNRS.COM program, and the assembler has saved the assembled version of the program on a disk, you are ready to convert your assembled program into true machine language using your C64 loader utility. We save that step for the next chapter.

THE PANTHER ASSEMBLER

To load the Panther C64 assembler into your computer's memory, all you have to do is type:

```
LOAD "ASM",8,1
```

or, if you have turned your computer on with the Panther disk in your disk drive, you can type:

```
LOAD "*",8,1
```

While the Panther assembler is loading, there will probably be a lot of disk drive noises and a lot of flashing of your disk drive's error light, but do not worry because that is normal with the Panther C64.

When the Panther assembler is loaded, you will see a] prompt on your screen, followed by a flashing cursor. Then you can start programming.

The Panther assembler, like the Commodore, uses BASIC-style line numbers, but it does not allow any spaces to be inserted between line numbers and labels; in a program using the Panther C64, line numbers and labels are jammed right up next to each other, with no spaces in between. That is probably why programs written using the Panther are always typed in lowercase; if the letters and numbers on a Panther source-code listing were all the same size, it would be difficult to distinguish where line numbers ended and labels began.

Here is the ADDNRS program typed on a Panther C64 assembler:

```
10;
20;addnrs.pan
30;
40 org $2000
50 obj $2000
60addnrs cld
70 clc
80 lda #2
90 adc #2
100 sta $02a7
110 rts
120 end
```

If you have a Panther assembler, this is a good time to get it up and running and type in the addnrs.pan program. When you have it all typed, you can type the word *list* after the] prompt, and this is what you see:

```
]list
   10;
   20;addnrs.pan
   30;
   40           org $2000
   50           obj $2000
   60addnrs     cld
   70           clc
   80           lda #2
   90           adc #2
```

```
100          sta $02a7
110          rts
120          end
```

The addnrs.pan program has one line that the ADDNRS.MER and ADDNRS.COM programs do not. The extra line is:

```
50 obj $2000
```

This line was included in the Panther program because the Panther C64 does not assemble a source-code listing into object code unless it is specifically instructed to do so. We will soon want an object-code listing of the addnrs.pan program, so an *obj* directive has been written into the program.

You may also notice that the *org* directive in Line 40 specifies a different object-code starting address from those in the other versions of the program. The origin directive in the Merlin and Commodore versions of the program directed that the object code begin at Memory Address $8000, but Line 40 of this listing specifies a starting location of Memory Address $2000. That is because the symbol tables that are generated by the Panther assembler/editor are stored in a block of memory that extends from $8000 to $8FFF. So the addnrs.pan program cannot start at Memory Address $8000. If it did, it would consume some of the memory space which the Panther uses for symbol tables. The Panther assembler does have a special *symbol* command that can be used to move the symbol table to another block of memory, but to type and run a program as short as the addnrs.pan routine, it is easier to start the program at another address, such as $2000, than to change the block of memory that the Panther assembler uses for its symbol table.

To assemble a program using the Panther C64 assembler, all you have to do is type the program, and then type the command *asm* following the] prompt. Then the program will immediately be assembled. When you have assembled addnrs.pan, you can save the program's source-code listing on a disk almost as quickly as it was assembled. Just make sure that an initialized disk is in your computer's disk drive, and that the] prompt is displayed on your computer screen. Then type the line:

```
save addnrs.pan
```

to perform the save.

The Panther assembler also has a special directive, the *sav* directive, which can be used to save object code to a disk, but when you use the *sav* directive, what you wind up with on your disk is a nonexecutable ASCII-style text file of your object code listing, not a pure machine-language file that can be run on a computer. So, although the Panther C64 instruction book does not warn you of this, you cannot use the *sav* directive to save a machine-language program in an executable form. The best way to save a machine-language program with the Panther assembler is to activate the assembler's machine-language monitor and issue a *save* command from there. We will do that in the next chapter.

5 Running an Assembly Language Program

From a BASIC Program and on Its Own

There are several ways to execute a machine-language program on a Commodore 64 or Commodore 128. For example, you can run a machine-language program or routine by:

- using a special debugging command that is provided by most assembler monitors—including the Merlin 64, Commodore 64, and Panther C64 assemblers. Monitors are often used to run Assembly language programs while the programs are being written and debugged. After a program has been completed, however, there is rarely any more need to run it using a monitor.

- using the BASIC/DOS command LOAD, as in LOAD "FILENAME",8 or LOAD "FILENAME",8,1.

- calling your machine-language program from a BASIC program using either the BASIC/DOS SYS directive or the Commodore BASIC USR(X) function.

In this chapter, each of these three methods of running machine-language programs will be covered in detail. We will start with the first method: running a machine-language program using a machine-language monitor.

Each of the three assemblers that we have been discussing in this book comes with a machine-language monitor, and, you would probably guess, each of these three monitors works in a slightly different way. So, to keep from confusing things even more than necessary, I will discuss the Merlin monitor, the Commodore 64 monitor and the Panther C64 monitor separately. To learn how to use your assembler's monitor, you will not have to read all three of these sections; it will be sufficient to read only the section dealing with your assembler.

THE MERLIN 64 MONITOR

If you have just finished Chapter 4, and still have your Merlin 64 data disk in your disk drive, you can start using your assembler's debugging facility immediately. But first, let's give readers who have turned their computers off between chapters a chance to get their systems up and running again.

If you have turned off your computer since the end of Chapter 4, please set your Commodore up again, with your master disk booted, your data disk in your disk drive, and the ADDNRS.MER program loaded into your computer's memory. When you have loaded the program, type L after the : prompt, and the program's source code will be listed on your screen.

When you have listed your program, you can type ASM, and your program will be assembled into machine language. Then you can type the command MON, and your Merlin assembler will go into its monitor mode. You will be able to tell easily when Merlin has exited to its monitor; the colon prompt on your screen will change to a dollar sign.

To run a program on the Merlin 64 monitor, the command to use is the g command. So when the $ prompt appears on your screen, type 8000g and the line that starts with the dollar sign should look like this:

```
$8000g
```

When you have typed that line on your screen, press the RETURN key. If everything is in order, you will not see anything special happen right away; all Merlin will present you with is another dollar sign. If so, you have just passed an important milestone on your quest to learn Assembly language; you have just run your first Assembly language program.

To see whether everything really did work, you can use another monitor command: the h command. Here is how the h command works:

As you may remember, the function of the ADDNRS.MER program is to add the digits 2 and 2, and to place their sum in Memory Address $02A7. When Merlin is in its monitor mode, the command that is used to peek into specific memory locations—or a series of memory locations—is the h command. So let's use the h command now.

After the dollar sign that you now see on your screen, type the string 02a7h and your command line should look like this:

```
$02a7h
```

Then press RETURN, and Merlin will display a line like this on your screen:

```
02A7-04 |
```

In case you are wondering, the vertical line after the number 02A7-04 means that Merlin has peeked into Memory Address $0287 and has found that it contains the value 4, or the sum of 2 and 2. So if that is the line you see on your screen now, you have just run your first machine-language program.

There are also a few other commands that can be used with Merlin's machine-language monitor. For example, there is the command :, which looks just like—but is not used like—the colon prompt in Merlin's executive mode. In the monitor mode, the : command is used to assemble values directly into memory addresses in the Commodore's memory. For example, type the line:

```
$9000: 02 02 02 02
```

and you will poke four 2's in Memory Locations $9000, $9001, $9002, and $9003. Do that right now, and you immediately can see if it worked by using the h command.

The letter l (a small L, not the digit one) is another useful Merlin monitor command. Type an l command (or a string of l commands) following a $ prompt, and Merlin will print out a source-code listing—or a disassembled listing—of any Assembly language program that starts at the address given. For example, in response to the line:

```
$8000l
```

—Merlin will display a source-code listing of the ADDNRS.MER program.

There are a few more commands that can be used with the Merlin monitor, and a listing of them can be found on pages 79 and 80 of the instruction guide that comes with the Merlin 64 assembler. For now, though, the only additional commands that are really important to remember are the $r command, which will return you to Merlin's editor mode, and the $q command, which will return you to the executive mode of your Merlin 64 assembler.

THE COMMODORE 64 MONITOR

The monitor that comes with the Commodore 64 Macro Assembler Development System is like the rest of the Commodore 64 system: it is extremely powerful and versatile, but also quite complicated.

When beginning-level Assembly language programmers try to use the monitor provided with the Commodore 64 assembler, they often find that they cannot get it to work. That is because the monitor will not work with raw code that is generated by the Commodore assembler. The problem is that the assembler in the Commodore 64 assembler/editor system does not produce pure binary machine-language code; instead, it generates ASCII-type listings that must be converted into binary code before they can be used by the Commodore 64 monitor. The utility that makes that conversion is a *loader* program that is also provided in the Commodore 64 assembler package.

Actually, there are two loaders in the Commodore assembler kit. One, called LOLOADER64, resides in a block of RAM that starts at Memory Address $0800 when it is loaded into a Commodore 64's memory. The other program, HILOADER64, is designed to be loaded into memory beginning at $C800.

(It might be helpful to point out at this juncture that the names of the loader programs that are printed on page 27 of the Commodore 64 assembler instruction manual are incorrect. On the Commodore 64 assembler/editor disk, the programs are called LOLOADER64 and HILOADER64, not LO-LOAD.C64 and HI-LOAD.C64, the names given in the instruction manual. If you try to load the programs using the latter file names, the result will be a loading error.)

In this chapter, we will use the LOLOADER64 program. To load it into your computer's memory, put your master Commodore 64 assembler/editor disk into your disk drive, and type:

```
LOAD "LOLOADER64",8
```

When the LOLOADER64 program has been loaded, type the command:

```
RUN
```

—to run the program.

If everything is working right, the next thing you see on the screen is a couple of title lines, followed by the line:

```
HEX OFFSET (CR IF NONE) ?
```

There is no need to worry right now about what this question means. In case you are curious, though, the "hex offset" referred to here is a two-byte hexadecimal value that you can add to the address given in the origin line of a program if you want the program to start at an address different from the one specified in the origin line. Matters like this fall into the category of memory organization, which will be the topic of a later chapter. For now, let's just assume that we do not want to use a hex offset (which is true), and press a RETURN.

After you have pressed RETURN, you will see a line on your screen that says:

```
OBJECT FILE NAME ?
```

Now you should try to locate the data disk you used in Chapter 4—the one on which you stored your ADDNRS.ASM program. When you have found that disk, insert it into your disk drive and respond to the OBJECT FILE NAME? prompt with the line:

```
ADDNRS.ASM
```

—followed by a RETURN.

Now you will see a display on your screen that looks like this:

```
8000.
8009
END OF LOAD
READY.
```

That means that a true machine-language version the ADDNRS.ASM has been loaded into your computer's memory, and is now just sitting there in RAM, waiting for you to make the next move.

Now we can get back to the current program. Remove your data disk from your disk drive, and reinsert your Commodore 64 assembler disk. Then, in response to the READY prompt on your computer screen, type the line:

```
LOAD "MONITOR$C000",8,1
```

—and press RETURN.

When you have typed LOAD "MONITOR$C000",8,1 your computer will respond with a READY prompt. Then you can type:

```
SYS 49152
```

When your monitor has been loaded into your computer's memory, it will identify itself by displaying a tiny period on your computer screen. This period is a prompt designed to show you that your monitor is loaded and functioning.

Now for a brief digression. Often, when you load the Commodore monitor into memory from a cold start, you will see a display on your screen that looks something like this:

```
   PC  SR AC XR YR SP
.;C03E 32 00 C3 00 F6
.
```

If you do not see that display right now, do not worry. If you have been following the instructions in this chapter, it will not be there, but, to prepare you for the occasions when you *will* see it, I will explain right now that it is nothing but a display of the contents of the six registers in your computer's 6510/8502 chip: the program counter, the processor status register, the accumulator, the X register, they Y register, and the stack pointer. Sometimes it helps to know the contents of those registers when you start to debug a program, so the Commodore assembler usually presents you with a list of their contents when you first load your monitor.

Now that you have activated your monitor, you can easily check to see whether the ADDNRS program has really been loaded into RAM. Following the . prompt, just type:

```
D 8000 8009
```

—and your Commodore should respond with a disassembled (source-code) listing of the ADDNRS program.

Now that you have your monitor up and running, you can easily run the ADDNRS program, too. After the . prompt, merely type:

```
G 8000
```

—and press RETURN. Your computer should respond with another READY prompt—that will mean that *something* has definitely happened, since you have now exited your monitor, and are back in BASIC again.

Why is that? It is because the ADDNRS program ends with an RTS instruction, and when the Commodore 64 monitor encounters an RTS instruction without any address to return to, it returns control to BASIC.

There is, by the way, an Assembly language instruction that can prevent the Commodore monitor from returning to BASIC after it finishes running a program using the G command. The instruction that can keep this from happening is the mnemonic BRK (which stands for "break.") Programmers often use the BRK mnemonic when they are debugging programs. By putting a BRK instruction at the end of an Assembly language routine, you can debug your program without having to worry about your assembler jumping back into BASIC every time it comes to the end of the routine. When you have finished debugging a program, though, you should always remember to remove your BRK instructions, since they can crash programs when the programs are run outside a monitor environment.

However, that is not our problem right now. What we want to do now is get our assembler back *into* its monitor mode—and that is easy enough to do. Just type SYS 49152—the same command that you used to activate your monitor in the first place—and your assembler will be back in its monitor mode again.

When the . prompt has returned to your screen, you can check to see whether your monitor really did run the ADDNRS program successfully. Just type:

```
M 02A7
```

—and your monitor will respond with a line like this:

```
.:02A7 04 00 00 00 00 00 00 00
```

That is a listing of the contents of Memory Location $02A7 and the next seven addresses in your computer's memory. Since the first number in that listing is 04, we now know that the ADDNRS program worked; it added the numbers 2 and 2, and stored their sum in Memory Address $02A7.

Now that you know that the ADDNRS program works, there is only one more thing to do—save it on a disk. The way to do that is with another monitor instruction: the S command.

To use the S command, all you have to do is respond to your monitor's . prompt with a line like this:

```
.S"ADDNRS.OBJ",08,8000,800A
```

The 08 in this series is the standard Commodore device number for a disk drive, but note that in this case, the 8 that is usually used to specify a disk drive is preceded by a zero. If the zero is omitted, your monitor's S command will not work.

The number that follows the number 08 is always the beginning address of the machine-language program being saved to disk, and the next number is always the address of the last byte in the program being saved, *plus one*. In this case, the addresses 8000 and 800A are the beginning address, and the ending address (plus one), of the ADDNRS program.

When you have used the S command to save the ADDNRS.OBJ program, you will have the program stored on your disk in three forms—its original source-code form (ADDNRS.SRC), an ASCII-style text file of the assembled program (ADDNRS.ASM), and an executable machine-language program, or binary file (ADDNRS.OBJ). You will always wind up with these three types of files when you write, assemble, and save a program using the Commodore 64 assembler/editor system.

THE PANTHER C64 MONITOR

The Panther C64 monitor is very easy to to use, but also annoyingly finicky about accepting commands—and a little inconsistent, too. For example, some commands demand that a space be typed in a certain place, and in other commands there can be no space in that position. So when you use the Panther monitor, be sure to enter your commands *exactly* as they are typed in the following examples.

Before you can run the addnrs.obj program using the Panther monitor, you will have to boot your Panther C64 disk and then put the assembler into its monitor mode by typing the command break. A list of the contents of your 6510/8502's memory registers, followed by a . prompt, should then pop onto your screen. That will show you that your monitor is now up and running.

Next, locate the data disk you used in Chapter 4—the one with the addnrs.pan and addnrs.obj programs on it. Slip your data disk into your disk drive, and type the line:

```
l"addnrs.obj",08
```

—following the . command. Your assembler will then respond with a list of the contents of your 6510/8502 chip's six registers, followed by a . prompt. You can then answer the period prompt by typing the line:

```
d 2000
```

—and your computer will acknowledge that command by presenting you with a disassembled listing of the addnrs.obj program that you have loaded into its memory.

Now that you know your program has been loaded correctly, you can run it. To run the addnrs program, move your cursor to the bottom of your screen using computer's down-arrow key, and press RETURN. Then, when you see your monitor's . prompt, type the line:

```
g 2000
```

If everything goes well, your assembler will reply to that command by exiting to its editing mode. You can then type a break command to get your assembler back into its monitor mode. Once you are there, you can issue the command:

```
m 02a7 02a7
```

In response to this line—in which the address 02A7 has been typed twice, since the Panther monitor requires two numbers after an m command—your monitor should present you with a listing of the contents of Memory Location $02A7 and the next seven addresses. That display will look like this:

```
.:02A7 04 00 00 00 00 00 00 00
```

Since the first number on the list is 04, you can tell that the ADDNRS program worked; it added the numbers 2 and 2 correctly, and stored their sum in Memory Address $02A7.

OTHER WAYS TO RUN A PROGRAM

Here is a new program that we will use to illustrate how programs written in Assembly language can be run using DOS and BASIC commands. When you run this program, it will display a character on your computer screen. Type it, assemble it, save it on a disk (in both its source-code and object code versions), and then run it. When you have done all of that, I will explain how the program works and what it does.

This is how you should type the program if you are using a Merlin 64 assembler:

```
*
* PRINTIT
*
CHROUT   EQU    $FFD2
         LDA    #$58
         JSR    CHROUT
         RTS
         END
```

(While typing this program, astute readers may notice that it contains no ORG directive. Programs written using the Merlin assembler do not absolutely have to have origin lines; if a Merlin program lacks such a line, Merlin will automatically assign it a default starting address of $8000.)

The Commodore 64 assembler and the Panther C64 assembler do require origin lines, however. Here is a listing of the PRINTIT program as it should be typed if you are using a Commodore assembler:

```
1000 ;
1010 ;PRINTIT.COM
```

```
1020 ;
1030  *=$8000
1040 ;
1050 CHROUT=$FFD2
1060 ;
1070 PRINT
1080  LDA #$58
1090  JSR CHROUT
1100  RTS
1110  .END
```

Finally, here is the program once again, as typed using the Panther C64 assembler:

```
10;
20;printit.pan
30;
40 org $2000
50 chrout equ $ffd2
60 lda #$58
70 jsr chrout
80 rts
90 end
```

HOW IT WORKS

To understand how the PRINTIT program works, you have to have at least a passing familiarity of a block of memory in your computer which Commodore calls a kernal, and before you can understand the Commodore kernal works, it helps to have a basic understanding of a type of machine-language routine called a vector.

In assembly language terminology, a *vector* is a subroutine that does nothing but cause a program to jump to another subroutine. That does not sound very useful unless you understand what vectors are for, but once you learn how vectors are used in the Commodore kernal and similar types of jump tables, their usefulness becomes clear.

The Commodore kernal is a *jump table,* a table of jump addresses, that resides in a special block of RAM from $E000 to $FFFF. The kernal was designed because the operating systems of microcomputers often change without notice, creating a potentially serious problem for programmers and computer owners. After a computer has been on the market for a while, bugs in its operating system are often discovered, and these have to be fixed. Operating systems are also sometimes altered just so they will run faster or work better in some other way.

That is where vectors come in. When the operating system of a computer is revised, the addresses of various routines in that operating system are often altered. This is one reason that programs written for early models of a computer sometimes do not work on later models, with different operating systems.

THE COMMODORE AND THE KERNAL

To prevent this kind of disaster, the addresses of important routines in a computer's operating system are often kept in a jump table, such as the Commodore kernal. A programmer who wants to use a routine included in the jump table writes a program that jumps to a *call address*, that is, to an address in the jump table, instead of to the actual address of the routine.

To make sure this system works, the manufacturers of computers that contain jump tables traditionally guarantee that the addresses in the tables will always be kept up to date and therefore always remain accurate. Programmers who use these "official" vectors in their assembly language programs can therefore be certain that the programs they write for a given computer will work, no matter how many times the operating system is changed.

That brings us back to PRINTIT. Examining the program, you will see that a constant labeled CHROUT is used in each of the three versions of the routine. CHROUT is one of the most important routines in the Commodore kernal—it is used to print a character on the screen. When you call the CHROUT routine, it checks whatever value is currently in the accumulator, interprets that value as an ASCII code, and outputs the appropriate ASCII character to whatever peripheral device is open—usually the screen.

The inclusion of the CHROUT routine in the Commodore kernal is a real boon to everyone who writes Assembly language programs for Commodore computers. What the CHROUT routine means to you is that you will never have to write an Assembly language routine that will print a character on a screen—and as legions of Assembly language programmers can tell you, writing even a simple PRINT routine in Assembly language can be quite a task. On computers that do not have kernals, writing a routine that will print a single character on a screen can require several pages of machine code, but thanks to the Commodore kernal, you can print a character on your screen by using a single command.

Later on in this book, there will be more information on your computer's kernal and how to use it. There is also an extensive section on the Commodore kernal on pages 269–305 of the *Commodore 64 Programmer's Reference Guide*. If you would like to learn more about the tremendous power of the Commodore kernal, it would be a good idea to study that part of your *Programmer's Reference Guide* very carefully. In that section, the people who designed your computer explain what each routine in the Commodore kernal does, and exactly what steps must be taken to call and use each one.

In the version of the program written on the Merlin 64 assembler, CHROUT is defined in Line 4, the line that reads:

```
4 CHROUT    EQU    $FFD2
```

In the variation of the program written on the Commodore 64 assembler, CHROUT is defined in Line 1050:

```
1050 CHROUT=$FFD2
```

In the Panther 64 version of the routine, the line defining CHROUT is Line 50:

```
50 CHROUT EQU $FFD2
```

As you can see by comparing each of these lines, the Commodore 64 assembler uses the = to define constants, while the Merlin and Panther assemblers use the directive EQU. Another constant that is used in all three versions of the program is the literal number #$58. That is the ASCII code for the character X, the character which the program will print on your computer screen.

HOW IT'S DONE

Once the constant CHROUT has been defined, the PRINTIT program is very straightforward. The accumulator is loaded with the literal number $58 (decimal 88, the ASCII code for an X), and then the mnemonic JSR is used to make the program jump to the call address of the CHROUT routine. CHROUT then takes the number $58 from the accumulator, interprets that value as the ASCII code for the letter X, and prints an X on the screen.

RUNNING THE PRINTIT PROGRAM

By inserting a BRK instruction before the RTS instruction in the PRINTIT program, you could run the program using your assembler's machine-language monitor, but you will not be using your monitor to run the program in this chapter. Instead, we will execute it using the two methods most often employed to run Assembly language programs. First, we will run the program using a DOS command, and then we will boot it and run it from a BASIC program.

It is extraordinarily easy to run an Assembly language program using a DOS command. If you have assembled the PRINTIT program and saved its object code on a disk, you load the program into your computer by simply typing a line like this:

```
LOAD "PRINTIT.OBJ",8,1
```

—or some other appropriate version thereof. As you may know, the last number in the above line—the 1—is a signal to your computer that the program to be loaded is a machine-language program, not a BASIC program, and must therefore be loaded in the block of RAM which it was written to occupy.

Once you have loaded an object-code program in the above fashion, you can run it by typing a line like this one:

```
SYS 32768
```

In this example, the number 32768 is the decimal form of the hexadecimal number $8000—and $8000 is, of course, the machine-language address of the Merlin and Commodore 64 assembler versions of the PRINTIT program. The address of the PRINTIT.PAN program's object code is $2000, so the DOS command that would be used to run that version of the program is:

```
SYS 8192
```

RUNNING MACHINE-LANGUAGE PROGRAMS FROM BASIC

One disadvantage of running a machine-language program using the SYS command is that you have to remember exactly where in your computer's memory the program resides. If you do not want to have to remember the program's address—or, more important, if you do not want the users of your program to have to remember its address—then you can easily write a short program in BASIC that will eliminate the necessity for using a SYS command.

There is a trick, though, to running a machine-language program from a BASIC program. Here, for example, is a program that will *not* work:

```
10 LOAD "PRINTIT.OBJ",8,1
20 SYS 32768
```

If you load that program and try to run it, all you will wind up with is an endless loop, in which your disk drive loads the PRINTIT.OBJ program into your computer's memory over and over again. Why?

Well, when your computer finishes loading a machine-language program into its memory using a BASIC command, it ordinarily returns to BASIC, and runs any BASIC program that has previously been stored into the section of its memory where BASIC programs are stored.

Now take another look at the two-line program above and try to figure out what happens when it runs. First, it loads the PRINTIT.OBJ program into its memory. Then it returns to BASIC, and runs the program it finds there, and that is the same program it just ran. So the computer loads the PRINTIT.OBJ program again, returns to BASIC, loads it again—and so on.

A NEAT TRICK

Now here is a little trick you can use to avoid that endless loop, and to load and run the PRINTIT program successfully. Just rewrite our two-line load-and-run BASIC program in this manner:

```
10 IF A=0 THEN A=1:LOAD "PRINTIT.OBJ",8,1
20 SYS 32768
```

Type the above BASIC program and run it, making sure that the disk on which you have saved the PRINTIT.OBJ program is in your disk drive. If everything is working properly, the program should automatically load the PRINTIT.OBJ program and run it, as soon as you type the BASIC command RUN.

Take a close look at this little BASIC program and see if you can figure out how it works. Here is the secret: when you create a BASIC variable on your computer—for example, the variable A in the above program—you can be virtually certain that until you give it a value, its value will be zero. So when your computer encounters Line 10 of the program, it will follow both of the instructions in that line; first it will change the value of A to one, and then it will load the PRINTIT.OBJ program.

Then your computer will move on to Line 20. It will run the program (provided the program is stored at $8000; if not, use the correct value). Then the SYS command will re-initialize BASIC and run our little two line program again.

This time, however, there will be an important difference. Now the value of A will be one—so the computer will not execute Line 10. Instead, it will move on to Line 20, and run the program.

That is how the endless loop that was set up by the previous BASIC program can be avoided.

THE USR(X) FUNCTION

There are several other methods for running machine-language programs, and you can find brief descriptions of several of them on pages 307 and 308 of your *Commodore 64 Programmer's Reference Guide*. Most of those techniques are a bit beyond the scope of this chapter, but one of them—the BASIC function USR(X)—deserves special mention.

The USR(X) function, like the BASIC statement SYS, is used to transfer control of a program from BASIC to a machine-language program, but USR(X) differs from SYS in two important ways. First, the USR(X) function can transfer information back and forth between a BASIC program and a machine-language subroutine—and a SYS statement cannot. In addition, USR(X) is a better tool than SYS for inserting machine-language "patches" into BASIC programs. That is because USR(X) does not return to BASIC in the same way that SYS does. The SYS statement, as we saw in the previous section of this chapter, tends to return to BASIC quite abruptly: by re-initializing your Commodore's built-in BASIC interpreter, and clearing the screen. USR(X) returns to BASIC much more gently; when a machine-language routine is called by USR(X) and then returns to BASIC, the BASIC program that was in progress simply resumes, beginning at the line following the USR call.

Now here is how the USR(X) function works. Before you can use the USR(X) function in a BASIC program, it is necessary to perform one preliminary operation: you have to tell your BASIC interpreter the starting address of the machine-language routine that you will be calling using the USR(X) function. To do that, you poke the starting address of your machine-language routine into Memory Registers 785 and 786 ($0311 and $0312 in hexadecimal notation). These two memory registers are designed to be used together as a single 16-bit register, and are often referred to in combination as USERADD. When the USR(X) function is used in a BASIC program, the first thing it does is look into Registers 785 and 786 for the address of a machine-language routine. Then it calls whatever machine-language routine starts at the address pointed to by

USERADD. So, before you use USR(X), you have to place the address of the machine-language routine which it will call in the USERADD registers.

We will now look at an example of how the USERADD registers can be used in a BASIC program. The two lines of BASIC used in this example are from a program called USRX.BAS. The complete program will be presented later on in this chapter.

```
10 HIBYTE=INT(32768/256):LOBYTE=32768-HIBYTE*256:REM
    DECIMAL 32768=$8000
20 POKE 785,LOBYTE:POKE 786,HIBYTE:ADDRESS OF USRX.S
    PROGRAM
```

These two lines of BASIC use a standard formula—one which you may remember from Chapter Two—to store a 16-bit memory address in the USERADD registers. Once the address of a machine-language routine has been stored in USERADD in this fashion, that routine can be accessed from any BASIC program with the help of the USR(X) function. Here is a line of BASIC that contains a USR(X) function:

```
40 Y=USR(X)
```

This line, like the two that you saw in the preceding example, is from the USRX.BAS program that will be presented in full later on in this chapter. In this line, the USR(X) function is used as part of an equation, and there are two variables in the equation: the X in the USR(X) function, and a Y on the other side of the equal sign. Later on, when we run the USRX.BAS program, the X variable in the USR(X) function will be used to pass a value to a machine-language routine. That routine will perform a calculation, and when control is returned to BASIC, the result of that calculation will be passed back to BASIC as the value of the variable Y. Then the result of the calculation by the machine-language routine can be printed on the screen with a simple BASIC statement such as:

```
50 PRINT Y
```

NOW THE BAD NEWS

Unfortunately, there is one problem that you may face if you ever decide to use the USR(X) function. In order to pass values back and forth between BASIC and machine language using the USR(X) function, it is necessary to express those values in a special form—as floating-point decimal numbers. Floating-point math, as you may know, is a type of math that is employed by most pocket calculators—and by the BASIC interpreter that is built into your Commodore. Programs that use floating-point math are extremely complex and quite difficult to write, so Assembly language programmers usually avoid using floating-point numbers if they possibly can. But floating-point arithmetic has one important advantage over conventional binary arithmetic: unlike binary arithmetic, it always yields results that are 100% accurate. So floating-point math is often used in high-precision, high-performance computer programs.

Because floating-point math is such a complex subject, we will not be using it very much in this book, but that does not mean that we cannot use the USR(X) function. The engineers who designed your computer had enough foresight to know that you (or some other Commodore owner) might someday want to take advantage of the capabilities of the USR(X) function without having to go through the hassle of using it with floating point numbers. To make this possible, the folks at Commodore equipped your computer with a pair of built-in routines that can be used to convert signed binary integers into signed floating-point integers, and vice versa. Once you know where these routines are, and how to use them, you can easily convert binary numbers into floating-point numbers, and vice versa, and that is a great loophole for Assembly language programmers; it means that you can avoid having to use floating-point arithmetic when you want to pass information back and forth between BASIC and machine language.

ANOTHER LOOK AT VECTORS

Now let's take a close at ADRAY1 and ADRAY2, the binary/floating-point routines that are built into your Commodore. ADRAY1 and ADRAY2, like the CHROUT routine that was discussed earlier in this chapter, are vectors: that is, they are memory registers that hold, or point to, the addresses of machine-language routines that are built into your Commodore. As mentioned, computer manufacturers often use vectors to ensure that software designed for a given computer will not become obsolete as the computer is updated and changes are made in its operating system. When a computer-maker builds a vector into a computer, and lists that vector's location in the computer's documentation, this usually constitutes a guarantee that the address of the vector will not change as the computer is updated and improved. When operating systems are improved, the memory addresses of important routines often change, but the addresses of the vectors that *point to* those routines do not. Instead, the *contents* of the vectors are changed so that they point to the new OS routines. So, when computer manufacturers design vectors into their machines and tell software developers where those vectors are, the software designers who use the vectors can be fairly sure that their programs will continue to work as the computers for which they were written are updated and improved.

Now that you know what documented vectors are, and how they are used, we can get on with our examination of ADRAY1 and ADRAY2, the two binary/floating-point conversion vectors that are built into your Commodore.

THE ADRAY1 AND ADRAY2 VECTORS

As you might guess, one of these two vectors is designed to convert floating-point numbers to binary integers, while the other is designed to convert binary integers to floating-point numbers. The floating-point to binary vector is called ADRAY1, and is situated at Memory Addresses $0003 and $0004. The binary to floating-point vector is called ADRAY2, and resides at Memory Addresses $0005 and $0006. At the time this chapter was written, the starting address of the conversion routine pointed to by the ADRAY1 function was $B1AA, and the address of the routine pointed to by the ADRAY2 function was $B391.

To use the ADRAY1 routine to convert a signed floating-point number to a signed binary number, all you have to do is place the signed number which you want to convert into your computer's floating-point accumulator—which, as we have seen, is located at Memory Addresses $61–$66. Then you can jump to the conversion routine pointed to by ADRAY1 with an instruction like this:

```
        JMP ($0003)
```

The syntax used in this statement—which is called an *indirect jump*—will be explained in a later chapter. For the moment, it is sufficient to remember that the statement works. When you have placed a floating-point number in your computer's floating-point accumulator (sometimes called FAC1), and have jumped through the ADRAY1 vector with an indirect jump, the number will be automatically converted into a two-byte signed binary integer. The low-order byte of the integer will be placed in the 6510/8502 chip's Y register, and the high byte will be placed in the 6510/8502's accumulator.

The ADRAY2 vector works just like the ADRAY1 vector, but in reverse. To convert a two-byte signed binary integer into a signed floating-point integer using ADRAY2, all you have to do is place the binary integer to be converted into the 6510/8502's Y and A registers, with the low byte in the Y register and the high byte in the accumulator. Then an indirect jump can be made to the routine pointed to by ADRAY2. When this process is completed, the binary number that was stored in the Y and A registers will be returned as a floating-point number stored in FAC1.

SOME ILLUSTRATIVE PROGRAMS

If this all sounds terribly complicated, an example should make it a little clearer. Here are two type-and-run programs—one in BASIC and the other in Assembly language—which illustrate the use of both the USR(X) function and the ADRAY1 and ADRAY2 vectors. The Assembly language program is called USRX.S, and the BASIC program is called USRX.BAS. To see how these programs work, just type, assemble, and save the USRX.S program, and then type the USRX.S program and save it also. Once you have done that, you can call and run the USRX.S program by simply running the USRX.BAS program.

WHAT THE PROGRAMS DO

As you will see when you run the USRX.BAS and USRX.S programs, they do not really do much; all they are designed to accomplish is to add the literal number 5 to a typed-in number, and then to print the result of that calculation on the screen. However,, even though they might not have much use in the real world, USRX.BAS and USRX.S do illustrate a valuable programming technique. With the USR(X) function—along with a knowledge of Assembly language—you can use Assembly language to write machine-language functions that will perform calculations at lightning speed. Then you can call those routines from BASIC at any time you like. In this way, you can combine the convenience and ease of use offered by BASIC with the speed and versatility of machine language.

This capability can be particularly useful when you are using BASIC to write graphics routines. As you will see later in this book, in a series of chapters devoted to Commodore graphics, there are many kinds of graphics routines that cannot be programmed in BASIC because BASIC is too slow, but when you know how to use the USR(X) function, you can write almost any kind of graphics routine that you like in Assembly language, and then call it from any BASIC program with the help of the USR(X) function.

Even when you do not want to perform machine-language calculations while running BASIC programs, the USR(X) function can be quite useful. For example, when you want to move quietly and easily from BASIC into machine language, and then get back into BASIC just as unobtrusively, the USR(X) function is usually a better choice than the SYS function is. And if you do not have any need to pass variables back and forth when you use the USR(X) function, you do not have to; just use dummy values (values that have no particular meaning) for the X and Y variables in your USR(X) equations. Then you can take advantage of the non-arithmetical virtues of the USR(X) function without concerning yourself at all with the intricacies of signed binary and floating-point numbers.

Now let's take a look at the first program that illustrates the USR(X) function.

THE USRX.S PROGRAM

```
PROGRAM 1: USRX.S
*
* USRX.S
*
          ORG    $8000
*
ADRAY1    EQU    $0003
ADRAY2    EQU    $0005
*
          JMP    START
FTOI      JMP    (ADRAY1)
START     JSR    FTOI
          TAX
          TYA
          CLC
          ADC    #5
          TAY
          TXA
          ADC    #0
          JMP    (ADRAY2)
          END
```

HOW THE USRX.S PROGRAM WORKS

When you have typed and assembled the USRX.S program, I suggest that you save it on a disk under the filename USRX.O. You can then load it, call it, and execute it from within one BASIC program: USRX.BAS. You will get a chance to type and run USRX.BAS after we take a brief look at the USRX.S program.

The USRX.S program makes use of some unusual programming tricks that are interesting and quite imaginative, but also rather weird. The program starts at Line 9, but jumps immediately to Line 11. Then it uses a JSR instruction to jump back to Line 10, but then something very odd occurs. The instruction JSR, as you may recall from previous chapters, stands for "jump to subroutine." However, there is no subroutine to jump to at Line 10. Instead, there is an indirect jump to the ADRAY1 vector, your Commodore's floating-point to integer conversion routine.

To understand why USRX.S jumps to the ADRAY1 vector in such a strange fashion, it helps to know something about how the JSR instruction works in 6510/8502 Assembly language.

In 6510/8502 Assembly language, the instructions JSR (jump to subroutine) and RTS (return from subroutine) usually work together—in much the same way that the instructions GOSUB and RETURN usually work together in BASIC.

In Commodore Assembly language, a statement containing a JSR instruction always requires three bytes: one byte for the machine-language equivalent of the JSR instruction, and two more bytes for an operand.

When a JSR instruction is encountered during the processing of an Assembly language program, the *next* instruction in the program is immediately saved in a special block of memory called the stack. Then the program jumps to whatever address is specified by the operand that follows the JSR instruction.

When the program jumps to that address, what it usually finds is a subroutine which ends with an RTS (return from subroutine) instruction. When an RTS instruction is encountered during the processing of a subroutine, the address that was placed on the stack when the subroutine is called is pulled off the stack. The program then returns to that address—which, as we have seen, is usually the address to which the program should return after the completion of the subroutine.

That brings us to a very unusual feature of the USRX.S program. If the instructions JSR and RTS are supposed to work together, then why is there no sign of an RTS instruction in the USRX.S program?

Well, there is one, but it is well hidden. It is not in the USRX.S program itself, but at the end of the machine-language routine that is pointed to by the ADRAY1 vector that is invoked by the indirect jump in Line 10 of the USRX.S routine.

JUMPING THROUGH THE ADRAY1 VECTOR

Now that you know where the RTS instruction in the USRX.S program is, we are ready to talk about how it works.

As we have seen, the USRX.S program starts at Line 9 with a direct jump to Line 11. Then, in Line 11, a JSR instruction is used to jump back to Line 10.

Now let's consider again just what happens when a JSR instruction is used in an Assembly language program. As I mentioned a few paragraphs back, the first thing a JSR instruction does is place the address of the *next* instruction in the program on your computer's hardware stack. What *is* the next instruction in the USRX.S program? Well, it is the TAX instruction in Line 12. Remember that fact as we continue:

After the JSR instruction in Line 11 has caused the address of the TAX instruction in Line 12 to be saved on the stack, the USRX.S program moves back to Line 10. Then, in Line 10, the program does an indirect jump to your computer's built-in floating-point to integer conversion routine: the routine whose starting address is pointed to by the ADRAY1 vector.

Now this floating-point to integer routine, as I have pointed out, ends with an RTS instruction, and when it reaches that RTS instruction, the address that was placed on the stack back in Line 11 is removed from the stack. Then the program jumps to that address—which, as we saw earlier, is the address of the TAX instruction in Line 12.

THE REST OF THE PROGRAM

Once you know how all of that works, it is not too hard to understand the rest of the USRX.S program. As I explained earlier in this chapter, here is what the Commodore's floating-point to integer routine does: it fetches a signed floating-point integer from the floating point accumulator, converts that integer into a signed binary integer, and places the binary integer in the A and Y registers—with the low byte in the Y register and the high byte in the accumulator, or A register.

When the USRX.S program reaches Line 12, the ADRAY1 vector has done its work and a binary integer is supposed to be in the A and Y registers. In Line 12, the high byte of that integer is moved from the accumulator into the X register. That is a safe place to store it, since the X register is not used for any other kind of chore during the course of the USRX.S program.

Now look at Line 12. When the high byte of the number we are working with has now been tucked safely away, the low byte of the number is moved from the Y register to the accumulator. Then the 6510/8502's carry bit is cleared, and the literal number 5 is added to the number in the accumulator.

In Line 16, this addition operation has been completed, and the sum that has resulted from it is placed in the Y register. Then the high bit of the integer we are working with—which, just a few operations ago, was placed in the X register—is moved back into the accumulator. Then, in Line 18, there is a little trick that is often used in 8-bit Assembly language. A zero is added to the high bit of the number we are working with, along with any carry that may have resulted from adding 5 to the low bit of the number. Since zero plus no carry bit is zero, and zero plus a carry bit is one, this operation results in a carry being added to the high bit of the number, if there is one. Since there is no specific

6510/8502 instruction for adding a carry bit to a byte, this is the way it is often done in 6510/8502 Assembly language.

Now we come to Line 19, where there is an indirect jump to the routine that is pointed to by your computer's ADRAY2 vector—a routine for converting binary numbers into floating-point numbers.

In the USRX.S program, the ADRAY2 vector is a lot easier to jump through than the ADRAY1 vector was. To convert the binary integer now stored in the A and Y vectors into a floating-point number, all we have to do is make an indirect jump through the ADRAY2 vector. When this has been done, the binary number in the A and Y registers—the one to which 5 has been added—will be converted back into a floating-point number and stored in your Commodore's floating-point accumulator. Then it can be passed back to BASIC as the value of Y in the equation Y=USR(X).

The routine vectored through ADRAY2, like the one vectored through ADRAY1, ends with an RTS instruction, but this time, the RTS instruction that ends the ADRAY subroutine brings us right back to BASIC—which is just where we need to be to read the value of Y in the Y=USR(X) equation.

THE USRX.BAS PROGRAM

Now here is the the USRX.BAS program, which calls USRX.O, the object-code version of USRX.S:

```
PROGRAM 2: USRX.BAS

10 HIBYTE=INT(32768/256):LOBYTE=32768-HIBYTE*256:REM
     DECIMAL 32768=$8000
20 POKE 785,LOBYTE:POKE 786,HIBYTE:PRINT
     CHR$(147):REM DEFINE USR(X) ADDRESS AND CLEAR
     SCREEN
30 PRINT ""TYPE IN NUMBER FROM 0 TO 32762: '':INPUT X
35 IF X<0 OR X>32762 THEN PRINT:GOTO 30
40 Y=USR(X)
50 PRINT:PRINT X;"" + 5 = '';Y
60 PRINT:GOTO 30
```

Before you can run the USRX.BAS program, you have to load USRX.O into memory. Then type USRX.BAS, save it for future use, and run it. If you have done everything properly, USRX.O and USRX.BAS should work together with no problems.

HOW USRX.BAS WORKS

After figuring out how the USRX.S program works, you should have no problem deciphering USRX.BAS. In Lines 10 and 20 of USRX.BAS, the address of the USRX.O program is stored in the USERADD registers at decimal addresses 785 and 786.

When the address of USRX.O has been stored in the USERADD registers, USRX.BAS asks you to type in a number. The number you type is placed in the Commodore's floating-point accumulator, and USRX.BAS passes control to USRX.S, which then uses the ADRAY1 vector to convert the number in the floating-point accumulator into a binary integer. Next, USRX.S performs a simple mathematical operation on that binary integer; the result is placed in the A and Y registers of the Commodore's 6510/8502 processor.

When all that is done, USRX.S performs an indirect jump to the ADRAY2 vector. ADRAY2 converts the binary integer in the A and Y registers into a signed floating-point number, which is placed in the computer's floating-point accumulator. Finally, control is passed back to BASIC, and the result of the operation performed by USRX.S, which is now in the floating-point accumulator, is passed back to BASIC as the value of X in the equation USR(X) = Y.

Before ending the chapter, I would like to point out one important restriction in the use of the USR(X) function. Since this function uses signed 16-bit binary numbers, it cannot handle any number greater than 32,767 or smaller than −32,768. Also, we have not yet covered negative numbers. Therefore, the user may not type in any number that will result in a sum smaller than 0 or greater than 32,767. If you do not quite understand why these limitations apply to USRX.BAS, do not be concerned—you will find out why in later chapters. Meanwhile, however, keep this limitation in mind if you decide to make use of the USR(X) function.

6 Addressing the Commodore

Telling Your Computer Where to Go

In the very first chapter of this book, I told you that there is a direct one-to-one correlation between Assembly language and machine language—that for every mnemonic in an Assembly language program, there is a numeric machine-language instruction that means exactly the same thing.

That is the truth, but it is not quite the whole truth. 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 6502/6510/8502 Assembly language.

In the world of Assembly language programming, an addressing mode is a technique for locating and using information stored in a computer's memory, and your Commodore's 6510/8502 chip can access the memory locations in your computer in thirteen different ways. So the 6510/8502 processor has thirteen different *addressing modes*.

In this chapter, we will examine all 13 of these addressing modes, and observe how they are used in 6502/6510/8502 Assembly language. Table 6-1 shows the eight addressing modes that can be used with the mnemonic ADC.

Look closely at Columns 2 and 3 in the following table, and 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—the Assembly language column—all eight addressing modes use the same mnemonic, but each uses a different operand, but in Column 3—the machine-language column—the statements have a different structure. In the machine-language column, each address uses a different op code, but there are only two kinds of operands; each two-byte statement includes a one-byte operand, and each three-byte statement has included a two-byte operand. That fact illustrates an important difference between Assembly language and machine language. In 6510/8502 *machine language,* the 13 available address modes are distinguished by differences in their *op codes.* On the other hand, in 6510/8502

Table 6-1 ADC ADDRESSING MODES

COLUMN 1: ADDRESSING MODE	COLUMN 2: SAMPLE STATEMENT	COLUMN 3: MACHINE-CODE EQUIVALENT	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

Assembly language, the 13 available addressing modes can be identified by differences in their *operands.*

Table 6-2 shows all 13 of the addressing modes that can be used with the 6510/8502 chip in your Commodore.

Table 6-2 THE 6510/8502's ADDRESSING MODES

	ADDRESSING MODE	FORMAT
1.	Implicit (Implied)	RTS
2.	Accumulator	ASL A
3.	Immediate	LDA #2
4.	Absolute	LDA $02A7
5.	Zero Page	STA $FB
6.	Relative	BCC LABEL
7.	Absolute Indexed,X	LDA $02A7,X
8.	Absolute Indexed,Y	LDA $02A7,Y
9.	Zero Page,X	LDA $FB,X
10.	Zero Page,Y	STX $FB,Y
11.	Indexed Indirect	LDA ($FD,X)
12.	Indirect Indexed	LDA ($FD),Y
13.	Unindexed Indirect	JMP

ADDNRS.SRC REVISITED

To see how some of these instructions work, we will take another look at ADDNRS.SRC, the 8-bit addition routine that has been used several times in this book already. Here is the program again, as it would be typed using the Commodore 64 assembler system. (If you have a Merlin 64 or Panther C64 assembler, you can probably alter the program to meet your assembler's demands without too many problems by now, since the important differences in the formats used by these assemblers have been described in previous chapters.)

```
THE ADDNRS SOURCE PROGRAM

10 ;
20 ;8-BIT ADDITION PROGRAM
30 ;
40  *=$8000
50 ;
60 ADDNRS CLD ;IMPLIED ADDRESS
70  CLC ;IMPLIED ADDRESS
80  LDA #02 ;IMMEDIATE ADDRESS
90  ADC #02 ;IMMEDIATE ADDRESS
100 STA $FB ;ZERO-PAGE ADDRESS
110  RTS ;IMPLIED ADDRESS
```

In this example, the three address modes used in the program are identified in the comments column. Let's look now at each of these three address modes.

IMPLICIT (OR IMPLIED) ADDRESSING (Lines 60, 70 and 110)

Formats: CLD, CLC, RTS, etc. When you use *implicit addressing,* all you have to type is a three-letter Assembly language instruction; implicit addressing does not require (in fact does not allow) the use of an operand.

The instruction in an implied address is thus similar to an intransitive verb in English; it has no object. The address it refers to—if it refers to an address at all—is not specified, but merely *implied* by the the mnemonic itself.

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, PLA, PLP, RTI, RTS, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS, and TYA.

IMMEDIATE ADDRESSING (Lines 80 and 90)

Formats: LDA #02, ADC #02, etc. When *immediate addressing* is used in an Assembly language instruction, the operand that follows the op-code mnemonic is a literal number—not the address of a memory location. So in a statement that uses immediate addressing, 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 stuffed directly into the accumulator. Then whatever operation the statement calls for can be immediately performed.

Instructions that can be used in the immediate address mode are ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA, and SBC.

ZERO-PAGE ADDRESSING (Line 100)

Formats: STA $FB, etc. It is not difficult to distinguish between a statement that uses immediate addressing and one that uses *zero-page* addressing. In a statement that uses zero-page addressing, the operand always consists of just one byte—a number ranging from $00 to $FF. That number equates to an address in a block of RAM called Page Zero.

The # symbol is not used in zero-page addressing because the operand in a statement that employs zero-page addressing is always a memory location, never a literal number. So the operation called for in the statement is performed on the *contents* of the specified memory location, not on the operand itself.

Zero-page addresses use one-byte operands because that is all they need. As we just said, the memory locations they refer to are in a block of your your computer's memory that is called, logically enough, Page Zero, and to address a memory location on Page Zero, a one-byte operand is all that is necessary.

Specifically, the memory block in your computer known as Page Zero extends from Memory Address $00 through Memory Address $FF. You could just as easily (and just as correctly) say that Page Zero extends $0000 to $00FF, but it is not really necessary to use those extra pairs of zeros when you want to refer to a zero-page address. When you follow an Assembly language instruction with a one-byte address, your computer knows that the address is on Page Zero.

Since zero-page addresses use memory-saving one-byte operands, Page Zero is the high-rent district in your Commodore's RAM; it is such a desirable piece of real estate, in fact, that the people who designed your computer took most of it for themselves. Most of Page Zero is used up by your computer's operating system and other essential routines, and not much space has been left there for user-written programs.

Later on in this book—in a chapter dedicated to memory management—we will discuss the memory space available on Page Zero in more detail. For now, the most important fact to remember about Page Zero is that it is an address mode that uses a memory address on Page Zero as a one-byte operand.

Instructions that can be used with zero-page addressing are ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, and STY.

NEW ADDRESSING MODES

Now we will describe the five 6510/8502 address modes we have not covered so far.

ACCUMULATOR ADDRESSING

Format: ASL A *Accumulator addressing* mode is used to perform an operation on a value stored in the 6510/8502 processor's accumulator. The command ASL A, for example, is used to shift each bit in the accumulator by one bit position, with the leftmost bit (Bit 7) dropping into the carry bit of the processor status (P) register. Other instructions that can be used in the Accumulator addressing mode are LSR, ROL, and ROR.

ABSOLUTE ADDRESSING

Format: STA $02A7 *Absolute addressing* is very similar to zero-page addressing. In a statement that uses absolute addressing, the operand is a memory location, not a literal number, and the operation called for in an absolute-address statement is always performed on the value stored in the specified memory location, not on the operand itself.

The difference between an absolute address and a zero-page address is that an absolute-address statement does not have to be on Page Zero; it can be anywhere in free RAM. So an absolute-address statement requires a two-byte operand—not a one-byte operand, which is all that a zero-page address requires.

This is what the Commodore assembler version of our ADDNRS program would look like if absolute addressing, instead of zero-page addressing, were used.

```
THE ADDNRS SOURCE PROGRAM
(With Absolute Addressing in Line 100)

10 ;
20 ;8-BIT ADDITION PROGRAM
30 ;
40  *=$8000
50 ;
60 ADDNRS CLD ;IMPLIED ADDRESS
70  CLC ;IMPLIED ADDRESS
80  LDA #02 ;IMMEDIATE ADDRESS
90  ADC #02 ;IMMEDIATE ADDRESS
100 STA $02A7 ;ABSOLUTE ADDRESS
110 RTS ;IMPLIED ADDRESS
```

The only change that has been made in this program is the one in Line 100. The operand in that line is now a 2-byte operand, and that change makes the program one byte longer, but now the address in Line 100 no longer has to be on Page Zero. Now it can be the address of any free byte in RAM.

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, and STY.

RELATIVE ADDRESSING

Format: BCC LABEL *Relative addressing* is an address mode used for a technique called *conditional branching*—a method for instructing a program to jump to a given routine under certain specific conditions.

There are eight conditional branching instructions—or relative-address mnemonics—in 6510/8502 Assembly language. All eight begin with B, which stands for "branch to".

Examples of the conditional-branching instructions that use relative addressing are:

BCC "branch to a specified address if the carry flag is clear," BCS "branch to a specified address if the carry flag is set," BEQ "branch to a specified address if the result of an operation is equal to zero," and BNE "branch to a specified address if the result of an operation is not equal to zero."

All eight of the conditional branching instructions will be described later on in this book, in a chapter devoted to looping and branching.

COMPARISON INSTRUCTIONS

The eight comparison mnemonics are often used with three other instructions called *comparison instructions*. Typically, a comparison instruction is used to compare two values, and the conditional branch instruction is then used to determine what should be done if the comparison turns out in a certain way.

The three comparison instructions are: CMP, which means "compare the number in the accumulator with . . ."; CPX, which means "compare the value in the X register with . . ."; and CPY, "compare the value in the Y register with. . . ."

Conditional branching instructions can also follow arithmetic or logical operations, and various kinds of testing of bits and bytes.

Usually, a branch instruction causes a program to branch off 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 than another, if the two numbers or equal, or if a certain operation results in a positive, negative, or zero value.

AN EXAMPLE OF CONDITIONAL BRANCHING

Here is an example of an Assembly language routine that uses conditional branching (this routine was typed using a Commodore 64 assembler).

```
AN ADDITION PROGRAM WITH ERROR-CHECKING

10 ;
20 ;ADD8BIT
30 ;
40  *=$8000
50 ;
60 ADD8BIT LDA #0
70  STA $02AA
80 ;
90  CLD
100  CLC
110 ;
```

```
120  LDA $02A7
130  ADC $02A8
140  BCS ERROR
150  STA $02A9
160  RTS
170 ERROR LDA #1
180  STA $02AA
190  RTS
```

This is an 8-bit addition program 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), then there will be an overflow error in addition, and the carry bit of the processor status register will be set.

If the carry bit is not set, then the sum of the values in $02A7 and $02A8 will be stored in $02A9. If the carry bit is set, however, this condition will be detected in Line 140, and the program will branch to the line labeled ERROR—Line 170.

In this sample program, an error will cause the values in Addresses $02A7 and $02A8 not to be added. Instead, a flag—the number 1—will be loaded into Memory Register $02AA, and the routine will end.

ABSOLUTE INDEXED ADDRESSING

Format: LDA $02A7,X or LDA $02A7,Y An *indexed address,* like a *relative address,* is calculated by using an offset, but in an indexed address, the offset is determined by the current content of the 6510/8502's X register or Y register.

A statement containing an indexed address can be written using either of these formats:

```
LDA $02A7,X
```

or

```
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.

Here is an example of a routine that makes use of indexed addressing. The routine is designed to move byte by byte through a string of ASCII characters, storing the string in a text buffer. When the string has been stored in the buffer, the routine will end.

The text to be moved is labeled TEXT, and the buffer to be filled with text is labeled TXTBUF.

The starting address of TXTBUF, plus the ASCII code number for a carriage return, are defined in a *symbol table* that precedes the program.

```
ROUTINE FOR MOVING A BLOCK OF TEXT
(Presented as an Example of Indexed Addressing)

10 ;
20 ;DATMOV
30 ;
40  TXTBUF=$02A7
50  EOL=$0D
70 ;
80  *=$8000
90 ;
95  JMP DATMOV
96 ;
100 TEXT .BYTE $54,$41,$4B,$45,$20,$4D,$45,$20
110  .BYTE $54,$4F,$20,$59,$4F,$55,$52,$20
120  .BYTE $4C,$45,$41,$44,$45,$52,$21,$0D
130 ;
140 DATMOV
150 ;
160  LDX #0
170 LOOP LDA TEXT,X
180  STA TXTBUF,X
190  CMP #EOL
200  BEQ FINI
210  INX
220  JMP LOOP
230 FINI RTS
250  .END
```

When the program begins, we know that the string ends with a carriage return (ASCII $0D), as strings often do in Commodore 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 that there are no more characters in the string, and the routine ends.

ZERO-PAGE,X ADDRESSING

Format: LDA $FB,X Zero-Page,X addressing is used just like Absolute Indexed,X addressing. However, the address used in the Zero-Page,X addressing mode must (logically enough) be located on Page Zero. 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.

ZERO-PAGE,Y ADDRESSING

Format: STX $FB,Y Zero-Page,Y addressing works just like Zero-Page,X addressing, but can be used with only two mnemonics: LDX and STX. If it were not for the Zero Page,Y addressing mode, it would not be possible to use absolute indexed addressing with the instructions LDX and STX—and that is the only reason that this addressing mode exists at all.

INDIRECT ADDRESSING

There are two subcategories of indexed addressing: indexed indirect addressing, and indirect indexed addressing.

Both indexed indirect addressing and indirect indexed addressing are used primarily to look up data stored in tables.

If you think the names of the two addressing modes are confusing, you are not the first one with that complaint. I never could keep them sorted out myself until I dreamed up a little memory trick to help eliminate the confusion.

Here is the trick. *Indexed indirect addressing*—which has an X in the first word of its name—is an addressing mode that makes use of the 6510/8502 chip's X register.

Indirect indexed addressing—which does *not* have an X in the first word of its name—uses the 6510/8502's Y register.

Now we will look at each of your Commodore's two indirect addressing modes—beginning with with *indexed indirect addressing.*

INDEXED INDIRECT ADDRESSING

Format: (LDA $FD,X) Indexed indirect addressing works in several steps.

First, the contents of the X register are added to a zero-page address—not to the contents of the address, to the address itself.

The result of this calculation must always be another zero-page address.

When this second address has been calculated, the value that it contains—together with the contents of the following byte—make up a *third* address. The third address is (at last) the address that will finally be interpreted as the operand of the statement in question. An example might help clarify this process.

Let's suppose that Memory Address $B0 in your computer held the number $00, that Memory Address $B1 held the number $80, and that the X register held the number 0.

Here are those equates in an easier-to-read form:

```
$B0 = #$00
$B1 = #$80
X = #$00
```

Now let's suppose you were running a program that contained the indexed-indirect instruction LDA ($B0,X).

If all of those conditions existed when your computer encountered the instruction LDA ($B0,X), your computer would add the contents of the X register (a zero) to the number $B0. The sum of $B0 and 0 would, of course, be $B0. So your computer would go to Memory Address $B0 and $B1. It would find the number $00 in Memory Address $B0, and the number $80 in Address $B1.

Since 6510/8502-based computers store 16-bit numbers in reverse order—low byte first—your computer would interpret the number found in $B0 and $B1 as $8000. So it would load the accumulator with the number $8000, the 16-bit value stored in $B0 and $B1.

Now let's imagine that when your computer encountered the statement LDA ($B0,X), its 6510/8502's X register held the number 04, instead of the number 00.

Here is a chart illustrating those values, plus a few more equates that we will be using shortly:

```
$B0 = #$00
$B1 = #$80
$B2 = #$0D
$B3 = #$FF
$B4 = #$FC
$B5 - #$1C
  X = #$04
```

If *these* conditions existed when your computer encountered the instruction LDA ($B0,X), your computer would add the number $04 (the value in the X register) to the number $B0, and would then go to Memory Addresses $B4 and $B5. In those two bytes, it would find the final address (low byte first, of course) of the data it was looking for—in this case, $1CFC.

Indexed indirect addressing is not used in many Assembly language programs. When it is used, its purpose is to locate a 16-bit address stored in a table of addresses on Page Zero. Since space on Page Zero is so hard to find, it is not very likely that you will ever be able to store many data tables there. So it is not too likely that you will ever find much use for indexed indirect addressing.

INDIRECT INDEXED ADDRESSING

Format: LDA ($FD),Y Indirect indexed addressing is not nearly as rare as indexed indirect addressing. In fact it is quite often used in Assembly language programs.

Indirect indexed addressing uses the Y register (never the X register) as an offset to calculate the base address of the start of a table. The starting address of the table has to be stored on Page Zero, but the table itself does not have to be.

When an assembler encounters an indirect indexed address in a program, the first thing it does is peek into the page-zero address that is enclosed in the parentheses that precede 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 is an example of indirect indexed addressing.

Your computer is running a program and comes to the instruction ADC ($B0),Y. It then looks into Memory Address $B0 and $B1. In $B0, it finds the number $00. In $B1, it finds the number $50, and the Y register contains a 4.

Here is a chart that illustrates those conditions:

```
$B0 = #$00
$B1 = #$50
Y = #$04
```

If these states existed when your computer encountered the instruction ADC ($B0),Y, then your computer would combine the numbers $00 and $50, and would come up (in the 6510/8502 chip's peculiar low-bit-first fashion) with the address $5000. It would then add the contents of the Y register (4 in this case) to the number $5000—and would wind up with a total of $5004.

That number—$5004—would be the final value of the operand ($B0),Y. So the contents of the accumulator would be added to whatever number was stored in Memory Address $5004.

Once you understand indirect indexed addressing, it can become 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.

UNINDEXED INDIRECT ADDRESSING

Format: JMP ($02A7) Unindexed indirect addressing is a special kind of addressing that can be used with only one 6510/8502 mnemonic: the JMP instruction. When unindexed indirect addressing is used, a 16-bit number is placed inside a pair of parentheses that follow the JMP instruction. This number serves as a pointer to a pair of memory registers which, taken together, contain the address to which the desired jump is to be made. Let us suppose, for example, that the memory address $02A7 contained the value $00 and that the address $02A8 held the value $06. Now let us suppose that the statement JMP ($02A7) were included in a Commodore Assembly language program. If that were the case, the program being executed would jump to the address $0600—*not* to the address $02A7, which would be the case if the jump instruction were simply JMP $02A7.

ANOTHER ADDRESS: THE STACK

That completes our examination of the thirteen official addressing modes used in 6510/8502 Assembly language programming. However, as long as we are talking about 6510/8502 addressing modes, I might as well introduce a programming tool that is related very closely to addressing: a tool called the *hardware stack*.

The hardware stack, or simply *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. So the number on top of the stack must be removed before the number under it—which was previously on top—can be accessed.

Although the stacked plate 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—and blocks of RAM do not really move up and down like a stack of plates inside your Commodore. When you place a number on your Commodore's hardware stack, here is what really happens.

Your computer's hardware stack, as I have already mentioned, is situated in a block of memory that extends from Memory Register $0100 to Memory Register $01FF, and this block of memory is used from high memory downward. That is, 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 from-the-top-down storage system the last stack address that can be used is Memory Register $0100.

Your Commodore's 6510 or 8502 chip keeps track of stack manipulations with the help of a special register called the stack pointer (the stack pointer was described briefly back in Chapter 3). When there is nothing stored on the stack, the value of the stack pointer is $FF. Add $100 to that number, and what you get is $01FF—the highest memory address on the stack, and 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 6510 or 8502 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 us suppose, now, that several numbers have been stored on the stack, and let us also suppose that the time has come to *retrieve* one of those values from the stack. If that were the case, what would happen?

You can probably guess the answer to that one. When a number that has been stored on the stack is retrieved, the value of the stack pointer is *incremented by one.* That effectively removes one value from the stack, since it means that the next value that will be stored on the stack will have the same position on the stack as the one that was removed. That is a little tricky to comprehend, given the upside-down nature of the stack. Figure 6-1 shows an empty stack, with the stack pointer pointing to the first available address on the stack: $01FF.

Now let's place a number (which value is arbitrary) on the stack.

Notice in Figure 6-2 that the value of the stack pointer has been decremented, and that the number we have placed on the stack is now stored at the highest address in the stack, Memory Register $01FF.

Now let's place another number (also with an arbitrary value) on the stack.

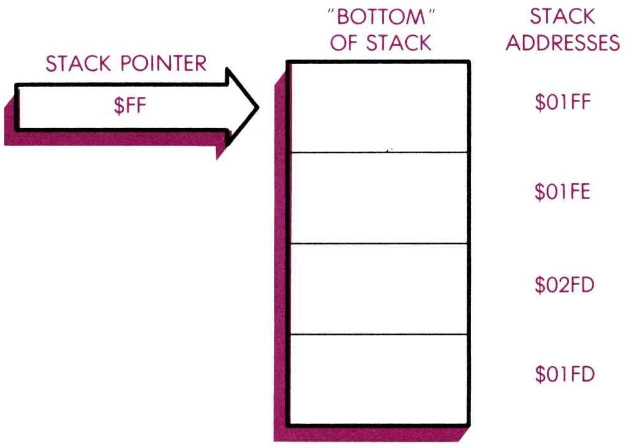

Figure 6-1 How the Stack Pointer Works

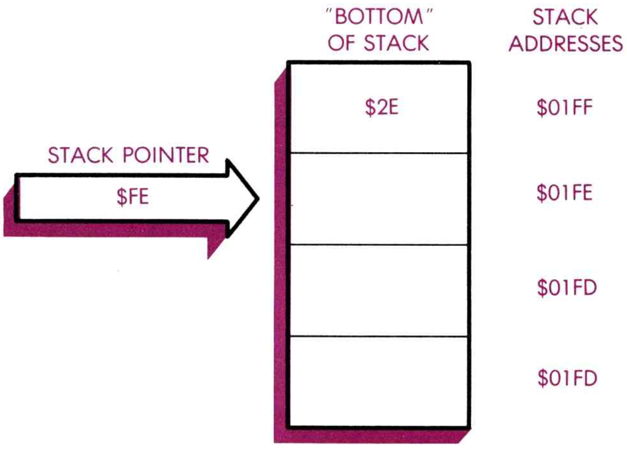

Figure 6-2 Placing a Number on the Stack

The stack pointer in Figure 6-3 has been decremented again, and a second number is now on the stack.

Now let's remove one number from the stack.

As you can see in Figure 6-4, Stack Address $01FE still holds the value $B0, but the value of the stack pointer has been decremented, and now points to Memory Address $01FE. So the next number that is placed on the stack will be stored at Memory Address $01FE, and when that happens, the number previously stored in that stack position—$B0—will be erased. To see how that works, we will now store one more number on the stack. This time, for no special reason, the value of the number placed on the stack will be $17.

See? Memory Register $01FE in Figure 6-5 now holds the value $17. The value of the stack pointer has been decremented, the value $B0 has been erased by the value $17, and the next number placed on the stack will be stored in Memory Register $01FD.

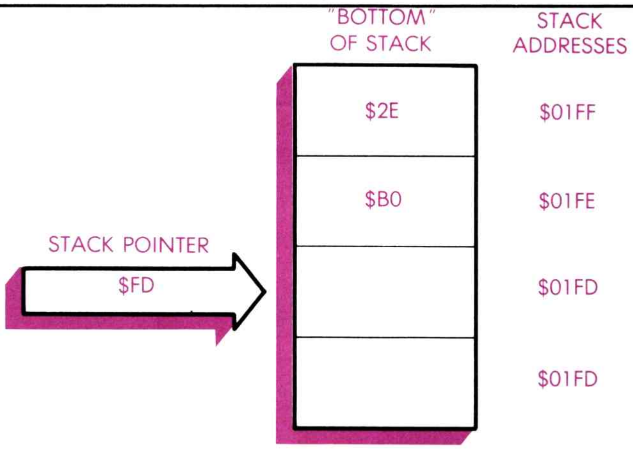

Figure 6-3 Placing Another Number on the Stack

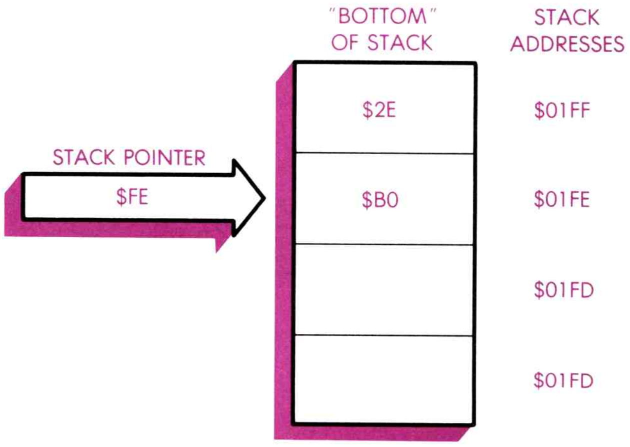

Figure 6-4 Pulling a Number off the Stack

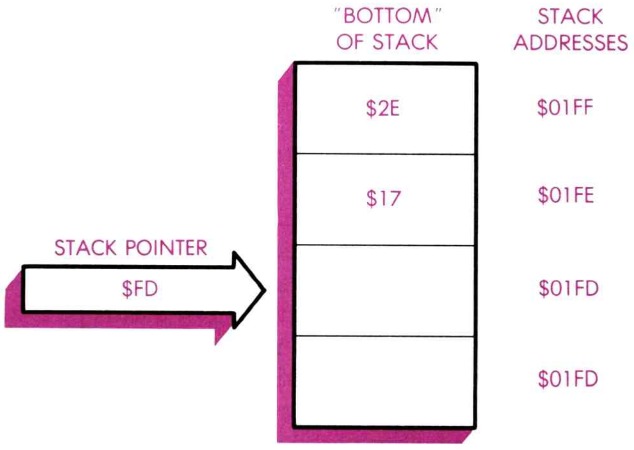

Figure 6-5 One Last Stack Manipulation

HOW OS USES THE STACK

As I have mentioned, the 6510/8502 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 6510/8502 chip takes the memory address that the program will later have to return to, and pushes that address onto 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 6510/8502's program counter. The program can 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 have been using in this book.

A TWO-PART PROGRAM

As you will see, this program is divided into two parts. In Routine 1, we will put two 8-bit numbers on the stack. In Routine 2, we will take them off the stack and add them. Routine 1 should be typed first. Before the program is assembled and executed, however, Routine 2 should be appended to Routine 1.

```
ROUTINE 1: PUTTING TWO NUMBERS ON THE STACK

10 ;
20 ;STACKADD
30 ;
40  *=$8000
50 ;
60  LDA #35 ;(OR ANY OTHER 8-BIT NUMBER)
70  PHA
80  LDA #49 ;(OR ANY OTHER 8-BIT NUMBER)
90  PHA
```

Now let's append Routine 2 to Routine 1:

```
ROUTINE 2: TAKING TWO NUMBERS OFF THE STACK AND
ADDING THEM

100 ;
110 ;WHEN THIS PROGRAM BEGINS, TWO
120 ;8-BIT NUMBERS ARE ON THE STACK
130 ;
140  CLD
150  CLC
160  PLA
170  STA $FD
180  PLA
190  ADC $FD
200  STA $FE
```

```
210  RTS
```

This program—a simple, straightforward 8-bit addition routine—shows how easy and convenient it can be to use the stack in Assembly language programs. In Line 160, a value is pulled from the stack and stored in the accumulator. Then in Line 170, the value is stored in memory address $FD.

In Lines 180 and 190, 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 place in which to store data temporarily. The stack can be a very memory-efficient tool, since it does not require the use of dedicated storage registers. It can 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

Beware: The stack can also be very dangerous for beginning programmers to play around with. When you use the stack in an Assembly language routine, it is extremely important to *leave the stack exactly as you found it when the routine ends.* If you have 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.* Otherwise, there might be "garbage" on the stack when the next routine is called, and that could result in program crashes, memory wipeouts, and various other programming disasters. Remember: Mismanagement of the stack is extremely hazardous to the health of Assembly language programs!

So, if you take care to manage the stack properly—in other words, if you make sure to clear your stored numbers from the stack after each use—it can be a very powerful programming tool.

If you mess up the stack while you are using it, though, you are surely bound for trouble!

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 PHP and PLP operations are often included in Assembly language subroutines so that the contents of the P register will not be wiped out during subroutines. When you jump to a subroutine that may change the status of the P register, it is always a good idea to 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. That way, the P register's contents will not be destroyed during the course of the subroutine.

7 Looping and Branching

GOSUBs and GOTOs in Assembly Language

Now we are going to start having some real fun with Commodore Assembly language. In this chapter, you will learn how to print messages on the screen, how to encode and decode ASCII characters, and how to perform a number of other neat tricks in Assembly language.

We are going to accomplish these feats with some advanced Assembly language programming techniques that we have not tried out so far, along with some new variations on techniques covered in earlier chapters.

These are some of the programming techniques we are going to cover in this chapter:

- Using the Commodore Assembly language .BYTE directive. (For Merlin 64 users, that is the same as the DFB [DeFine Bytes] directive. Panther C64 owners may use the the DFC [DeFine Constant] directive, which also means the same thing.)
- Incrementing and decrementing the X and Y registers.
- Using comparison and branching instructions together.
- Advanced looping and branching.

In this chapter, we will be making extensive use of the CHROUT routine in the Commodore kernal (call address $FFD2). Remember that routine? It is the one you used to print the character X on your computer screen back in Chapter 5.

You will get a chance to print some messages on your screen by incorporating the CHROUT routine into a couple of interesting programs that you will be writing. Here is the first of those programs. It was written using the Commodore 64 Macro Assembler, but you will be provided with all of the instructions that you will need to type and run the program in Merlin-compatible and Panther-compatible versions.

SETTING OUT ON A QUEST

Here is one of the programs that we will be working with for the rest of this chapter:

```
THE QUEST

10 ;
20 ;THE QUEST
30 ;
40  *=8000 ;OR 'ORG EQU $8000' (OR $2000)
50 ;
60 BUFLEN=23
70 CHROUT=$FFD2
80 ;
90  JMP BEGIN
100 ;
110 TEXT .BYTE 87,72,69,82,69,32,73,83
120  .BYTE 32,84,72,69,32,67,79,77
130  .BYTE 77,79,68,79,82,69,63
140 ;
150 BEGIN LDX #0
160 ;
170 LOOP LDA TEXT,X
180  JSR CHROUT
190  INX
200  CPX #BUFLEN
210  BNE LOOP
220  RTS
```

This program is called (for reasons you will soon discover) *THE QUEST*. It is designed to print a cryptic message on your video screen.

Another listing of the same program, but typed using a Merlin 64 assembler, is given in Appendix B.

RUNNING THE QUEST

When you have finished typing THE QUEST, you can run it immediately. Just assemble it, and save it on a disk in both its source-code and object-code versions. For the sake of consistency, I suggest that you save the program under the file names QUEST.SRC, QUEST.ASM (if you use the Commodore 64 assembler), and QUEST.OBJ.

Once THE QUEST program is safely stored on a disk, you can run it using any of the three methods that were covered in Chapter 5: with your machine-language monitor, under DOS control, or from a BASIC program.

When you run the program, you will see what it does, but how does it do it? I will explain.

Let's start with an explanation of the Assembly language .BYTE directive, which appears in Lines 110 to 130. (As previously mentioned, the Merlin 64 equivalent of this directive is DFB, and the Panther C64 equivalent is DFC.)

The .BYTE/DFB/DFC directive is sometimes called a *pseudo-operation code,* or *pseudo-op,* because it appears in the op-code column of Assembly language source-code listing but is not actually a part of the 6510/8502 Assembly language instruction set. Instead, it is a specialized directive that varies in format from assembler to assembler. There are also many other pseudo-ops with formats that differ from one assembler to another. There are no generally accepted standards for writing pseudo-op directives, so pseudo-op codes designed for one assembler often will not work with another.

When the .BYTE directive (or one of its equivalents) is used in a program, the bytes that follow the directive are assembled into consecutive locations in RAM. In the program THE QUEST, the bytes that follow the label TEXT are ASCII codes for a series of text characters—as you can see when you run the program.

LOOPING THE LOOP

As explained in Chapter 6, the X and Y registers in the 6510/8502 chip can be progressively incremented and decremented during loops in a program. In the Quest program, the X register is incremented from 0 to 23 during a loop in which characters in a text string are read. The characters to be read are written as ASCII codes in Lines 110 through 130.

In Line 150, the statement LDX #0 is used to load the X register with a zero. Then, in Line 170, the loop begins.

INCREMENTING THE X REGISTER

The first statement in the loop is LDA TEXT,X. Each time the loop cycles, this statement uses indexed addressing to load the accumulator with an ASCII code for a text character. Then, in Line 180, the kernal routine CHROUT is used to print each character in the screen.

By the time the loop ends, all 23 characters in lines 110 through line 130 have been printed on the screen.

The first time the program hits Line 170, there will be a 0 in the X register (since a 0 has just been loaded into the X register). So the first time the program encounters the statement LDA TEXT,X, the accumulator is loaded with the hexadecimal number $54—what programmers sometimes call the 0th byte after the label TEXT. The hex number $54 is, of course, the ASCII code for the letter T. So, in Line 180, the kernal routine CHROUT takes the hex number $54 from the accumulator, recognizes it as the ASCII code for a T, and prints a T on the screen.

Incidentally, there is no need for #'s in front of the numbers in Line 110, since numbers that follow the .BYTE directive and its variants are automatically interpreted as literal numbers.

Now let's move on to line 190. The mnemonic you see there—INX—means "increment the X register."

The first time the program makes its way through the loop that starts at Line 170, the X register will hold a 0. But as soon as the CHROUT routine has printed its first character on the screen, the INX instruction in Line 190 will increment that 0 to a 1.

Next, in Line 200, we see the instruction CPX #BUFLEN. Look back at Line 60, and you will see that BUFLEN is a constant which has been equated to the number 23. So the instruction CPX #BUFLEN means "compare the value in the X register to the literal number 23."

The reason we want this comparison to be performed is so we can determine whether 23 characters have been printed on the screen yet. There are 23 characters in the text string that we are printing, and when we have printed all of them, we will want to print a carriage return and end our program.

COMPARING VALUES

There are three comparison instructions in 6510/8502 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 whether or not they are equal—and, if they are not equal, which one is larger than the other.

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, then the carry (C) flag of the P register will be left in a clear state.

If the value in the accumulator is greater than or equal to the tested value, then the Z flag will be set to 1 and the carry flag will also 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, respectively. They have the same effects that CMP has on the status flags of the P register.

USING COMPARISON AND BRANCHING INSTRUCTIONS TOGETHER

The three comparison instructions in Commodore Assembly language are usually used in conjunction with eight other Assembly language instructions—the eight conditional branching instructions that I mentioned in Chapter 6.

The sample program that I have called THE QUEST contains a conditional branching instruction in Line 210. 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."

This instruction uses what can be a confusing convention of the 6510/8502 chip. In the 6510/8502's 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* zero.

This is all quite academic, however, as far as the *result* of the statement BNE LOOP is concerned. When your computer encounters the BNE LOOP instruction in Line 210, it will keep branching back to Line 170 (the line labeled LOOP) as long as the value of the X register has not yet reached 23.

Once the value of the X register has reached 23, the statement BNE LOOP in Line 210 will be ignored, and the program will move on to the next line, and that line contains an RTS instruction that terminates the program.

CONDITIONAL BRANCHING INSTRUCTIONS

Now let's talk a little more about conditional branching instructions. As you may recall from Chapter 6, there are eight conditional branching instructions in 6510/8502 Assembly language. They all begin with the letter B, and they are also called relative-addressing, or *branching* instructions. These eight instructions, and their meanings, are:

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 is not 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.

HOW BRANCHING DIFFERS FROM JUMPING

In a few moments, we will use some of these instructions in another Assembly language program. First, though, this might be a good time to point out some important differences between the *branching instructions* in the preceding list and another category of 6502/6510/8502 instructions: *jump instructions,* which have been mentioned earlier in this book and which we will briefly review now.

There are two *jump instructions* in 6502 Assembly language: JMP and JSR. As you may recall from previous chapters, 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 Assembly language program, the memory address of the *next* instruction in the program is stored on the hardware stack. Then the program jumps to whatever memory address is specified by the operand that follows the JSR instruction.

The mnemonic JSR is designed primarily for use with subroutines. In 6502/6510/8502 Assembly language, subroutines almost always end with RTS instructions.

In Assembly language, RTS is the 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, then the address pulled from the stack will ordinarily be the one that was deposited there by the JSR instruction that was used to invoke the subroutine. Processing of the program will resume where it left off—at the line following the JSR instruction that was used to invoke the subroutine.

I have already described one difference between branching instructions and the jump instructions JMP and JSR: branching instructions are *conditional;* jump instructions are *unconditional.* When a jump instruction is encountered in a program, it will always be carried out, but when a branching instruction is encountered in a program, it will be carried out only if certain specific conditions are fulfilled.

OTHER DIFFERENCES

There is also another important difference between a jumping instruction and a branching instruction. In machine language, the operand that follows a jump instruction is always expressed as a *two-byte* value and is always interpreted as the starting address of the destination of the jump instruction. But when a branching instruction is assembled into machine language, the operand that follows the branching instruction, is always converted to a signed one-byte number. Then, when the program is executed, this signed one-byte number is interpreted as an *offset* that *points to* the starting address of the destination of the branch instruction.

This all sounds quite complicated, but some simple examples should make it clearer. Let's start with a sample statement containing a jump instruction:

```
JMP $C000
```

If the above 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 occur.

Unfortunately, branching instructions are a little more complicated than jump instructions. Here is a sample program that uses a branching instruction: BCC, which means "branch if carry set." I have named the program BRANCHIT.S, for obvious reasons.

```
THE BRANCHIT.S PROGRAM (SOURCE-CODE VERSION)

BRANCHIT.S

 1  ORG $8000
 2 *
 3 WHAZIS EQU $02A7
 4 *
 5  LDA #5
 6  CLC
 7  ADC WHAZIS
 8  BCS RETURN
 9  TAX
10 RETURN RTS
```

This little routine is very straightforward. In Line 5, the literal number 5 is loaded into the accumulator. Then the 6510/8502 carry flag is cleared, and the value stored in Memory Address $02A7 (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—that is, if the sum of 5 and WHAZIS is greater than 255—then the routine will branch to Line 10 and will end, but if the sum of 5 and WHAZIS does not result in a carry—that is, if the sum is less than 255—then the sum will be transferred to the X register before the routine ends.

ASSEMBLING THE BRANCHIT.S ROUTINE

Now let's take a look at an assembled listing of the BRANCHIT.S program:

```
THE BRANCHIT.S PROGRAM (ASSEMBLED VERSION)
8000:                 1           ORG   $8000
8000:                 2 *
8000:                 3 WHAZIS    EQU   $02A7
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
```

HOW BRANCHING INSTRUCTIONS WORK

Carefully examine Lines 8 through 10 of this assembled listing, and you will see how the branching instruction in the BRANCHIT.S program works. In Line 8, to the left of the line number, these figures appear:

```
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, and the third number—01—is an offset value that must be computed by your computer's 6510 or 8502 chip before it can carry out the BCS instruction.

OFFSET VALUES

What exactly *is* an offset value? Well, in a 6510/8502 branching instruction, an offset value is a signed number that must be added to a given memory address in order to compute 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, and that is the address of the RTS instruction that ends the BRANCHIT.S program.

Here are more details about how that works. 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. But then, when your program is assembled into machine language, your assembler will convert the literal address or label which you have used into an offset value. From then on, each time your computer's 6510 or 8502 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.

ANOTHER IMPORTANT POINT

Another fact that is important to remember is that an offset which follows a branching instruction can never be more than one byte long. Since this one byte is always interpreted by the 6510/8502 chip as a signed number, a branching offset can be no smaller than –128 and no larger than +127. 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 bytes from the current address. So branches which occur as the result of branching instructions are subject to certain length limitations; specifically, the destination address of a branching instruction cannot be more than 126 bytes lower than, or more than 129 bytes higher than, the address of the first instruction that *follows* the branching instruction.

What if a programmer wants to write an instruction that will branch to an address that does not fall within these limitations? Well, that is not too difficult. If you want to exceed the distance limitations of a branching instruction, all you have to do is use the instruction to branch to a *jumping* instruction, which has no such restrictions. Here is an example of how that can be done:

```
THE BRANCHIT.S PROGRAM
(WITH A JUMP INSTRUCTION ADDED)

 ORG $8000
*
WHAZIS EQU $02A7
*
 LDA #5
 CLC
 ADC WHAZIS
 BCC CONT
 JMP FARJMP ;(CAN BE ANYWHERE IN MEMORY)
CONT TAX
 RTS
```

SOMETHING FANCY

As you can see, a very neat trick has been used here to overcome the distance limitations of a branching instruction. In this version of the BRANCHIT.S program, the BCS instruction that appeared in the original program has been replaced by a BCC instruction, and 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 that has been labeled FARJMP, which 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

As you may have noticed from the programming examples provided so far in this chapter, the usual way to use a conditional branching instruction in 6510/8502 Assembly language is as follows. First, you load the A register (or a memory register) with a value to be used for a comparison, then do some type of comparison. Then you use a conditional branching instruction to tell your computer what P register flags to test, and what to do if these tests succeed or fail.

This all sounds very complicated, and, until you get the hang of it, it may be. However, once you understand the general concept of conditional branching, you can use a simple table for writing conditional branching instructions. Here is one such table:

Table 7-1. 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
Y < VALUE	CPY #VALUE	BCC
Y = (ADDR)	CPY $ADDR	BEQ

Y <> (ADDR)	CPY $ADDR	BNE
Y >= (ADDR)	CPY $ADDR	BCS
Y > (ADDR)	CPY $ADDR	BEQ and then BCS
Y < (ADDR)	CPY $ADDR	BCC

ASSEMBLY-LANGUAGE LOOPS

In 6510/8502 Assembly language, comparison instructions and conditional branch instructions are usually used together. In the sample program called THE QUEST, 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, and each time the program comes to Line 200, the value in the X register is compared to the literal number 23. When that number is reached, the loop ends.

The program will therefore keep looping back to Line 170 until 23 characters have been printed on the screen.

Got it? Good. Then we are ready to make some improvements in the program called THE QUEST. These improvements will make the program more versatile and more useful, and will even make it easier to understand.

```
IMPROVING THE QUEST PROGRAM

10 ;
20 ; RESPONSE
30 ;
40  *=$8000 ;OR 'ORG EQU $8000' (OR $2000)
50 ;
60  EOL=13 ;END-OF-LINE CHARACTER
70  BUFLEN=40 ;LENGTH OF TEXT BUFFER
80  FILLCH=$20 ;ASCII CODE FOR A SPACE
90  CHROUT=$FFD2
100 ;
110  JMP START
120 ;
130 TEXT .BYTE 'YOU CAN FIND HIM IN 64K',13
140 ;
150 ;CLEAR TEXT BUFFER
160 ;
170 START LDA #FILLCH
180  LDX #BUFLEN
190 STUFF DEX
200  STA TXTBUF,X
210  BNE STUFF
220 ;
230 ;STORE MESSAGE IN BUFFER
240 ;
```

```
250  LDX #0
260 LOOP1 LDA TEXT,X
270  STA TXTBUF,X
280  CMP #EOL
290  BEQ PRINT
300  INX
310  CPX #BUFLEN
320  BCC LOOP1
330 ;
340 ;PRINT MESSAGE
350 ;
360 PRINT LDX #0
370 LOOP2 LDA TXTBUF,X
380  PHA
390  JSR CHROUT
400  PLA
410  CMP #EOL
420  BNE NEXT
430  JMP FINI
440 NEXT INX
450  CPX #BUFLEN
460  BCC LOOP2
470 ;
480 FINI
490  RTS
500 ;
510 TXTBUF=*
520  *=*+BUFLEN
530 ;
540  .END
```

Appendix B contains another listing of the RESPONSE.S program, typed on a Merlin 64 assembler.

As you can see, the RESPONSE program is quite similar to THE QUEST. With a few modifications you can also type it, assemble it, and run it on a Panther assembler/editor system. If you have a Panther assembler, you will have to use a DFC directive in place of the .BYTE directive in the version of the program prepared on the Commodore assembler. If you are using the Merlin 64, you can replace the .BYTE directive with an ASC directive, and you will have to use the HEX directive to enter the number 13 (the ASCII code for a carriage return) that follows the text string in Line 130. For an example of how to do that, you can take a look at the listing for the NAME GAME program later in this chapter, which was written using a Merlin 64 assembler.

Now, if you like, you may type, assemble, and save the RESPONSE program. Then we will discuss the difference between THE QUEST and the RESPONSE program.

When you run the RESPONSE program, you will see that it performs essentially the same kind of operation that THE QUEST did, but it does it in a slightly different way. The most obvious difference between the two programs is the way they handle text strings. In the program called THE QUEST, we used a text string made up of ASCII codes. There is also a text string in RESPONSE, but it is made up of actual characters. Because of that difference, RESPONSE was a much easier program to write than THE QUEST and it is much easier to read, too.

Another important difference between RESPONSE and its predecessor is the way the loop that reads the characters is written. In THE QUEST, the loop counted the number of characters that had been printed on the screen, and ended when the count hit 23.

Now that is a perfectly good system—for printing text strings that are 23 characters long. Unfortunately, it is not so great for printing strings of other lengths. So it is not a very versatile routine for printing characters on a screen.

TESTING FOR A CARRIAGE RETURN

RESPONSE is more versatile than THE QUEST because it can print strings of almost any length on a screen. That is because the RESPONSE program does not keep track of the number of characters it has printed by maintaining a running count of how many letters have been printed. When the program encounters a character, it tests the character to see whether 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 prints it on the screen and goes on to the next character in the string. If the character is an EOL character, then the computer prints a carriage return on the screen and the routine ends.

TEXT BUFFER

Another difference between THE QUEST and RESPONSE is that the latter program does not read characters and print them on the screen 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 in many ways: from a keyboard, for example, or from a telephone modem, or even directly from a computer's memory. 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. So, 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, which will then be accessible with great ease and in many different ways.

Before you use a text buffer, though, it is always a good idea to clear it out; otherwise, it might be cluttered up with leftover characters. So a buffer-clearing routine has been written into the RESPONSE program. It is a short and simple routine, but it does the job. It will clear a text buffer—or any other block of memory that does not exceed its length limitations—and will stuff the buffer with spaces, zeros, or any other value you might choose. In the RESPONSE program, the routine stuffs the buffer with a string of spaces, which will appear as blank spaces on your computer screen.

As you continue to work with Assembly language, you will find that memory-clearing routines such as this one can come in very handy in many different 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 RESPONSE program is not very complicated. Using indirect 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 program ends.

The buffer-clearing routine in RESPONSE will work with any 8-bit fill character, and with any buffer length (BUFLEN) up to 255 characters. Later on in this book, you will find some 16-bit routines that can stuff values into longer blocks of RAM.

ONE MORE PROGRAM

The final program in this chapter will make use of many of the programming techniques we have learned so far. The program is called THE NAME GAME, and I wrote it in BASIC years ago. In fact, it was the first game program I ever wrote that worked. Later I wrote an Assembly language version of it for *Atari Roots* (Datamost, 1984), my book on Atari Assembly language. The current version was written on a Commodore 64 using a Merlin 64 assembler, but, if you own a Panther or Commodore 64 assembler, you should not have much trouble converting it into source code that is compatible with your assembler system. All of the programming conventions that THE NAME GAME contains have either been explained or will be before this chapter is finished, so if you own a Commodore or Panther assembler, you should be able to translate it into source code that will work on your assembler without too much difficulty.

```
*
* THE NAME GAME
*
 ORG $8000
*
EOL EQU $0D ;RETURN
EOF EQU $03 ;EOF CHR
FILLCH EQU $20 ;SPACE
BUFLEN EQU 40
CHRIN EQU $FFCF
CHROUT EQU $FFD2
TEMPTR EQU $FB
*
 JMP START
*
TXTBUF DS 40
```

```
*
TITLE ASC 'THE NAME GAME'
 HEX 0D
HELLO ASC 'HELLO, '
 HEX 03
QUERY ASC 'WHAT IS YOUR NAME?'
 HEX 0D
NAME ASC 'GEORGE'
 HEX 0D
REBUFF ASC 'GO AWAY, '
 HEX 03
DEMAND ASC 'BRING ME GEORGE!'
 HEX 0D
GREET ASC 'HI, GEORGE!'
 HEX 0D
*
* CLEAR TEXT BUFFER
*
FILL LDA #FILLCH
 LDX #BUFLEN
DOFILL DEX
 STA TXTBUF,X
 BNE DOFILL
 RTS
*
PRINT LDY #0
SHOW LDA (TEMPTR),Y
 CMP #EOF
 BEQ DONE
 PHA
 JSR CHROUT
 PLA
 CMP #EOL
 BNE NEXT
 JMP DONE
NEXT INY
 CPY #BUFLEN
 BCC SHOW
DONE RTS
*
* PRINT 'THE NAME GAME'
*
START LDA #EOL
 JSR CHROUT
 LDA #<TITLE
 STA TEMPTR
 LDA #>TITLE
 STA TEMPTR+1
 JSR PRINT
 LDA #EOL
 JSR CHROUT

*
* PRINT 'HELLO . . .'
*
 LDA #<HELLO
 STA TEMPTR
 LDA #>HELLO
 STA TEMPTR+1
 JSR PRINT
*
* PRINT 'WHAT IS YOUR NAME?'
*
ASK LDA #<QUERY
 STA TEMPTR
 LDA #>QUERY
 STA TEMPTR+1
 JSR PRINT
 LDA #EOL
 JSR CHROUT
*
* INPUT A TYPED LINE
*
 JSR FILL
 LDX #0
KEY JSR CHRIN
 STA TXTBUF,X
 CMP #EOL
 BEQ COMPARE
 INX
 JMP KEY
*
* IS THE NAME 'GEORGE'?
*
COMPARE JSR CHROUT ;PRINT RETURN
 LDX #0
CHECK LDA TXTBUF,X
 CMP NAME,X
 BNE NOGOOD
 CMP #EOL
 BEQ DUNIT
 INX
 CPX #BUFLEN
 BCS DUNIT
 JMP CHECK
*
```

```
113 * NO; PRINT 'GO AWAY . . .'
114 *
115 NOGOOD LDA #EOL
116  JSR CHROUT
117  LDA #<REBUFF
118  STA TEMPTR
119  LDA #>REBUFF
120  STA TEMPTR+1
121  JSR PRINT
122 *
123 * PRINT PLAYER'S NAME
124 *
125  LDA #<TXTBUF
126  STA TEMPTR
127  LDA #>TXTBUF
128  STA TEMPTR+1
129  JSR PRINT
130  LDA #EOL
131  JSR CHROUT
132 *
133 * PRINT 'BRING ME GEORGE!'
134 *
135  LDA #<DEMAND
136  STA TEMPTR
137  LDA #>DEMAND
138  STA TEMPTR+1
139  JSR PRINT
140  LDA #EOL
141  JSR CHROUT
142   JMP ASK
143 *
144 * YES; PRINT GREETING
145 *
146 DUNIT LDA #EOL
147  JSR CHROUT
148  LDA #<GREET
149  STA TEMPTR
150  LDA #>GREET
151  STA TEMPTR+1
152  JSR PRINT
153  RTS
```

PLAYING "THE NAME GAME"

If you have typed, assembled and executed the programs called THE QUEST and RESPONSE, you should not have much trouble understanding how this one works. Using several fairly simple subroutines, it prints a short message on your screen and then waits for you to type in a response. If you type a response which the program considers 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.

This little routine, as simple as it may seem, is actually an excellent introduction into the fascinating field of artificial intelligence. When you play THE NAME GAME, you actually do match wits with a computer—and, since computers are much more patient than most people are, THE NAME GAME is, in the long run, a game that the computer usually wins. Most people who play THE NAME GAME eventually give in and type the line that the computer has been waiting for. The computer therefore wins the game, and the game ends.

After you have played THE NAME GAME two or three times, it is quite likely that you will grow tired of it and will never care to play it again. However, the programming principles that were used to create the game could also be used in the development of commercial-quality entertainment and educational programs. With a little imagination and a dash of creativity, THE NAME GAME could serve as the basis for some very interesting programs.

HOW THE PROGRAM WORKS

In addition to the kernal routine CHROUT, which is often used in Commodore programs to print characters on a computer screen, THE NAME GAME makes use of another kernal routine—called CHRIN—which can read characters that are typed in on a computer keyboard. The call address of the CHRIN routine is $FFCF, and detailed instructions on how it is used can be found on page 277 of the *Commodore 64 Programmer's Reference Guide*.

CHRIN, CHROUT, and other constants are defined in Lines 6 through 12 of THE NAME GAME program. Then, in Line 14, there is a jump instruction that causes execution of the program to start at Line 59.

Before the program begins, some space is set aside for a text buffer (in Line 16), and the lines of text that are used in THE NAME GAME are listed as strings of data in Lines 18 through 31. Next there are two subroutines that will be used later on in the program. One, labeled FILL, will clear the text buffer whenever it is called. The other subroutine, called PRINT, uses the Commodore CHROUT routine to print messages on the screen.

As you type, assemble, and run THE NAME GAME, you may notice that it uses its text buffer for lines that are typed in at the keyboard, not for lines that are called from RAM. Some kind of buffer is obviously needed for typed-in lines, since the computer must hold them in its memory long enough to do some comparison and printing operations. Another text buffer could have been set up for the lines stored in RAM, but it would have accomplished no real purpose except to consume more memory and take up more processing time.

Another part of THE NAME GAME program that may be worth a special mention is the technique that is used for storing 16-bit numbers in high-order and low-order 8-bit memory locations. The technique first appears in Lines 72 through 75:

```
72  LDA #<HELLO
73  STA TEMPTR
74  LDA #>HELLO
75  STA TEMPTR+1
```

You can probably figure out what that sequence does without too much difficulty. In source code produced by the Merlin 64 assembler and the Commodore 64 assembler, 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'." (Not the *contents* of the address, by the way, but the address itself, since in 6502/6510/8502 Assembly language, the symbol # is used to identify a literal number.) So here is what the above sequence of code does. First it 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.

Now here is an important warning: Not all assemblers interpret the symbols < and > in the same way. In source code produced by the Panther C64 assembler, for example, #<HELLO would mean the *high* byte of the address of the line labeled HELLO, and #>HELLO would mean the *low* byte of the address of the line labeled HELLO—precisely the opposite of what these two strings mean in source-code programs written on the Merlin and Commodore 64 assemblers.

RUNNING THE PROGRAM

The main part of THE NAME GAME program starts at Line 57 with a routine that prints the program's title on the screen. The next two routines print the line "Hello, what is your name?"

After this question is asked, the program clears the text buffer and waits for the player of THE NAME GAME to type in a response. As the player types in an answer, each character that is typed is placed in the text buffer. That is all that happens until the player stops typing characters and enters a carriage return.

Once a carriage return is typed, the program examines the characters that have been stored in the text buffer to see whether they spell the name GEORGE. If the player has not typed in the name GEORGE, the computer prints "GO AWAY, [TYPED NAME], BRING ME GEORGE!" Then the game starts again. This process will continue until the player weakens and types the name GEORGE. Then the computer will print "HI, GEORGE," and the game will end.

8 Programming Bit by Bit

Single-Bit Operations on Binary Numbers

There are 65,536 bytes of memory in a Commodore 64 computer, and up to 524,288 bytes of memory in a Commodore 128. Since there are eight bits in every byte, this means that there are 524,288 *bits* in a Commodore 64, and up to 4,194,304 bits in a Commodore 128. If you know how to perform single-bit operations on binary numbers, you can control every binary bit in your Commodore automatically. That is a tremendous amount of control to have over a computer—and you can wield that kind of control if you are proficient in Assembly language.

In an earlier chapter of this book, you learned how to control one of the most important bits in your Commodore's central microprocessor, the carry bit of the 6510/8502 processor status register. Manipulating the P register's carry bit is one of the most important bit-manipulation techniques in 6510/8502 Assembly language, and you have already had considerable experience in using the carry bit in addition programs.

In this chapter, you will have an opportunity to teach your computer how to perform some new tricks using the carry bit of its 6510/8502 processor status (P) register.

USING THE CARRY BIT

As I have pointed out a number of times now, your Commodore's 6510/8502 microprocessor is an 8-bit chip; it cannot perform operations on numbers larger than 255 without putting them through some fairly tortuous contortions.

In order to process numbers that are larger than 255, the 6510/8502 must split them up into 8-bit chunks, and then perform the requested operations on each chunk of a number. Then each number that has been split must be put back together and made whole again.

Once you are familiar with how this is done, it is not nearly as difficult as it sounds. In fact the electronic scissors that are used in all of this electronic cutting and pasting are actually contained in one tiny bit—the carry bit in the 6510/8502's processor status register.

FOUR BIT-SHIFTING INSTRUCTIONS

You have seen how how carry operations work in several programs in this book, but in order to get a clearer look at how the carry works in 6510/8502 arithmetic, 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 extensively in 6510/8502 Assembly language. We will look at them one a time, starting with the ASL (Arithmetic Shift Left) instruction.

ASL (ARITHMETIC SHIFT LEFT)

As you may recall from the chapter on binary arithmetic, every round number in binary notation is equal to twice the preceding round binary number. In other words, 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), and so on.

It is extremely easy to multiply a binary number by 2. All you have to do is shift every bit in the number left one space, and place a zero in the bit that has been emptied by this shift—Bit 0, or the rightmost bit of the number.

If Bit 7 (the leftmost bit) of the number to be doubled is a 1, then provision must be made for a carry.

The entire operation we have just described—shifting a byte left, with a carry—can be performed by a single instruction in 6510/8502 Assembly language. That instruction is ASL, which stands for "arithmetic shift left."

Figure 8-1 shows how the ASL instruction works.

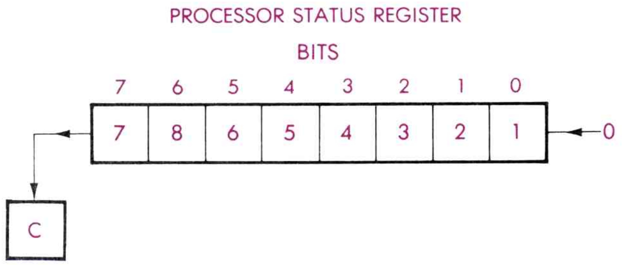

Figure 8-1 The ASL (Arithmetic Shift Left) Instruction

As you can see from this illustration, 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 is used for many purposes in 6510/8502 Assembly language. It is often used, for instance, as an easy way of multiplying numbers by two. Here is what a number-doubling routine might look like in a program created using a Commodore 64 assembler:

```
10 ;
20  *=$8000
30 ;
40  LDA #$40 ;REM 0100 0000
50  ASL A ;SHIFT VALUE IN ACCUMULATOR TO LEFT
60  STA $FB
70  .END
```

If you run this little program, and then use your assembler's machine-language monitor to examine the contents of Memory Address $FB, you will 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, and thus increase a computer's effective memory capacity. Later in this chapter, there will be an example of how to pack data using the ASL instruction.

LSR (LOGICAL SHIFT RIGHT)

The instruction LSR ("logical shift right") is the exact opposite of the instruction ASL—as you can see from Figure 8-2.

LSR, like ASL, works on whatever binary number is in the 6510/8502's accumulator, but 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 in with a zero, and the LSB will be dumped into the carry flag of the P register.

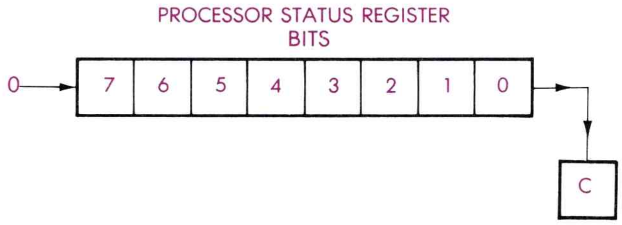

Figure 8-2 The LSR (Logical Shift Right) Instruction

The LSR instruction can be used to divide any even 8-bit number by 2, as follows:

```
10 ;
20 ;DIVIDING BY 2 USING LSR
30 ;
```

```
 40  VALUE1=$FB
 50  VALUE2=$FC
 60 ;
 70  *=$8000
 80 ;
 90  LDA #6 ;OR ANY OTHER 8-BIT NUMBER
100  STA VALUE1
110 ;
120 ;NOW WE'LL DIVIDE BY 2
130 ;
140  LDA VALUE1
150  LSR A
160  STA VALUE2
170  .END
```

As you can see, this routine divides the number stored in VALUE1 by 2, and stores the result of that calculation in VALUE2. As a side benefit, it can also tell you whether the number it has divided is odd or even. It leaves that bit of information (no pun intended) in the carry bit of the 6510/8502 P register. If the routine leaves the carry bit set, the number that was just divided is odd. If the carry bit is clear, the value is even.

Here is a program you can type, execute, and check using your assembler's machine-language monitor to see whether a number is even or odd. It is not very useful, but it does illustrate a point. In Lines 120 and 130 of this routine, a memory register called FLGADR (for "flag address") is cleared to zero. Then the contents of another memory register, called VALUE1, are shifted to the right one position and stored in a third register, called VALUE2. If the value being shifted is even, then the shift operation does not set the carry bit, and the subroutine ends. If this operation does set the carry bit, the program jumps to Line 220, and the carry bit, now set, is rotated into the register called FLGADR using an instruction called ROL (which you will learn more about in just a moment). So if the routine leaves a 0 in FLGADR, then the number that was divided is even, but if the routine ends with a 1 stored in FLGADR, then the number that was divided is odd.

```
 10 ;
 20 ;ODDTEST
 30 ;
 40  VALUE1=$FB
 50  VALUE2=$FC
 60  FLGADR=$FD
 70 ;
 80  *=$8000
 90 ;
100  LDA #7 ;(ODD)
110  STA VALUE1
120  LDA #0
130  STA FLGADR ;CLEARING FLGADR
```

```
140 ;
150  LDA VALUE1
160  LSR A ;PERFORM THE DIVISION
170  STA VALUE2 ;DONE
180 ;
190  BCS FLAG
200  RTS ;END ROUTINE IF CARRY CLEAR . . .
210 ;
220 FLAG
240  ROL FLGADR
250  RTS ;. . . AND END THE PROGRAM
```

As previously mentioned, you can also use LSR to unpack data that has been packed using ASL, but to unpack data, you also have to use another type of Assembly language instruction called a logical operator. We will discuss logical operators (and look at some sample routines for packing and unpacking data) later in this chapter.

Meanwhile, let's examine two more bit-shifting operators—ROL and ROR.

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 in a totally different manner. Figure 8-3 shows how the ROL instruction works.

ROL, like ASL, can be used to shift the contents the accumulator or a memory register one place to the left, but ROL does *not* place a zero 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, then a 1 is placed in the Bit 0 position of the byte being shifted. If the carry bit is clear, then a zero goes into the Bit 0 position of the shifted register.

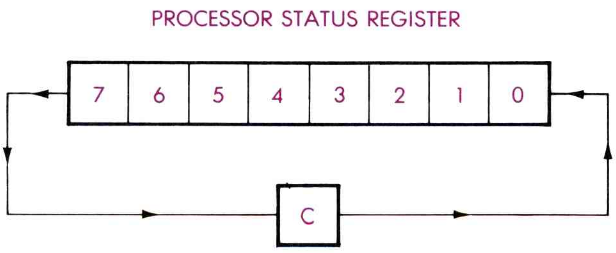

Figure 8-3 The ROL (Rotate Left) Instruction

ROR works just like ROL, but in the opposite direction (see Figure 8-4). 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. Meanwhile, as part of the same rotation process, Bit 0 of the shifted byte is moved into the carry bit of the processor status register.

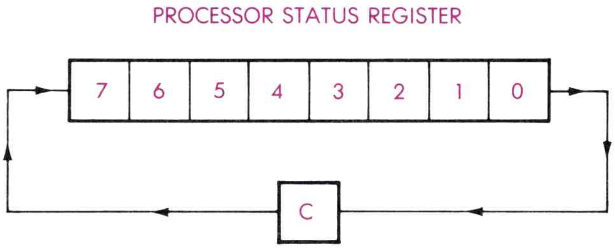

Figure 8-4 The ROR (Rotate Right) Instruction

ROL and ROR are often used in 6502/6510/8502 multiplication and division routines, and in many other types of routines in which bits are shifted and tested.

THE LOGICAL OPERATORS

Before we move on to conventional binary arithmetic, let's take a brief glance at four important Assembly language mnemonics called *logical operators*. These instructions are AND ("and"), ORA ("or"), EOR ("exclusive or"), and BIT ("bit").

The four 6510/8502 logical operators look very mysterious at first glance, but, in typical Assembly language fashion, they lose much of their mystery once you understand how they work.

AND, ORA, EOR and BIT are all used to compare values, but they work differently from the comparison operators CMP, CPX and CPY. The instructions CMP, CPX, and CPY all yield very general results. All they can determine is whether two values are equal—and, if the values are not equal, which one is larger than the other.

AND, ORA, EOR, and BIT are much more specific instructions. They are used to compare single bits of numbers, and hence have all sorts of uses.

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 proved true, its value in Boolean logic is said to be 1. If it is false, its value is said to be 0.

THE AND OPERATOR

In 6510/8502 Assembly language, the operator AND has the same meaning that the word "and" has in English.

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, but 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.* Here is a truth table for the AND operator:

TRUTH TABLE FOR AND

```
      0         0         1         1
  AND 0     AND 1     AND 0     AND 1
  -----     -----     -----     -----
      0         0         0         1
```

In 6510/8502 Assembly language, the AND instruction is often used in an operation called bit masking. The purpose of bit masking is to clear specific bits of a number. The AND operator can be used, for example, to clear any number of bits by placing a zero in each bit that is to be cleared.

This is how that kind of bit-masking operation could work:

```
100  LDA #AA ;BINARY 1010 1010
110  AND #F0 ;BINARY 1111 0000
```

If your computer encountered this routine in a program, the following AND operation would take place:

```
    1010 1010   (contents of accumulator)
AND 1111 0000
-------------
    1010 0000   (new value in accumulator)
```

As you can see, this operation would clear the low nibble of $AA to $0 (with a result of $A0), and the same technique would work with any other 8-bit number. No matter what the number being passed through the mask 1111 0000 might be, its lower nibble would always be cleared to $00—and its upper nibble would 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. This is the truth table for ORA:

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
```

Suppose that the number in VALUE were $22 (binary 0010 0010). This is the masking operation that 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 a pair being tested is a 1.

This is the truth table for the EOR operator:

TRUTH TABLE FOR EOR

0	0	1	1
EOR 0	EOR 1	EOR 0	EOR 1
0	1	1	0

The EOR instruction is often used for comparing bytes to determine if they are identical, since if any bit in two bytes is different, the result of a comparison will be non-zero. Here is an illustration:

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 zero. In Example 2, one bit is different, so the result of the comparison is non-zero.

The EOR operator is also use to *complement* values. If an 8-bit value is EOR'd with $FF, every bit in it that is a 1 will be complemented to a 0, and every bit that is a 0 will be complemented to a 1—like this:

```
    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 restored to its original value. For example:

```
    1110 0101   (in accumulator)
EOR 0101 0011
    1011 0110   (new value in 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 put one image over another without destroying the one underneath.

PACKING DATA IN MEMORY

Now we are 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-byte 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 kinds of 4-bit values.

Using the ASL instruction, you could pack two such values into every byte of the block of memory in which they were stored. You could thus store the values in half the memory space that they had previously occupied in their unpacked form.

Here is a routine you could use in a loop to pack each byte of data:

```
10 ;
20 ;PACKDATA
30 ;
40  *=$8000
50 ;
60  NYB1=$FB
70  NYB2=$FC
```

```
 80  PKDBYT=$FD
 90 ;
100  LDA #$04 ;OR ANY OTHER 4-BIT VALUE
110  STA NYB1
120  LDA #$06 ;OR ANY OTHER 4-BIT VALUE
130  STA NYB2
140 ;
150  CLC
160  LDA NYB1
170  ASL A
180  ASL A
190  ASL A
200  ASL A
210  ORA NYB2
220  STA PKDBYT
230  RTS
```

This routine will load a 4-bit value into the accumulator, will shift that value to the high nibble in the accumulator, and will then—using the ORA logical operator—place another 4-bit value in the low nibble 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 your computer, and you can then execute it using your assembler's machine-language monitor. Then, when you have run the program, you can use your monitor to peek into your computer's memory to see exactly what has been done. If you are using a Commodore or Panther assembler, you can check the results of your work by typing the monitor command M (for "memory display.") If you are a Merlin owner, you can use the H (for "hex dump") command. Your computer will then respond with a line that looks something like this:

```
.:00FB 04 06 46 00 00 00 00 00
```

or like this:

```
00FB-04 06 46 00 00 00
```

What you can tell from either of these lines is that your program has stored the number $04 in Memory Address $FB, and has stored the number $06 in Memory Address $FC. Then both of these values have been packed into Memory Address $FC. It does not take much imagination to see how this technique can increase your computer's capacity to store 4-bit numbers—or ASCII characters, which can be stored in memory in the form of 4-bit numbers. By packing data, you can actually double the text-storage capacity of an 8-bit computer, since you can store two characters in each 8-bit register in the computer's memory.

UNPACKING DATA IN MEMORY

It would not do any good to pack data, of course, if it could not later be *unpacked*. It so happens that data packed using ASL can be unpacked using the complementary instruction LSR ("logical shift right"), together with the logical operator AND. Here is a sample routine for unpacking data:

```
 10 ;
 20 ;UNPACKIT
 30 ;
 40  PKDBYT=$FB
 50  LOWBYT=$FC
 60  HIBYT=$FD
 70 ;
 80  *=$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
```

This routine works much like the previous one, but in reverse. First, 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 nibble of the byte is stored into a memory register called LOWBYT.

After that is all done, 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. The result of this maneuver is 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

That brings us to the BIT operator, an instruction that is 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 BIT instruction can be used only with absolute or zero-page addressing. Here are two examples of correct formats for the BIT instruction:

```
BIT $02A7
```

or:

```
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, then the Z flag will be cleared. If there are no set bits that match, the Z flag will be set.

Here is a sample routine in which the BIT instruction is used:

```
1   LDA #01
2   BIT $02A7
3   BNE MATCH
4   JMP NOGOOD
5 MATCH RTS
```

In this routine, a check is made to determine whether BIT 0 is set in the value stored in Memory Register $02A7. If the bit is set, the zero (Z) 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 processor status register. That can be a very useful thing to know, since Bit 6 and Bit 7 are very important flags in the 6510/8502 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 and easy method for checking either Bit 6 or Bit 7 of any 8-bit value. If Bit 6 of the value being tested is set, then the P register's V flag will also wind up being 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 wind up being set, and a BPL or BMI instruction can be used to determine the outcome of the routine.

It is also important to note that after all of these actions take place, the value in the accumulator (and the memory location being tested) always remains unchanged. So 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.

9 Assembly Language Math

Addition, Subtraction, Multiplication, and Division

In Chapters 1 through 8, we have been doing a lot of reading, and a lot of writing, and now it is time to do some arithmetic. In this chapter, you will learn how your Commodore adds, subtracts, multiplies, and divides.

As you may have noticed by now, your Commodore 64 or 128 can deal with many kinds of numbers—including binary, decimal, hexadecimal, signed, and unsigned numbers. Other kinds of numbers that your computer can deal with include binary coded decimal numbers, floating-point decimal numbers, signed and unsigned numbers, and possibly a few other varieties of numbers. In this chapter, we are going to take at least cursory looks at each of these types of numbers—and maybe a few other kinds, too.

To understand how your computer works with numbers, it is essential to have a fairly good understanding of the busiest flag in the 6510/8502 microprocessor chip: the carry flag of the 6510/8502's processor status register. So let's take a good close look at the 6510/8502's carry flag right now.

THE CARRY BIT

The best way to get a closeup view of how the carry bit works is to examine it through an "electronic microscope" at the bit level. Look at these two simple 4-bit hexadecimal and binary addition problems and you will see clearly how neither addition operation generates a carry in either binary or hexadecimal notation.

```
HEXADECIMAL        BINARY

      $04            0100
    + $01          + 0001
    -----          ------
      $05            0101
      $08           1000
    + $03         + 0011
    -----         ------
      $0B           1011
```

Now let's look at a couple of problems that use larger (8-bit) numbers. The first of these two problems does not generate a carry, but the second one does.

```
HEXADECIMAL          BINARY

      $8E          1000 1110
    + $23        + 0010 0011
    -----        -----------
      $B1          1011 0001

      $8D          1000 1101
    + $FF        + 1111 1111
    -----        -----------
    $018C      (1) 1000 1100
```

Note that the sum in the second problem is a 9-bit number: 1 1000 1100 in binary and 18C in hexadecimal.

Here is an Assembly language program that will perform that very addition problem. Type it into your computer and run it, and you will be able to see how the carry flag in your computer works:

8-BIT ADDITION WITH A CARRY

```
10 ;ADDNCARRY
20  *=$8000
30  CLD
40  CLC
50  LDA #$8D
60  ADC #$FF
70  STA $FB
80  RTS
```

When you have typed this program, assemble it and then run it using your assembler's machine-language monitor. When the program has been executed, use your monitor to take a look at the contents of Memory location $FB. Just type an M or H instruction (depending on what kind of assembler you have), and your computer should respond with a line that looks something like this:

```
00FB  8C 00 00 00 00 00
```

That line shows us that Memory Address $FB now holds the number $8C. That is not the sum of the numbers $8D and $FF, but it is close. In hexadecimal arithmetic, the sum of $8D and $FF is $18C—exactly the sum we got, plus a carry.

So where is the carry?

Well, if everything you have learned about the carry bit so far is true, then our missing carry must be tucked away just where it is supposed to be: namely, in the carry bit of your computer's processor status register. So let's go there and look for it, right now.

A BIT IN A HAYSTACK?

Looking for a carry bit inside a Commodore may seem like looking for a needle in a haystack, but a carry bit really is not too hard to find, once you know where to look for it. For example, here is one way to locate the carry that is missing from the ADDNCARRY program. All it takes is the insertion of a few additional lines into the ADDNCARRY program. Here is an expanded version of the program, with those extra lines inserted:

```
ADDITION WITH A CARRY (IMPROVED VERSION)

 10 ;ADDNCARRY2
 20  *=$8000
 30  CLD
 40  CLC
 50  LDA #$8D
 60  ADC #$FF
 70  STA $FB
 80  LDA #0
 90  ROL A
100  STA $FC
110  RTS
```

In the lines we have now added to this program, the accumulator is cleared, and the bit-shifting operator ROL is then used to rotate the P register's carry bit into the accumulator. The contents of the accumulator are then deposited into Memory Register $FC using an ordinary STA instruction. If this routine works, it means that we have found our missing carry bit.

The best way I can think of to see whether the program works is to type it, assemble it, and run it. Once you have done that, you can peek into Memory Addresses $FB and $FC using your machine-language monitor and see whether the calculation in the ADDNCARRY program resulted in a carry.

So let's do it. Assemble the program, execute it, and then use your monitor to take a look at the contents of Memory Address $FB and the seven memory locations that follow. You should then see this line (or a variation thereof, depending upon the kind of assembler you are using):

```
00FB-8C 01 00 00 00 00
```

This line tells you two things: 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.

16-BIT ADDITION

We will now take a look at a program that 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.

Here is the program:

```
A 16-BIT ADDITION PROGRAM

10 ;
20 ;ADD16
30 ;
40 ;THIS PROGRAM ADDS A 16-BIT NUMBER IN $FB AND $FC
50 ;TO A 16-BIT NUMBER IN $FD AND $FE
60 ;AND DEPOSITS THE RESULTS IN $02A7 AND $02A8
70 ;
80  *=$8000
90 ;
100  CLD
110  CLC
120  LDA $FB;REM LOW HALF OF 16-BIT NUMBER IN $FB AND
    $FC
130  ADC $FD;REM LOW HALF OF 16-BIT NUMBER IN $FD AND
    $FE
140  STA $02A7 ;LOW BYTE OF SUM
150  LDA $FC ;REM HIGH HALF OF 16-BIT NUMBER IN $FB
    AND $FC
160  ADC $FE ;REM HIGH HALF OF 16-BIT NUMBER IN $FD
    AND $FE
170  STA $02A8 ;HIGH BYTE OF SUM
180  RTS
```

When you look at this program, remember that your Commodore computer stores 16-bit numbers in the reverse order from what you might expect—with the low-order byte first, and the high-order byte second. Once you understand that fluke—a characteristic of all 6502/6510/8502-based computers—then 16-bit binary addition is not hard to comprehend.

In this program, we first clear the carry flag of the P register. Then 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 $02A7. 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, then 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 $02A8. Then, finally, the results of our completed addition problem are stored—low byte first—in Memory Addresses $02A7 and $02A8.

16-BIT SUBTRACTION

Here is a 16-bit subtraction program:

```
10 ;
20 ;SUB16
25 ;
30 ;THIS PROGRAM SUBTRACTS A 16-BIT NUMBER IN $FB AND
    $FC
40 ;FROM A 16-BIT NUMBER IN $FD AND $FE
50 ;AND DEPOSITS THE RESULTS IN $02A7 AND $02A8
60 ;
70  *=$8000
80 ;
90  CLD
100  SEC ;REM SET CARRY
110  LDA $FD;REM LOW HALF OF 16-BIT NUMBER IN $FD AND
    $FE
120  SBC $FB;REM LOW HALF OF 16-BIT NUMBER IN $FB AND
    $FC
130  STA $02A7 ;LOW BYTE OF THE ANSWER
140  LDA $FE ;REM HIGH HALF OF 16-BIT NUMBER IN $FD
    AND $FE
150  SBC $FC ;REM HIGH HALF OF 16-BIT NUMBER IN $FB
    AND $FC
160  STA $02A8 ;HIGH BYTE OF THE ANSWER
170  RTS
```

Since subtraction is the exact opposite of addition, the carry flag is set, not cleared, before a subtraction operation is performed in 6510/8502 binary arithmetic. In subtraction, the carry flag is treated as a borrow, not a carry, and it must therefore be set so that if a borrow is necessary, there will be a value to borrow from.

After the carry bit is set, a 6510/8502 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 our subtraction problem—including a borrow from the high byte, if one was necessary—is then stored in Memory Addresses $02A7 and $02A8, in the low-byte-first convention that is typical in 6502/6510/8502-based computers.

BINARY MULTIPLICATION

Binary numbers are multiplied in the same way that decimal numbers are. Here is an example:

```
      0110      ($06)
    × 0101      ($05)
    ------
      0110
     0000
    0110
   0000
   -------
   0011110      ($1E)
```

Unfortunately, however, there are no 6510/8502 Assembly language instructions for multiplication or division. To multiply a pair of numbers using 6510/8502 Assembly language, you have to perform a series of addition operations. To divide numbers, you have to perform subtraction sequences.

Look closely at the multiplication problem presented above, however, and you will see that it is not difficult to split a multiplication problem into a series of addition problems. In the example given above, the binary number 0110 is first multiplied by 1. Then the result of this operation—also 0110—is written down.

WHAT HAPPENS NEXT

Next, 0110 is multiplied by 0. The result of that operation —a string of zeros—is also shifted one space to the left and written down. Then 0110 is multiplied by 1 again, and the result is once again shifted left and written down. Finally, another multiplication by zero results in another string of zeros, which are also shifted left and duly noted.

Once that is done, all of 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 fine, but it is really quite arbitrary. 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 6510/8502 multiplication, that is exactly what is often done; instead of shifting each partial product to the left before storing it in memory, many 6510/8502 multiplication algorithms shift the preceding partial product to the left before adding it to the new one.

Here is a program that will show you how that works:

```
10 ;
20 ;MULT16
30 ;
40  MPR=$FD ;MULTIPLIER
```

```
50  MPD1=$FE ;MULTIPLICAND
60  MPD2=$02A7 ;NEW  MULTIPLICAND AFTER 8 SHIFTS
70  PRODL=$02A8 ;LOW BYTE OF PRODUCT
80  PRODH=$02A9 ;HIGH BYTE OF PRODUCT
90 ;
100  *=$8000
110 ;
120 ;THESE ARE THE NUMBERS WE WILL MULTIPLY
130 ;
140  LDA #250
150 STA MPR
160  LDA #2
170  STA MPD1
180 ;
190 MULT CLD
200  CLC
210  LDA #0 ;CLEAR ACCUMULATOR
220  STA MPD2 ;CLEAR ADDRESS FOR SHIFTED MULTIPLICAND
230  STA PRODH ;CLEAR HIGH BYTE OF PRODUCT ADDRESS
240  STA PRODL ;CLEAR LOW BYTE OF PRODUCT ADDRESS
250  LDX #8 ;WE WILL USE THE X REGISTER AS A COUNTER
260 LOOP LSR MPR ;SHIFT MULTIPLIER RIGHT; LSB DROPS
    INTO CARRY BIT
270  BCC NOADD ;TEST CARRY BIT; IF ZERO, BRANCH TO
    NOADD
280  LDA PRODH
290  CLC
300  ADC MPD1 ;ADD HIGH BYTE OF PRODUCT TO
    MULTIPLICAND
310  STA PRODH ;RESULT IS NEW HIGH BYTE OF PRODUCT
320  LDA PRODL ;LOAD ACCUMULATOR WITH LOW BYTE OF
    PRODUCT
330  ADC MPD2 ;ADD HIGH PART OF MULTIPLICAND
340  STA PRODL ;RESULT IS NEW LOW BYTE OF PRODUCT
350 NOADD ASL MPD1 ;SHIFT MULTIPLICAND LEFT; BIT 7
    DROPS INTO CARRY
360  ROL MPD2 ;ROTATE CARRY BIT INTO BIT 7 OF MPD1
370  DEX ;DECREMENT CONTENTS OF X REGISTER
380  BNE LOOP ;IF RESULT ISN'T ZERO, JUMP BACK TO
    LOOP
390  RTS
400  .END
```

QUITE A PROBLEM

As you can see, 8-bit binary multiplication is not exactly a snap. There is a lot of left and right bit-shifting involved, and it is hard to keep track of. In the above program, 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 is 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 out on your keyboard and assemble it. Then use the G command of your monitor to execute it. Then, while you are still in the monitor mode, you can take a look at the contents of Memory Addresses $02A8 and $02A9. These two registers should now hold the number $01F4 (low byte first, remember). That is the hex equivalent of the decimal number 500, and that, of course, is product of the number 2 and the decimal number 250, which our program was supposed to multiply.

AN IMPROVED MULTIPLICATION PROGRAM

Although the program you have just tried works fine (provided you have done it right), it is not the only 16-bit multiplication program around; in fact, it is not 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. The following program, for example, is considerably shorter, and therefore is both more memory-efficient and faster-running. One of the neatest tricks in this improved multiplication program is that it uses the 6510/8502's accumulator, rather than a memory address, for temporary storage of the problem's results.

```
10 ;
20 ;MULT16B
30 ;(AN IMPROVED 16-BIT MULTIPLICATION PROGRAM)
40 ;
50  PRODL=$FD
60  PRODH=$FE
70  MPR=$02A7
80  MPD=$02A8
90 ;
100  *=$8000
110 ;
120 VALUES LDA #10
130  STA MPR
140  LDA #10
150  STA MPD
160 ;
170  LDA #0
180  STA PRODH
190  LDX #8
200 LOOP LSR MPR
210  BCC NOADD
```

```
220  CLC
230  ADC MPD
240 NOADD ROR A
250  ROR PRODH
260  DEX
270  BNE LOOP
280  STA PRODL
290  RTS
```

You may want to test out this improved multiplication program the same way we tested the previous one: by executing it using your machine-language monitor, and then using your monitor to take a look at its results.

You can play around 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 those two tools of our forefathers, a pencil and a piece of paper. Work enough binary multiplication problems on paper, and you will soon begin to understand the principles of 6510/8502 multiplication.

MULTIPRECISION BINARY DIVISION

Earlier in this chapter, we discovered that subtraction is reverse addition. Now it is time to point out that division is nothing but reverse multiplication. This being the case, it would be logical to assume that the 6510/8502 chip, which has no specific instructions for multiplying numbers, would also have no instructions for dividing numbers, and, unfortunately, that is true.

Still, it is possible to perform division—even multiple-precision long division—using instructions that are available to the 6510/8502 microprocessor. As we have seen, the 6510/8502 chip can multiply numbers, provided that the multiplication problems presented to it are broken down into sequences of addition problems. So you probably will not be surprised to learn that the 6510/8502 chip can also divide numbers—provided that the division problems presented to it are broken down into sequences of subtraction problems. The next sample program you see in this chapter will be a routine designed to divide one number into another number by breaking the division process down into a series of subtraction routines.

During the execution of this division program, 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 380, the remainder will be moved out of the accumulator and into a variable called RMDR. Finally, an RTS instruction will end the program.

```
A BINARY LONG-DIVISION PROGRAM

10 ;
20 ;DIV8/16
30 ;
40  *=$8000
50 ;
60 DVDH=$FD ;LOW PART OF DIVIDEND
70 DVDL=$FE ;HIGH PART OF DIVIDEND
80 QUOT=$02A7 ;QUOTIENT
90 DIVS=$02A8 ;DIVISOR
100 RMDR=$02A9 ;REMAINDER
110 ;
120  LDA #$1C ;JUST A SAMPLE VALUE
130  STA DVDL
140  LDA #$02 ;THE DIVIDEND IS NOW $021C
150  STA DVDH
160  LDA #$05 ;ANOTHER SAMPLE VALUE
170  STA DIVS ;WE'RE DIVIDING BY 5
180  ;
190  LDA DVDH ;ACCUMULATOR WILL HOLD DVDH
200  LDX #08 ;FOR AN 8-BIT DIVISOR
210  SEC
220  SBC DIVS
230 DLOOP PHP ;SAVE P REGISTER (ROL & ASL AFFECT IT)
240  ROL QUOT
250  ASL DVDL
260  ROL A
270  PLP ;RESTORE P REGISTER
280  BCC ADDIT
290  SBC DIVS
300  JMP NEXT
310 ADDIT ADC DIVS
320 NEXT DEX
330  BNE DLOOP
340  BCS FINI
350  ADC DIVS
360  CLC
370 FINI ROL QUOT
380  STA RMDR
390  RTS
```

The above program 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 $02A7, and the remainder, if any, is stored in Memory Register $02A9.

Type the program, assemble and run it, and then use your monitor to inspect the contents of Memory Addresses $02A7 and $02A9. Address $02A7 should now hold the hexadecimal number $6C (108 in decimal notation), and there should be a zero in address $02A9, since the quotient of 540 divided by 5 is 108, with no remainder.

NOT THE ULTIMATE DIVISION PROGRAM

As you can see, it is even more difficult to write a division routine for a Commodore than it is to write a Commodore 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 is worth. When you have to write a program in which just a few calculations have to be made, you can sometimes get away with using short, simple routines such as the ones presented in this chapter, but 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 of the math—in BASIC. That way, you can take advantage of the excellent floating-point math package that is built right into the BASIC interpreter in your Commodore. If you cannot do that, it might still be best to write the program in some language—almost any language—besides Assembly language. Because of the extraordinary amount of work that it takes to write mathematical routines for computers in the Commodore class, it is usually much better to write complex mathematically oriented programs in BASIC, Pascal, COBOL, Logo, or almost any other high-level programming language than it is to try to write them in Assembly language.

If, despite this warning, you *still* have a yen to write complex math routines in 6510/8502 Assembly languages, there are a few books that may provide you with some help. One text that contains an abundance of fairly complex math routines that are yours for the typing is 6502 *Assembly Language Subroutines,* 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

Before we move on to the next chapter, there are three more topics that we should briefly cover: signed numbers, binary-coded decimal (BCD) numbers, and floating-point numbers. First we will talk about signed numbers.

To represent a signed number in binary arithmetic, all you have to do is let the leftmost bit (Bit 7) represent a positive or negative sign.

In signed binary arithmetic, if Bit 7 of a number is zero, the number is positive, but 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.

MORE THAN A BIT OF WORK

It should also be obvious that 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)
```

That answer is wrong. The answer should be +1.

The reason we arrived at 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.

A REASONABLE ASSUMPTION

It might be 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 for the purpose of representing the number's sign).

This technique—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 is 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)
```

Oops. That is wrong, too. The answer should be plus 3.

Well, that takes us back to the drawing board. One's complement arithmetic does not work.

However, there is another technique, which comes very close to one's complement, that does work. It is called *two's complement,* and it works as follows.

First calculate the one's complement of a positive number. Then simply *add one,* and 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—and, if you do not make any mistakes, they will work every time.

Here is how:

```
    0000 0101 (+5)
  + 1111 1000 (-8) (two's complement)
  -----------
    1111 1101 (-3)
```

Another two's complement addition problem:

```
    1111 1011 (-5) (two's complement)
  + 0000 1000 (+8)
  -----------
    0000 0011 (+3) (plus carry)
```

A FEW EXAMPLES

As I said, it works every time. It is not easy to explain why, but when you have worked with signed binary numbers for a while, you begin to get a feel for them. It helps to remember this: since the highest bit of a binary number is always interpreted as a sign in two's complement notation, a binary number with the highest bit set is always interpreted as a negative number. So 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. Increment the hex number $7F, and you will see why this is true. In binary notation, $7F is written %0111 1111—a binary number in which the high bit is not set. But increment $7F, and what you will get is $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. So 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 backwards, just as conventional negative numbers do in ordinary arithmetic. In conventional arithmetic, there is no such number as –0, so when you decrement a 0, what you get is not –0, but –1. Decrement –1, and what you get is –2, which may look at first glance like a larger number, but is really a smaller one. Decrement –2, and you get –3, and so on.

Two's complement arithmetic works in a similar fashion. Decrement 0 using 8-bit two's complement arithmetic, and, since there is no such number as –0, what you will get is $FF, which equates to –1 in decimal notation. Decrement $FF in two's complement, 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, and so on.

Keep working backwards, and you will 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 do not quite understand all of that, please do not panic. It will probably be quite a while before you will have any need to write any programs that make use of signed binary numbers, and if ever do start writing such programs, you will need instructions that are much more detailed than the ones you will find in this book. In this chapter, my intention is not to make you an expert in signed binary numbers, but merely to introduce you to some of the techniques that are often used in programs that contain signed binary numbers. Then, when you run across a program that makes use of signed binary numbers, you will at least know something about what is going on.

USING THE OVERFLOW FLAG

Before we move on to another topic, there is one more fact that I would like to point out about 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 for this is as follows. The carry flag of the P register is set when there is 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. So the carry flag cannot be used to detect a carry in an operation that involves signed numbers.

You can solve this problem by using the overflow bit of the processor status register. The overflow bit is set when there is an overflow from Bit 6, not bit 7. So it can be used as a carry bit in arithmetic operations on signed numbers.

As you may recall from high-school algebra and beyond, 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. So it should come as no surprise 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. So let's leave the topic of signed binary numbers, and move on to our next subject: BCD (binary-coded decimal) numbers.

BCD 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 must therefore 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, is 100% accurate. BCD numbers are also sometimes used when it is desirable to print them out 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, or chaos can result. You must also decide whether you want to use an 8-bit byte for each digit, which wastes memory since it really only takes 4 bits to encode a BCD digit, or pack two digits into each byte, which saves memory but consumes processing time.

FLOATING-POINT NUMBERS

Now we have come to a topic that strikes fear into the heart of even expert Assembly language programmers: 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 is built into your Commodore. In the Commodore's floating-point package, numbers are broken down into three-parts—an exponent, a mantissa and a sign—and are stored in a block of memory called a floating-point accumulator. This accumulator resides in Memory Registers $61 to $66, and there is another one just like it in Memory Registers $69 through $6E.

YOUR COMPUTER'S FLOATING-POINT PACKAGE

Unfortunately for Assembly language programmers, it is extremely difficult to understand how floating-point routines work, and it is even more difficult to write them. So it is nice to know that there is a very good floating-point package built right into your Commodore, and that whenever you decide you need it, it is right there. To use the Commodore 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 back in Chapter 5. When you write a program in this fashion, you can use Assembly language for the portions of the program in which high speed or high performance are required; you can use BASIC to access your computer's built-in floating-point package for portions of the program that require high-precision math.

Since this package exists, and is so easy to use, you may never need to know most the programming techniques that have been described in this chapter. However, an understanding of how they work will definitely make you a better Commodore Assembly language programmer.

Knowing how to write complex Assembly language math routines is a little like knowing how to perform first aid; you may never have to use your knowledge, but if you do, it will certainly come in handy. Furthermore, you have to know at least the fundamentals of 6510/8502 arithmetic if you want to become a good Commodore Assembly language programmer. After all, mathematical processing, in one form or another, is what computer programming is all about. Since your Commodore adds, subtracts, compares, and bit-shifts its way through every program it processes, it would be difficult to imagine writing Commodore 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 will 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.

Assembly Language Graphics and Sound

10 Memory Magic
The Memory Map of the Commodore

The engineers who designed the Commodore 64 accomplished quite a feat: they stuffed 88K of memory into a 64K machine. The Commodore 128 has an even more prodigious memory; it has 128 kilobytes of memory built in and can be expanded into a 512K computer with the addition of an optional *RAM disk* module.

Since this book is designed to be used with either a Commodore 64 or a Commodore 128, the extra memory that is built into the C128 will not be discussed in this chapter. Instead, we focus our attention on the 64K of memory that is common to both machines. If you own a Commodore 64, everything in this chapter is applicable to your computer, and, since the Commodore 128 has a Commodore 64 built in, what you read in this chapter is also applicable to the Commodore 128, as long as the machine is operating in its C64 mode.

MAPPING THE MEMORY OF THE COMMODORE 64

From a memory-organization standpoint, the Commodore 64 is a rare breed of computer. Most 64K computers have only 48K or so of addressable RAM, plus around 16K of ROM, for a total of 64K. But the Commodore 64 has a full 64K of *user-addressable RAM,* plus 24K of built-in ROM: 88K of built-in memory.

This 88K of memory is controlled by a pair of special memory registers that occupy Memory Addresses $0000 and $0001. These two registers are all that distinguish the 6510 chip used in the Commodore 64 from its predecessor, the 6502, which is used in computers manufactured by Apple, Atari, and several other companies.

A PROGRAMMER'S DELIGHT

With the help of these two memory registers, at RAM Addresses $0000 and $0001, a skilled programmer can wield a tremendous amount of control over the features and capabilities of the Commodore 64. By simply switching certain bits in this pair of registers on and off, a programmer can actually determine what portions of the computer's memory will be used as RAM and what blocks will be used as ROM.

If you wanted to, you could switch off every byte of ROM in the C64, opening up every bit of the the computer's memory for use as RAM. So, if you were a good enough programmer, you could theoretically turn the Commodore 64 into a totally customized computer, with an operating system and a set of input/output drivers of your own design.

We are not going to get nearly that ambitious, but we are going to discuss how large blocks of RAM and ROM can be juggled around inside the Commodore 64, adapting it to the individual needs of the C64 programmer.

Once you know how to manipulate Memory Registers $0000 and $0001, you can perform some pretty fancy tricks. For example, you can switch off the Commodore 64's BASIC interpreter and use the memory space it consumes as additional RAM. You can copy the C64's built-in character set from ROM into RAM, and then modify it and use it in programs however you choose. With the additional assistance of some other special memory registers, you can even use some blocks of the C64's memory capacity as *both* RAM and ROM—at the same time.

READING THE MEMORY MAP

Before we start discussing how Registers $0000 and $0001 work, let's take a look at a memory map of the Commodore 64. Figure 10-1 is a simplified map that shows its default configuration—that is, what its memory configuration looks like when its power is first turned on. This simple map is only a general guide to the C64's memory; a much more detailed memory map can be found on pages 310 through 334 of the *Commodore 64 Programmer's Reference Guide*.

Most of the labels on this map have not yet been covered in this book. So let's take a look now at what they mean.

ADDRESSES $0000 THROUGH $0FFF

(Page-Zero RAM)

Page Zero, the block of memory that extends from $0000 to $00FF, is the "high-rent district" in your computer's RAM. Memory space on Page Zero is so desirable that the engineers who designed the Commodore 64 used most of it for the computer's operating system and BASIC interpreter. Consequently, very little Page Zero space is available for user-written programs.

This shortage of space on Page Zero can create tough problems for the Assembly language programmer. It restricts the use of Page-Zero addressing, which could make programs run faster, and it also makes it difficult to use indirect indexed addressing, which will not work at all unless space on Page Zero is available. To write good Assembly language programs, therefore, it is

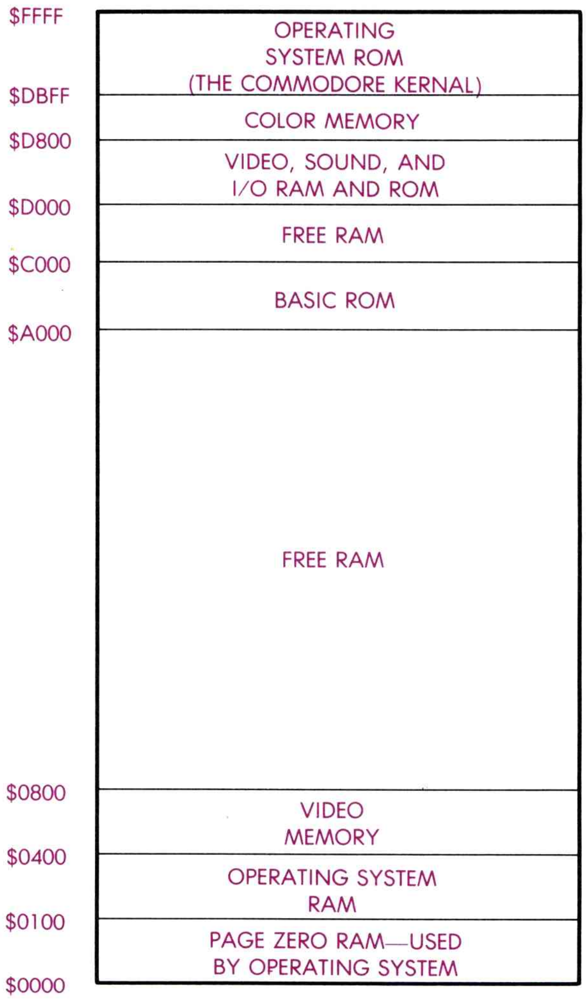

Figure 10-1 A Simplified Memory Map of the Commodore 64 (Not to Scale)

absolutely essential to find at least a few free memory locations on Page Zero. If you look carefully, that is how many free memory locations you will find on Page Zero: a few.

A MAP OF THE HIGH-RENT DISTRICT

MEMORY ADDRESSES	DESCRIPTIONS OF REGISTERS' FUNCTIONS
$00–$01	Special 6510 I/O control registers (more about these later)
$0002	Not used
$03–$FA	Registers used by BASIC and by C64 operating system
$FB–$FE	Bytes left free for user-written programs
$00FF	Used by BASIC interpreter

Specifically, these are the memory addresses on Page Zero that you can and cannot use in your programs, with the conditions under which they are and are not available.

$FB, $FC, $FD, AND $FE

As you can see from the memory map, there are only four bytes on Page Zero, $FB, $FC, $FD, and $FE, that are permanently reserved for use by user-written programs. But a good programmer can sometimes find other Page Zero addresses that can be safely used in Assembly language programs. For example, many of the addresses on Page Zero are reserved for use by the Commodore 64 BASIC interpreter, and a number of others are used only by the floating-point math routines that are built into the C64's operating system. In Assembly language programs that are not designed to be called from BASIC and so do not require the use of the C64's floating-point math package, many of these registers are free for use by user-written programs. (A complete listing of Page-Zero memory addresses, with brief annotations that can help you select which ones you might be able to use in your programs, can be found on the memory map that starts on page 310 of the *Commodore 64 Programmer's Reference Guide.*)

ADDRESSES $0100 THROUGH $01FF

(The Hardware Stack)

Page One of the Commodore 64's memory, the RAM space that extends from $0100 through $01FF, is reserved for use by the computer's hardware stack. The hardware stack is a section of memory that the 6510 processor uses to keep track of the return addresses of machine-language subroutines and *interrupts* (temporary interruptions in normal program processing). The stack is also used for temporary storage of the contents of memory registers. It is heavily used by the Commodore 64's operating system and is also available for user-written programs.

ADDRESSES $0200 THROUGH $03FF

(OS RAM and Free RAM)

Most of the RAM space that extends from $0200 through $03FF is reserved for use by the Commodore 64 operating system, but there is one small block of memory in this area that is usually free for use in user-written programs. This free block, 88 bytes long, extends from Memory Register $02A7 through Memory Register $02FF. This segment of memory is too short to do much programming in, but it is often ideal for data tables that are used in Assembly language programs.

If you own a Merlin 64 assembler, however, a word of caution about the $02A7-$02FF memory block is in order. The Merlin 64 uses some of this memory for its own purposes, so programs that are written with the Merlin and use the $02A7–$02FF area for data storage may not always work properly when they are executed *using the Merlin editor or monitor*. Such programs usually wind up working fine, however, when they are removed from their Merlin environment and executed *without* the help of the Merlin assembler/editor/monitor package, using BASIC and/or DOS commands instead.

ADDRESSES $0400 THROUGH $07FF

(Video Memory RAM in Text Mode, Color RAM in Bit-Mapped Mode)

At power-up time, Memory Registers $0400 through $07FF are designated as video RAM, and are used to store the memory map which the Commodore 64 uses to generate its screen display. When the C64 is in its high-resolution (bit-mapped mode), however, this segment of memory is far too short to hold a complete screen map. (A low-resolution *screen map* consumes only 1,000 bytes of memory, while a high-resolution screen map requires 8,000 bytes.) When a programmer wants to use a high-resolution screen display, therefore, it is the programmer's responsibility to find a segment of memory that does contain enough space for a screen map, and to set up a high-resolution screen at that location. Programs illustrating exactly how this is done will be presented in the two chapters that follow this one.

Since the $0400–$07FF memory block is not used for high-resolution screen mapping, it is generally used for another purpose when the C64 is in its bit-map mode: to determine what *colors* will be used for the computer's screen display.

ADDRESSES $0800 THROUGH $9FFF

(Free RAM)

The 38K block of memory that extends from $0800 through $9FFF is totally free RAM (RAM that is permanently set aside for use by user-written or user-purchased programs). When you write a program in Assembly language, or in BASIC, or any other language, this is the block of memory in which it will usually be stored.

At first glance, this looks like quite a large chunk of memory, but the more closely you examine it, the smaller it gets. When you are writing an Assembly language program, for example, your assembler, editor, and monitor usually consume quite a bit of the memory space in this segment of memory. When you write a program that requires both BASIC and Assembly language, that can cramp your style even more, since you will then have to take special precautions to keep the BASIC and machine-language portions of your program from running into each other. So good memory management is always important in Assembly language programming, even in a block of memory this big. (Later on in this chapter, you will be provided with some tips on how to steer clear of RAM that is used by assemblers and BASIC programs.)

ADDRESSES $A000 THROUGH $BFFF

(RAM or BASIC ROM)

When you turn on the Commodore 64, Memory Addresses $A000 through $BFFF are usually occupied by BASIC ROM, your computer's built-in BASIC interpreter. If you do not need a BASIC interpreter, you can switch this block of memory from ROM to RAM, and can thus add 8K of user-accessible RAM to your computer's memory. Explanations of how to do this will be provided later in this chapter and the two chapters that follow.

ADDRESSES $C000 THROUGH $CFFF

(Free RAM—With Some Exceptions)

In Memory Locations $C000 through $CFFF, there is another 4K of RAM that is usually free for use by user-written programs, but it is not always available to the Assembly language programmer. In fact, all three of the assemblers that were used to write the programs in this book make use of this block of memory in one way or another. This does not mean that you cannot use this memory block in Assembly language programs; there are ways to make use of it, if you are a clever enough Assembly language programmer. A map showing how your assembler uses the $C000-$CFFF memory block can help you in this kind of effort, and memory maps of all three assemblers that were used to write the programs in this book can be found at the end of this chapter.

ADDRESSES $D000 THROUGH $D800

(I/O RAM and Character ROM)

Memory Registers $D000 through $D800 serve double duty in the Commodore 64. With the help of some sophisticated bank-switching techniques, these addresses are used as both RAM and ROM by the C64's operating system. When they are used as RAM registers, their main job is to help control I/O devices. When they are used in their ROM mode, they are used to hold the data which the C64 uses to print characters on its video monitor. The techniques that are used to switch this block of addresses back and forth between RAM and ROM will be explained later on in this chapter.

ADDRESSES $D800 THROUGH $DBFF

(Color Memory in Text Mode; Otherwise, Free RAM)

When the Commodore 64 is in its text mode, Memory Addresses $D800 through $DBFF are used for the storage of color data, the data that determines the colors of characters displayed on the computer's video screen. When the C64 is in its high-resolution mode, this block of RAM is not used for the storage of color data and is free for use by user-written programs.

ADDRESSES $DBFF through $FFFF

(Kernal ROM or Free RAM)

This block of memory is almost always occupied by the Commodore 64 kernal, a block of machine-language input/output routines that are extensively used by the C64 operating system and are also available for use in user-written programs. If you really wanted to, you could switch the C64's kernal ROM out of this area, and use it as free RAM, but if you did that, you would have to write your own operating system.

A complete list of kernal routines, and detailed instructions for using them, can be found in Chapter 5 of the *Commodore 64 Programmer's Reference Guide*. That is a good chapter to become familiar with since the Commodore kernal is an extremely valuable resource for the Commodore 64 Assembly language programmer.

Now that you know how the Commodore 64's memory is organized, we are ready to take a look at Memory Registers $0000 and $0001, which control all memory-management operations of the C64.

$0000: THE DATA DIRECTION REGISTER

Memory Register $0000 is the 6510 chip's *data direction register*. In Assembly language programs, it is often labeled D6510. The D6510 register is used to control the direction of the flow of data into and out of certain blocks of memory, and also to control the direction of data flow to and from the Commodore 64 data cassette recorder.

$0001: THE I/O PORT

Memory Register $0001 is the Commodore 64's *input/output port,* or *control port.* In Assembly language programs, it is often labeled R6510. The chief function of the R6510 register is to determine which blocks of memory will be used as RAM and which blocks will be used as ROM.

The R6510 register and the D6510 have eight bits each, but only five of the bits in each register are significant. There is a direct one-to-one correspondence between the remaining five bits in each register. Each significant bit of the data direction register controls the direction of the data flow, which is controlled by the corresponding bit of the I/O control register.

Bits 6, 7, and 8 of the D6510 and the R6510 registers are the bits that are not significant. Bits 3 through 5 of each register are used to control a data cassette recorder, if one is connected to the Commodore 64. Bits 0, 1 and 2 of each register are used to determine whether specific blocks of the C64's memory will be used as ROM or RAM, and what kinds of data will appear in the blocks being used as RAM.

BIT	5	4	3	2	1	0
	1	0	1	1	1	1

Figure 10-2 The Significant Bits of the D6510 Register

THE D6510 REGISTER

When the Commodore 64 is powered up, the five significant bits of the D6510 register are set as shown in Figure 10-2.

There is rarely any need to change these bit settings in a user-written program. The bits in the Commodore 64's other "magic register"—Register R6510—are reset much more often.

THE R6510 REGISTER

The R6510 register, like the D6510, has eight bits, five of which are significant. When an R6510 bit is set, the function that it controls becomes an output function. When a bit is cleared, the function that it controls becomes an input function.

Bits 6 through 8 of the R6510 register, like the corresponding bits of the D6510 register, are not significant. Bits 3 through 5, like the same bits of the D6510 register, are used to control the Commodore 64 data cassette recorder. That leaves only three bits—Bits 0 through 2—for memory-control purposes, but these are three of the most powerful bits on the Commodore 64's memory map.

To illustrate, here is a table showing the five significant bits of the R6510 and D6510 registers, along with their functions.

TABLE 10-1. THE R6510 AND D6510 REGISTERS

BIT	NAME	SETTING AT POWER-UP	FUNCTION
0	LORAM	1 (Output)	On: $A000–$BFFF is BASIC ROM Off: $A000–$BFFF is RAM
1	HIRAM	1 (Output)	On: $E000–$FFFF is kernal ROM Off: $E000–$FFFF is RAM
2	CHAREN	1 (Output)	On: $D000–$DFFF is I/O ROM Off: $D000–$DFFF is character ROM
3		1 (Output)	On: Write to cassette line Off: Read from cassette line
4		0 (Input)	On: Cassette switch pressed Off: Cassette switch not pressed
5		1 (Output)	On: Cassette motor on Off: Cassette motor off

BIT 0: LORAM

As you can see from Table 10-1, Bit 0 of the R6510 register (the bit called LORAM) controls whether Memory Addresses $A000 through $BFFF will be used as BASIC ROM or as user-addressable RAM. If Bit 0 is set, then the Commodore 64's built-in BASIC interpreter will consume Memory Registers $A000 through $BFFF, and can be used for writing and running BASIC programs. If Bit 0 of the R6510 register is cleared, then Memory Addresses $A000 through $BFFF can be used as free RAM, and the computer's built-in BASIC interpreter will not be available for use in writing or running BASIC programs.

BIT 1: HIRAM

Bit 1 of the R6510 register (the bit called HIRAM) controls whether the Commodore 64 kernal will occupy Memory Registers $E000 through $FFFF, or whether those registers will be made available for use as free RAM.

BIT 2: CHAREN

Bit 2 of the R6510 register (the CHAREN bit) is the "magic bit" that determines whether Memory Addresses $D000 through $DFFF will be used as RAM registers by the Commodore 64's operating system, or will be used as character generator ROM. When Bit 2 of the R6510 register is set, the $D000-through-$DFFF block of memory is used as RAM by the C64's operating system, primarily the portion of the operating system that controls the operation of I/O devices. When the CHAREN bit is clear, all RAM stored in the $D000-$DFFF area becomes temporarily inaccessible, and 4K of character-generator ROM—better known to most Commodore users as the C64's built-in character set—is switched in. That sort of electric hocus-pocus is known in computer circles as bank-switching. It is a technique that is often used to stretch the memory capacity of a computer beyond the machine's nominal limits. Here is how it is used in the Commodore 64:

SOME MEMORY MAGIC

As you may recall from Chapter 1, the Commodore 64's screen graphics are produced by a sophisticated microprocessor called the *Video Interface Chip*, or *VIC-II*. To generate the characters that it displays on the Commodore 64's video screen, the VIC-II chip uses a 4K character set that is permanently stored in ROM addresses $D000 through $DFFF, but the VIC-II does not need access to this data all the time. Under ordinary conditions, it has to refer to the data only 60 times each second (a snail's pace to a computer), during a brief period of time known as a *video refresh cycle*. A video refresh cycle, as you may know, is a split-second "blackout" period during which a computer's screen goes blank between video screen displays, and it is only during this lightning-fast flash of time that the VIC-II chip has any need to access the ROM data stored in Memory Registers $D000 to $DFFF. The rest of the time—which is, of course, most of the time—access to the ROM data in this segment of memory is not ordinarily required. So, to keep this rather large hunk of memory occupied when it is not being used to generate video characters, the ROM data that it contains is temporarily bank-switched right out of the Commodore 64's memory, and a 4K block of RAM is switched in, and all of this happens so rapidly, 60 times every second, that the whole process is totally transparent to the Commodore 64 user.

A MINOR PROBLEM AND HOW TO HANDLE IT

This bank-switching technique is an ingenious method for conserving memory, but it sometimes creates a minor problem for Commodore 64 programmers. Fortunately, this problem is not very difficult to solve, provided one knows how.

Here is the problem. Occasionally, a programmer needs access to the Commodore 64's character-generator ROM for a slightly longer period of time than the split-second refresh period between video frames provides. For example, sometimes a programmer needs to copy the Commodore 64's character set from ROM to RAM, so that it can be altered and then accessed in its new form. (Yes, the C64's character set *can* be modified, and the ability to create customized character sets is one of the most powerful graphics techniques available to the Commodore 64 programmer. You will learn how to create customized character sets in the next two chapters.)

To copy the C64's character set from ROM to RAM, it is usually necessary to have access to the ROM data in $D000–$DFFF for a longer period of time than the computer's video refresh cycle provides. This is where Bit 2 of the R6510 register comes in handy. To prevent the C64's character set from being switched off while it is being copied, all a programmer has to do is turn off the CHAREN bit while the data-duplication process is taking place. While the CHAREN bit is off, the ROM data in the $D000-$DFFF ROM bank can be safely copied into RAM. Then, when the copying process is finished, CHAREN can be turned back on, and Memory Registers $D000 through $DFFF can be accessed once again by the C64's operating system. A program that shows exactly how this process is carried out will be presented in Chapter 11.

FOUR KINDS OF GRAPHICS DATA

We have now covered four types of graphics data that are used by the Commodore 64. They are:

1. *Character-generator ROM* (the character set that is permanently stored in ROM Registers $D000 through $DFFF). As we have mentioned, this character set can be moved into RAM, where it can be modified and then used in its altered form in user-written programs. In all, the C64's character set can be stored in any one of 32 blocks of memory. A table showing where each of these blocks is situated will be presented at the end of this chapter.

2. *Screen memory* (the C64 screen map). When you turn on the Commodore 64, the computer's screen memory map extends from Memory

Address $0400 to Memory Address $0800. But, by storing certain values in the upper four bits of a certain memory register (specifically, Memory Address $D018), you can place the Commodore's screen map in many other locations. You will also be provided with a table showing all of these locations before you finish this chapter.

3. *Sprite data.* Sprites, as you may know, are user-definable graphics characters that can be easily animated, completely independently of any other graphics that may appear on a computer screen. The VIC-II chip can display up to eight sprites on a screen at once, and the data used to create sprites may be placed anywhere in the Commodore 64's memory. But when sprites are used, the VIC-II must be provided with eight pointers telling it where it must look to find the data that it needs to draw each sprite on the screen. And these eight pointers are always situated in the same place; they are the last eight bytes of the 1K block of RAM used as screen memory.

4. A block of *color memory RAM* that extends from $D800 to $DBE7. This block of memory cannot be moved. When it is used (it is not needed when the C64 is in bit-mapped mode), it always occupies the 1,000 bytes of RAM beginning at Memory Address $D800.

THE VIC-II CHIP AND ITS FOUR MEMORY BANKS

To understand how the Commodore 64 handles all of these types of data, it is essential to understand one important concept about the C64's VIC-II graphics chip.

The VIC-II chip can control up to 16 kilobytes of RAM, but that entire 16K must be situated in one in block of memory called a *memory bank.* The Commodore 64 has only four of these memory banks; Figure 10-3 shows their positions.

We have covered all of the blocks of memory that are identified on this map, except for the two blocks labeled "character memory (ROM image)" that appear in Banks 0 and 2. Here is an explanation of what those blocks are, and how they are used.

As I have mentioned several times now, the Commodore 64's character-generator ROM, better known as the C64's built-in character set, is permanently stored in Memory Addresses $D000 through $DFFF. But, since the VIC-II chip can "see" only 16K of memory at a time, this block of ROM is not always accessible to the VIC-II. So the engineers who designed your computer decided to provide the VIC-II chip with two so-called "ROM images" of the C64 character set, one at Memory Locations $1000 to $2000, and the other in Memory Locations $9000 to $A000.

ROM IMAGES

A *ROM image address* is a kind of phantom memory address—a location that has no meaning at all, except to the VIC-II chip. Since the block of memory in

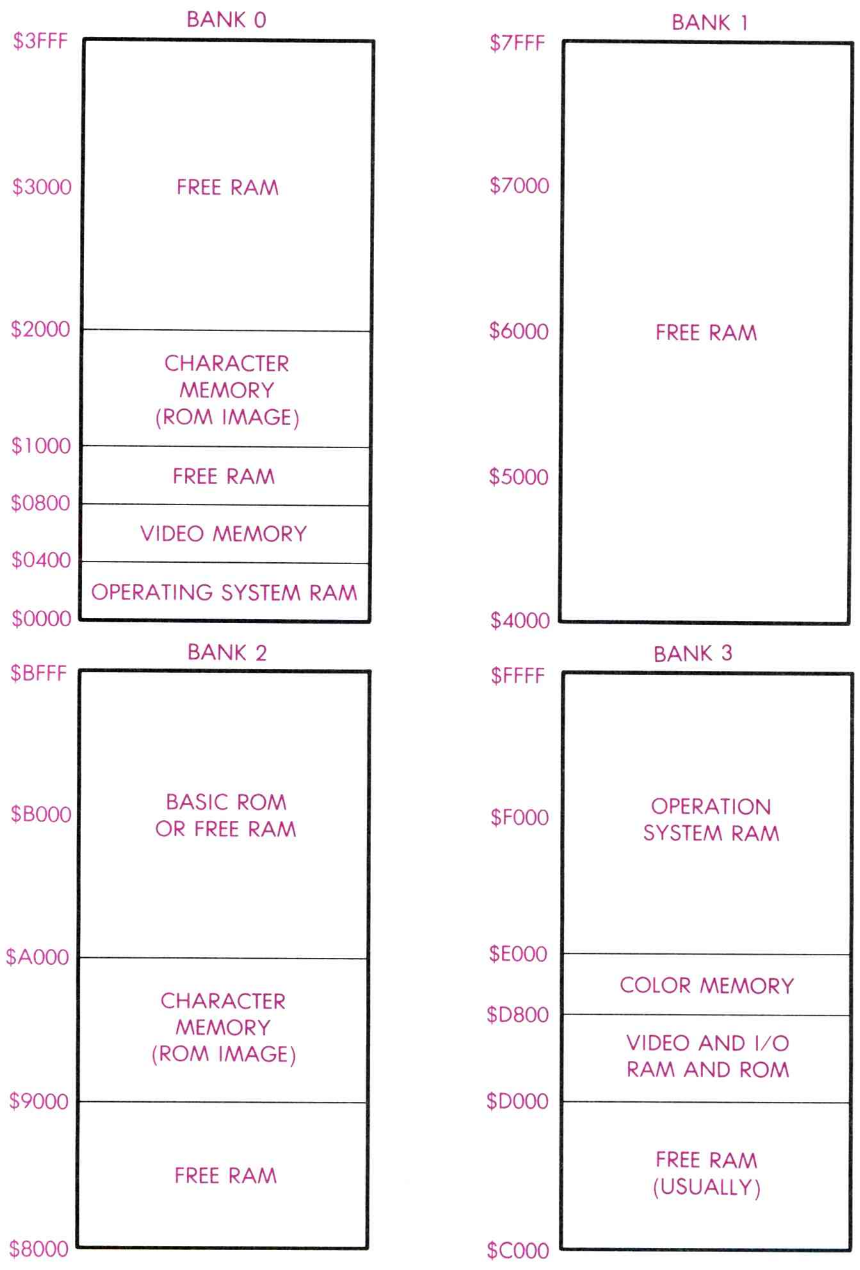

Figure 10-3 A Four-Bank Memory Map

which the C64's character ROM actually resides is not always accessible to the VIC-II chip, the engineers who built the VIC chip purposely crossed the wires in its direction-finding circuitry, fooling it into thinking that the C64 character set is in a different place from where it actually is. So, although you and I know that the C64's character set is in the $D000–$DFFF memory block, the VIC-II chip does not know that. Instead, like a desert traveler looking at a mirage, VIC has been designed to look at a certain expanse of RAM and see an "image" of character ROM that is not really there.

Actually, there are two such ROM images on the C64 map: one at Memory Address $1000, and the other at Memory Address $9000. As you can see from the four-bank map, the ROM images at $1000 and $9000 occupy the same respective positions in Memory Banks 0 and 2. So the VIC-II chip sees the ROM image at $1000 when it is looking at Bank 0, and sees the one at $9000 when it is looking at Bank 2.

In reality—and this is the most difficult part of this concept to understand—both of these memory blocks contain just ordinary RAM, and this RAM can be used just like any other RAM for non-graphics purposes in an Assembly language program. But, since the VIC-II chip sees character data when it looks at these two particular segments of RAM, neither of them can be used for any other graphics purposes. If you try to store any kind of graphics data under either of these "ROM images," the VIC-II chip will not be able to see it; VIC will always see only "ROM image" character data in these two blocks of memory.

BANKS 1 AND 3

If you are still with me, you might now be asking another question: When the VIC-II chip is looking at Bank 1 or Bank 3, which contain no ROM images, then where does VIC find the character data that it needs to generate its screen displays? The answer is nowhere. When the VIC-II chip has been set to look at Bank 1 or Bank 3, it simply will not be able to find any character data at all—unless the character ROM that actually resides at $D000 to $DFFF has been copied into RAM, and VIC has been told where in RAM to find it. When you have copied a character set from ROM to RAM, you can tell VIC where the new character set can be found by setting a certain series of bits (specifically, Bits 1 through 3) of a certain memory register (specifically, Memory Register $D018). Once that has been done, the VIC-II chip can find the duplicate character set in its new location.

If you do not quite understand all of that yet, please do not be too concerned; it will become clearer in the next two chapters, when sample programs will be presented illustrating how the VIC-II chip finds and uses the character data stored in the Commodore 64's ROM.

Meanwhile, let's move on to our next topic: how to use the four memory banks that are accessible to the VIC-II chip.

HOW TO SELECT A MEMORY BANK—AND WHY

When the Commodore 64 is first powered up, the memory bank that it uses for storage of graphics data is Bank 0. As you can see from the four-bank memory map presented in Figure 10-3, Bank 0 is a pretty crowded place; so crowded, in fact, that it is not always the best spot in which to store graphics data. Some programs require the use of a high-resolution screen, which consumes eight times as much of memory as a text screen, and some programs even call for two or more high-resolution screen displays. In addition, there are programs that use more than one character set, and programs that use large amounts of sprite data. So a program that includes a lot of graphics can easily exceed the space limitations of Bank 0.

Fortunately, when a program requires more room for graphics than Bank 0 offers, it is not too difficult to switch the Commodore 64's graphics memory area from one memory bank to another. All you have to do to make such a switch is stuff a certain two-code number into Bits 0 and 1 of a memory register situated at Memory Address $DD00. A couple of sample programs that are designed to perform this operation are presented in the next two chapters of this book. In case you cannot wait that long, Table 10-2 lists the codes that must be stored in Bits 0 and 1 of Memory Register $DD00 to determine what memory bank will be used for the storage of graphics data.

Table 10-2 DESIGNATING A MEMORY BANK

TO SELECT BANK NO.	AT THESE ADDRESSES	STORE THESE VALUES IN BITS 0 AND 1 OF MEMORY REGISTER $DD00 (Binary)	(Decimal/Hexadecimal)
0	$0000–$3FFF	00	00
1	$4000–$7FFF	01	01
2	$8000–$BFFF	10	02
3	$C000–$FFFF	11	03

AN IMPORTANT WARNING

There is one important note of caution regarding the use of Table 10-2. Memory Register $DD00 is a multi-purpose register that controls various input/output functions of the 6510 chip, as well as designating the memory bank that is to be used by the VIC-II chip. So, when you load a value from the table into Bits 0 and 1 of Register $DD00, it is important not to disturb the contents of the other bits in the register. To alter Bits 0 and 1 without disturbing Bits 2 through 7, you can use an Assembly language routine such as this:

```
LDA $DD00
AND #$FC ;CLEAR BITS 0 AND 1
ORA #$02 ;A VALUE FROM THE ABOVE CHART
STA $DD00
```

YOUR NEXT DECISION

Once you have decided what memory bank you want to use for graphics data, you can also decide exactly where in that block you want to put the various kinds of data that your computer's VIC-II chip must address: your screen map(s), your character set(s), and your sprite data (if you will be using any sprites in your program). In the next section of this chapter, you will find detailed examples of how to make all of these decisions—and how to carry them out once they have been made.

MOVING A CHARACTER SET

There are actually two character sets in the Commodore 64. One, which contains uppercase letters and graphics characters, begins at Memory Address $D000. The other set includes upper- and lowercase letters but no graphics characters. It starts at $D800. (Complete listings of both character sets can be found in Appendix E and Appendix F of the *Commodore 64 User's Guide*, and in Appendix B and Appendix C of the *Commodore 64 Programmer's Reference Guide*.) You can copy either of these character sets from ROM to RAM—or, if you wish, you can copy both of them. But whether you use a full character set or a partial set, it must begin at a memory address that is evenly divisible by $800—that is, on a 2K boundary. In a moment, you will be presented with a table that lists all of the permissible starting addresses for a full or partial character set copied from ROM to RAM, but first, here is an explanation of how to use the table.

As mentioned a few paragraphs ago, you can put a character set in any of the four banks of memory in the Commodore 64, provided that you tell the VIC-II chip exactly where you put it, and the way to do that is to store a code number into three bits of a certain memory register: specifically, $D018.

THE VIC-II MEMORY CONTROL REGISTER ($D018)

Memory Register $D018 is called the *VIC-II Memory Control Register,* and is often labeled *VMCSB* in Assembly language programs. To tell the VIC-II chip where a RAM character set has been located, all you have to do is store the proper code in Bits 0 through 3 of the VMCSB register, as illustrated in Table 10-3.

As you can see from this table, the meaning of the value stored in Bits 0 through 3 of the VMCSB register can vary, depending upon which bank of memory the VIC-II chip is looking at. So, before this table is used, the VIC-II chip must be told (in accordance with the instructions given in the previous section of this chapter) which memory bank to look into for graphics data.

Here is another important tip: although bits 1 through 3 of the VMCSB register are used to indicate the starting address of character sets, Bits 4 through 7 of the same register are used to inform the VIC-II chip of the starting address of screen memory. So when you use Bits 1 through 3 of this register, you have to be careful not to disturb whatever value might be stored in Bits 4 through 7. (You do not have to worry about Bit 0, since it is not significant.)

To set Bits 1 through 3 of VMCSB without disturbing the upper four bits of the register, you can use a routine like this one:

```
ALTERING THE LOWER NIBBLE OF VMCSB

LDA VMCSB
```

```
AND #$F0 ;CLEAR LOWER NIBBLE
ORA #$0E ;(SAMPLE VALUE FROM CHART BELOW)
STA VMCSB
```

Here is a table showing values that can be used in this equation.

Table 10-3 RAM CHARACTER SET STARTING ADDRESSES (Store Starting Address Code in $D018 (VMCSB) as Follows:)

BITS TO SET	HEX NO.	STARTING ADDRESSES Bank 0	Bank 1	Bank 2	Bank 3
XXXX111X	$0E	$3800	$7800	$B800	N/A*
XXXX110X	$0C	$3000	$7000	$B000	N/A*
XXXX101X	$0A	$2800	$6800	$A800	N/A*
XXXX100X	$08	$2000	$6000	$A000	N/A*
XXXX011X	$06	$1800**	$5800	$9800**	N/A*
XXXX010X	$04	$1000**	$5000	$9000**	N/A*
XXXX001X	$02	$0800	$4800	$8800	$C800
XXXX000X	$00	N/A*	$4000	$8000	$C000

*Memory block not normally available for storage of character data.
**These blocks are where ROM character images are stored; RAM stored there is not visible to the VIC-II chip, and thus cannot be used for storage of user-generated character sets or any other kinds of graphics data.

MOVING A SCREEN MAP

The address of screen memory, as previously noted, is changed by altering the upper four bits of the VMCSB register (Memory Register $D018). When these four bits are changed, care must be taken not to disturb the lower nibble of the VMCSB register, since altering that nibble controls the location of the C64 character set. A routine such as this can be used to change the upper nibble of VMCSB without changing the lower nibble:

```
ALTERING THE UPPER NIBBLE OF VMCSB

LDA VMCSB
AND #$0F ;CLEAR UPPER NIBBLE
ORA #$80 ;(SAMPLE VALUE FROM CHART BELOW)
STA VMCSB
```

Here is a table of values that can be used in this equation.

Table 10-4 SCREEN MEMORY STARTING ADDRESSES (Store Starting Address Code in $D018 (VMCSB) as Follows:)

BITS	HEX	STARTING ADDRESSES			
TO SET	NO.	Bank 0	Bank 1	Bank 2	Bank 3
1111XXXX	$F0	$3C00	$7C00	$BC00	N/A*
1110XXXX	$E0	$3800	$7800	$B800	N/A*
1101XXXX	$D0	$3400	$7400	$B400	N/A*
1100XXXX	$C0	$3000	$7000	$B000	N/A*
1011XXXX	$B0	$2C00	$6C00	$AC00	N/A*
1010XXXX	$A0	$2800	$6800	$A800	N/A*
1001XXXX	$90	$2400	$6400	$A400	N/A*
1000XXXX	$80	$2000	$6000	$A000	N/A*
0111XXXX	$70	$1C00**	$5C00	$9C00**	N/A*
0110XXXX	$60	$1800**	$5800	$9800**	$D800***
0101XXXX	$50	$1400**	$5400	$9400**	N/A*
0100XXXX	$40	$1000**	$5000	$9000**	N/A*
0011XXXX	$30	$0C00	$4C00	$8C00	$CC00
0010XXXX	$20	$0800	$4800	$8800	$C800
0001XXXX	$10	$0400	$4400	$8400	$C400
0000XXXX	$00	N/A*	$4000	$8000	$C000

*Block not normally available for storage of screen memory data.
**These blocks are where ROM character images are stored; RAM stored there is not visible to the VIC-II chip, and thus cannot be used for storage of other kinds of graphics data.
***Default storage area for color memory. This memory block is not large enough for storage of a high-resolution screen map, and must be used for storage of color data when C64 is in text mode. So this address is normally not available for storage of screen-map data in either text or bit-map mode.

MEMORY MANAGEMENT OF PROGRAMS CALLED FROM BASIC

When a machine-language program has to share memory space with a BASIC program, problems in memory allocation may arise. BASIC uses memory registers all over the Commodore 64's memory map. So unless special precautions are taken, a BASIC program may overwrite any machine language program that is stored in the block of free RAM that extends from $0800 to $9FFF, even if the machine-language program is located in the higher reaches of this block of RAM.

One way to keep a BASIC program from clobbering a machine-language program is to store the machine-language program in the $C000–$CFFF block of memory, a segment into which BASIC does not intrude. However, machine-language programs are often too long to fit into this 4K block of RAM. And, as previously mentioned, the memory requirements of some assemblers (such as the Merlin 64 and the Panther 64) make it difficult to write machine-language programs that can be stored in this block of memory.

Fortunately, there is a way out of all this. You can keep a BASIC program from taking over the higher regions of the $0800–$9FFF block of memory by changing the values stored in two pairs of memory registers, one called FRETOP and one called MEMSIZ.

FRETOP is a pointer that resides at Memory Registers $33 and $34 (51 and 52 in decimal notation). Its job is to point to the top of the memory area in which BASIC stores text strings and other kinds of data. BASIC is not permitted to store data at any address higher than the one indicated by this pointer.

MEMSIZ is a pointer stored in Memory Registers $37 and $38 (55 and 56 in decimal notation). BASIC programs cannot advance beyond the memory limit set by this pointer.

To keep a BASIC program from overwriting a machine-language program, all you have to do is lower the values of FRETOP and MEMSIZ, and then start your machine-language object code at a higher address. Normally, BASIC programs are allowed to extend all the way from $0800 to $A000, the top of the Commodore 64's free RAM, but with the help of FRETOP and MEMSIZ, the top of the block of RAM available to BASIC can be lowered to almost any desired value. The amount of free memory that should be allocated to a BASIC program will depend, of course, on the size of the BASIC program being used, but very few BASIC programs come anywhere close to consuming the 38K of free RAM that is made available to BASIC programs by the Commodore 64.

Suppose now, that you were working with a very long BASIC program that consumed a total of 24K of RAM. If such a program started at Memory Address $800—as BASIC programs written for the Commodore 64 usually do—then it would not require any RAM space past Memory Address $6000. In such a case as this, the contents of both FRETOP and MEMSIZ could be set at $6000, and the 16K block of memory extending from $6000 to $A000 could then be used for storage of a machine-language program.

The contents of FRETOP and MEMSIZ could be altered in either the BASIC segment or the machine-language segment of a BASIC/machine-language program. However, since machine-language programs that share memory space with BASIC programs are usually called from BASIC, FRETOP and MEMSIZ are most commonly altered using BASIC commands.

Here is an example of a BASIC routine that could be used to change the contents of the FRETOP and MEMTOP pointers to a value of $6000 (24576 in decimal notation):

```
CHANGING THE TOP OF BASIC RAM

10 NEW=24576:REM $6000
15 HI=INT(NEW/256):LO=NEW-HI*256
20 POKE 51,LO:POKE 52,HI:POKE 55, LO:POKE 56,HI:CLR
```

If you used a routine such as this one in a BASIC program, nothing contained in the program would be permitted to extend past Memory Address $6000 (24576 in decimal notation), and the block of RAM extending from $6000 to $A000 would thus be a safe location for a machine-language program.

One noteworthy feature of this program is the use of the instruction CLR in Line 20. This instruction removes all previously declared variables from memory, leaving all RAM above BASIC's new limits free for use by machine-language programs. Caution must be exercised in the use of CLR, of course, since any BASIC variables that have been declared prior to its appearance in a program will be wiped out.

MEMORY PROBLEMS CAUSED BY ASSEMBLERS

Here is another point to remember. Occasionally, an Assembly language programmer runs into unexpected difficulties when a machine-language program overwrites the machine code used by the assembler, editor, or monitor on which the program is being written. When that happens, the program being written and the assembler that is being used to write it can mess each other up terribly, sometimes with disastrous results.

To help keep that kind of calamity from taking place in Assembly language programs that you write, here are three short memory maps—one for each of the three assembler/editor packages that were used to write the programs in this book.

MEMORY MAP OF THE MERLIN 64 ASSEMBLER

ADDRESSES	FUNCTIONS
\$0000–\$01FF	Used by Merlin and C64 operating system (\$FB–\$FF not used by Merlin or C64)
\$0200–\$0258	Not used by Merlin (but C64 uses this area for BASIC and cassette recorder I/O)
\$0259–\$089F	Used by Merlin and C64 operating system
\$08A0–\$08FF	Not used by Merlin or C64; free RAM
\$0900–\$09FF	Used by Merlin; not used by C64
\$0A00–\$7FFF	Used for source code; can be altered with a Merlin editing command (WO)
\$8000–\$9FFF	Used for object code (starting address can be altered)
\$A000–\$CFFF	Area in which Merlin 64 assembler/editor program is stored

You can see from this map why \$8000 is a good starting address for object code written using the Merlin 64 assembler, but it is not the only starting address that can be used. You can declare a higher starting address with the ORG pseudo-op, and you can also declare a lower starting address, provided that you first use Merlin's WO command to alter the upper limit of the assembler's source-code text buffer.

As you can also see from the above map, the program used to run the Merlin assembler starts at \$A000. User-generated programs can extend beyond that address, but if they do, a special technique has to be employed to assemble them. When you assemble a program that will extend beyond Memory Address $A000, its object code has to be written directly to disk instead of being stored in the Commodore 64's memory. Once an object-code program has been assembled on a disk in this fashion, it can be run like any other machine-language program; it can be loaded into the C64's memory when Merlin is not present, and can then be executed without the help of the Merlin package.

MEMORY MAP OF THE COMMODORE 64 ASSEMBLER

From a memory-management point of view, the Commodore 64 Macro Assembler Development System is by far the most versatile of the three assembler/editor packages used in the writing of this book. The Commodore assembler has two monitors—one at $8000 and the other at $C000—and two machine language loaders, one at $0800 and the other at $C800. So, if you are skilled at using the Commodore 64 assembler, you can use it to write machine-language programs that can be stored anywhere in your computer's memory. Here is a table showing the memory locations of each program in the Commodore 64 Macro Assembler package.

ADDRESS	PROGRAM
$0800	LOADER64 (Machine-Language Loader No. 1)
$C800	HILOADER64 (Machine-Language Loader No. 2)
$8000	MONITOR$8000 (Machine-Language Monitor No. 1)
$C000	MONITOR$C000 (Machine-Language Monitor No. 2)
$C000	EDITOR64 (Assembly Language Editor)
$080F	Assembler64 (Machine-Language Assembler)
$CC00	DOS WEDGE64 (DOS Wedge)

MEMORY MAP OF THE PANTHER C64

Here is a map showing the memory requirements of the Panther C64 assembler.

ADDRESSES	PROGRAM
$4000-$7FFF	Source-Code Text Buffer (Default Location)
$8000-$8FFF	Source-Code Symbol Table (Default Location)
$9000-$C7EC	Panther C64 Assembler
$C7ED-$CFFF	Panther C64 Monitor

As you can see, the programs in the Panther C64 assembler package are all situated in high RAM—from $4000 on up—and consume most of the Commodore's free RAM space above that address. That is why most of the programs in this book that were written using the Panther C64 assembler start in low RAM—memory address $2000. This is not my favorite location for machine-language programs, since it severely restricts the size of any BASIC programs that may be used together with machine-language programs.

The designers of the Panther C64 did provide a method for changing a couple of the default settings of two of the programs in the Panther package. The assembler is equipped with a TEXT command that can be used for changing the starting address of the assembler's source-code text buffer, and there is a SYMBOL command that can be used to move the symbol table, but, since the other two Panther programs consume all free RAM from $9000 to $CFFF, there are not many places left for the source-code text buffer and the symbol table, even if one did want to move them.

11 High-Resolution Commodore Graphics

Joystick Operations

If you are itching to start writing graphics and sound programs in Assembly language, then this chapter is the one you have been waiting for. In this chapter, and the ones that follow, you will have a chance to start learning (finally) how to write Assembly language graphics and sound programs for your Commodore.

The programs in this chapter, like those throughout the rest of the book, were designed to be compatible with both the Commodore 64 and the Commodore 128. So if you own a Commodore 128, you will have to put it into its C64 mode before you type, assemble, and run the programs in this chapter and the ones that follow. The Commodore 128 has a few special features —such as an expanded memory and 80-column color graphics—that will not be specifically covered in the chapters, but if you learn everything that is in this section of this book, you will have no trouble picking up the rest of what you need to program the Commodore 128 when it has all its bells and whistles on.

PREVIEWS OF COMING ATTRACTIONS

In this chapter, you will get a chance to type, assemble, and run a program that can turn the Commodore 64 screen into a bit-mapped graphics tablet. The program is called SKETCHER, and once you understand how it works, you will be able to create high-resolution pictures on your C64 screen using a light pen, a trackball, or a joystick controller. In later chapters, you will learn how to create custom-designed characters, how to create giant-sized headline characters, how to program and animate sprites, and how to create music and special effects on the Commodore 128 and the Commodore 64.

We will start with a short program that is written in BASIC rather than Assembly language. In addition to creating a nice, colorful display with some entertaining animation, it illustrates some important principles that are often used in Commodore 64 graphics. So please do not skip over this program; you will run across some of those same principles later on in the next few chapters, when we start examining some more complex graphics programs that are written in Assembly language.

```
A GRAPHICS PROGRAM IN BASIC: FOLLOW THE BOUNCING BALL

10 REM **** BALLBOUNCE.BAS ****
20 PRINT CHR$(147):REM CLEAR SCREEN
30 BALL=81:SPACE=-96:RULE=99:REM CODES TO PRINT
   THINGS ON THE SCREEN
40 FOR L=55616 TO 55975:POKE L,2:NEXT L:REM MAKE BALL
   RED
50 FOR L=55976 TO 56015:POKE L,7:NEXT L:REM MAKE
   FLOOR YELLOW
60 POKE 53281,0:POKE 53280,6:REM BLACK BACKGROUND,
   BLUE BORDER
70 PRINT CHR$(5):REM WHITE TEXT
80 PRINT:PRINT:PRINT "    FOLLOW THE BOUNCING BALL . . ."
90 FOR L=1704 TO 1743:POKE L,RULE:NEXT L:REM DRAW
   FLOOR
100 PSN=1664:CT=1:REM STARTING POSITION AND FRAME
    COUNTER
110 FOR INC=1 TO 8:GOSUB 210:REM THIS LOOP DRAWS THE
    BALL GOING UP
120 PSN=PSN-40+1:REM THE BALL GOES UP
130 IF CT>40 THEN PSN=1344:CT=1:GOTO 150:REM BALL OFF
    SCREEN-BACK TO BEGINNING
140 NEXT INC
150 FOR DEC=1 TO TO 8:GOSUB 210:REM THIS LOOP DRAWS
    THE BALL COMING DOWN
160 PSN=PSN+41:REM THE BALL COMES DOWN
170 IF CT>40 THEN 100:REM BALL OFF SCREEN-LOOP BACK
180 NEXT DEC
190 GOTO 110:REM DONE-START AGAIN
200 REM **** PRINT BALL ON SCREEN ****
210 POKE PSN,BALL
220 FOR L=1 TO 50:NEXT L
230 POKE PSN, SPACE
240 CT=CT+1:RETURN
250 END
```

HOW IT WORKS

If you are familiar with Commodore 64 graphics, you may know that the C64 is capable of displaying up to 1,000 characters at a time on its screen, in a format that measures 40 columns by 25 lines. To hold the 1,000 characters that it can display on its screen, the Commodore 64 uses a specific block memory that is—

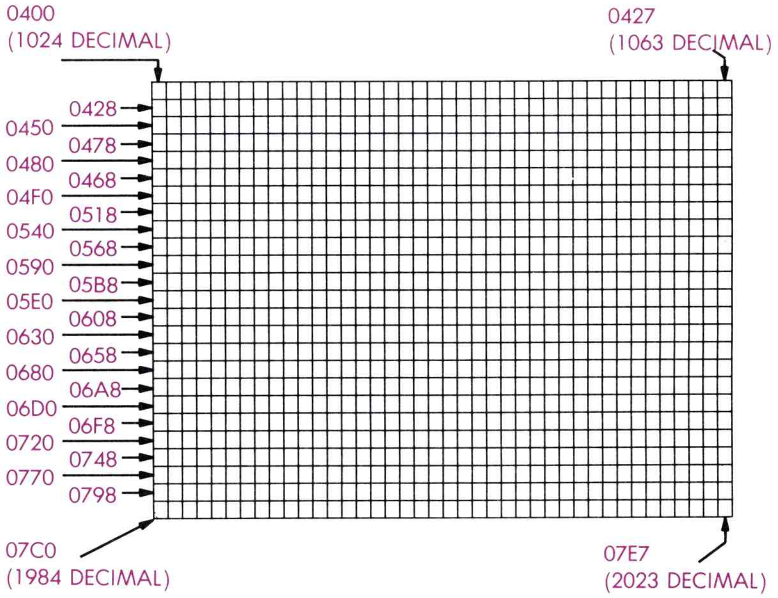

Figure 11-1 Screen Memory Map of the Commodore 64 (Default Addresses)

not surprisingly—1,000 bytes long. This block of memory, called *screen memory,* normally starts at Address No. 1024 ($400 in hexadecimal notation) and extends to Address No. 2023 ($7E7 in hex). This block of screen memory can be visualized as a grid of squares measuring 40 columns wide by 25 lines high, with each square representing one screen location.

When you type a character on the Commodore 64's keyboard, your computer's operating system translates the character you have typed into a special code, and then prints the character on your screen by storing its code number in the appropriate screen memory location. The character codes that are used for this purpose are not the standard ASCII codes that are commonly used by computer printers and for computer-to-computer communications. Instead, the Commodore 64 uses a special set of screen codes that includes many special characters as well as the standard set of ASCII characters. You can find a complete listing of this set of screen codes beginning on page 132 of your *Commodore 64 User's Guide*, and on page 376 of the *Commodore 64 Programmer's Reference Guide*.

Once you know what these screen display codes are, and where the screen display memory in your computer is, you can print text and graphics characters on your computer's screen by poking their screen-code values directly into the appropriate screen memory locations. In this way, you can bypass your computer's operating system and screen editor at any time you like, and can print anything you like directly on your screen.

In addition to its 1,000-byte block of screen memory, the Commodore 64 also has a corresponding 1,000-byte block of *color memory*. This block of color RAM begins at Memory Location 55296 ($D800 in hexadecimal notation) and extends to Memory Location 56295 (hexadecimal $DBE7). This bank of color memory can also be thought of as a 40-column by 25-line matrix of squares, with each square representing one *pixel* (picture element) on your computer screen.

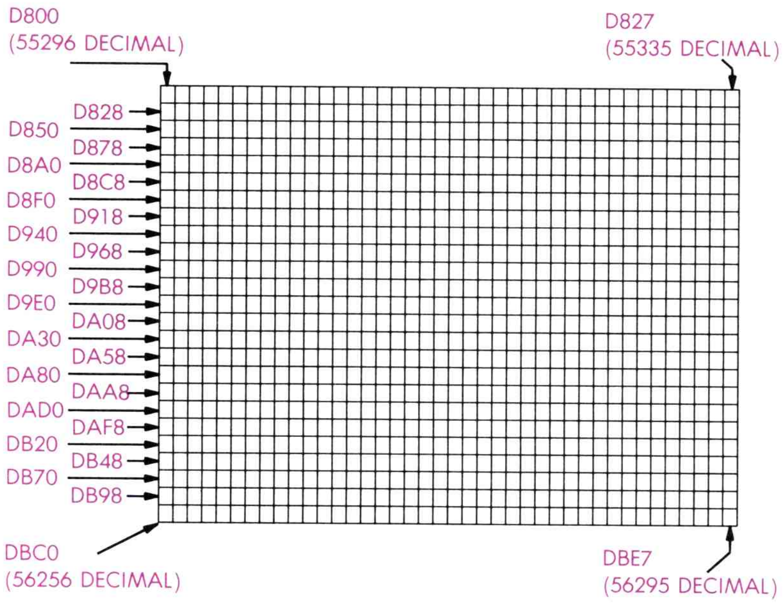

Figure 11-2 Color Memory Map of the Commodore 64 (Default Addresses)

The block of screen memory and the block of color memory inside your computer are designed to be used together. In addition to its screen display codes, the Commodore 64 also has a set of 16 color codes. And, by poking those codes into your computer's color memory map, you can determine the color of each individual text or graphics character that appears on your computer screen.

Table 11-1 lists the color codes used by the Commodore 64.

Table 11-1 COMMODORE 64 COLOR CODES

0	Black	8	Orange
1	White	9	Brown
2	Red	10	Light Red
3	Cyan	11	Gray 1
4	Purple	12	Gray 2
5	Green	13	Light Green
6	Blue	14	Light Blue
7	Yellow	15	Gray 3

Take a look at the two screen maps that were presented earlier in this chapter—the screen map and the color map—and you can see how the C64's screen memory and color memory work together. Since both maps have exactly the same measurements—40 columns wide by 25 columns deep—the 16 colors that can be used on the color map can be visualized as a set of 16 colored overlays. By placing these overlays on the appropriate portions of your computer's screen map, you can make the characters on the grid appear in any color you choose.

Look at the screen color map, and you can see exactly how this color overlay concept was used in creating the BALLBOUNCE program. In Line 40, a loop is used to place a red overlay over the top two thirds of the screen—from the top line of the screen all the way down to the line that begins at Memory Location 55976. When this red overlay is first laid down, it is invisible, since nothing has been drawn on the screen so far. But as soon as something is printed on the portion of the screen covered by the overlay—for example, a bouncing ball—the character will show up in red, as you have just seen if you ran the program.

After the red overlay is in place, a yellow one is laid down. This yellow overlay is just one pixel high; it runs across the screen horizontally in the form of a line that extends from Location 55976 to Location 56015. Next, a line is drawn across the screen, and comes out yellow because it lies under our yellow overlay.

Once the red and yellow overlays are in place, the words "FOLLOW THE BOUNCING BALL . . ." are printed across the top of the screen in white letters, using conventional PRINT commands. Then, in Lines 210 through 240, a red ball is sent bouncing across the screen. The animation technique employed here is quite simple; a circle (Screen Code 81) is drawn on the screen, then suddenly erased and redrawn in a new location. The square in which the ball appears keeps changing, and the effect is one of crude animation. When you consider that the program is written in BASIC, and that sprites (movable, arcade-style graphics characters) are not used, the animation in this little program is pretty effective. But a number of far better animation techniques are available to Assembly language programmers, as you will discover later on in this book.

CONTROLLING ANIMATION WITH JOYSTICKS

Now we will continue our exploration of Commodore 64 graphics with a BASIC program called JOYSTIX.BAS. This program also contains some very important principles that are also applicable to graphics programs written in Assembly language. Type it and run it; it will help you understand the Assembly language programs that follow.

To run the program, you will need one game controller: a joystick or, even better, a trackball. Plug the controller into Joystick Port A, load the program into RAM, and type RUN. After a few moments, a flickering yellow dot should appear in the middle of your screen.

Move your joystick, and the yellow dot will move around the screen. Press your joystick's trigger button while you are moving the stick, and the yellow dot will leave a trail of dots behind as it moves around. When you move the dot over other dots, with the trigger button up, it will erase the other dots. Try it, and watch the show.

Here is the program:

```
10 REM ***** JOYSTIX.BAS **********
20 PRINT CHR$(147):REM CLEAR SCREEN
30 BASE=1024:REM START OF SCREEN MEMORY
40 X=INT(40/2):REM POSITION X HALFWAY ACROSS SCREEN
50 Y=INT(25/2):REM POSITION Y HALFWAY DOWN SCREEN
60 POKE 53280,4:POKE 53281,0:REM PURPLE BORDER, BLACK
   BACKGROUND
70 FOR L=55296 TO 56295:REM LOW-RES SCREEN COLOR MAP
80 POKE L,7:NEXT L:REM YELLOW CHARACTERS
90 REM **** READ JOYSTICK ****
100 POKE BASE+X+40*Y,81:REM PRINT DOT AT SCREEN
    POSITION X,Y
110 JV=PEEK(56320):REM GET JOYSTICK VALUE
120 TB=JV AND 16:REM GET TRIGGER BUTTON STATUS
130 JV=15-(JV AND 15):REM CONVERT SWITCH VALUES TO A
    NR BETWEEN 0 AND 10
140 IF TB=16 THEN POKE BASE+X+40*Y,32:REM IF TB NOT
    PRESSED, PRINT SPACE
150 IF JV<>0 THEN 170:REM JOYSTICK HAS BEEN
    ACTIVATED; READ IT
160 GOTO 100
170 ON JV GOTO
    1100,1200,1300,1400,1500,1600,1700,1800,1900,2000
1000 REM ******** READ POINTER *********
1100 Y=Y-1:IF Y<0 THEN Y=24:REM UP
1110 GOTO 100:REM PRINT PIXEL
1200 Y=Y+1:IF Y>24 THEN Y=0:REM DOWN
1210 GOTO 100
1300 GOTO 100:REM NO ACTION
1400 X=X-1:IF X<0 THEN X=39:REM LEFT
1410 GOTO 100
1500 X=X-1:IF X<0 THEN X=39:REM LEFT...
1510 Y=Y-1:IF Y<0 THEN Y=24:REM AND UP
1520 GOTO 100
1600 X=X-1:IF X<0 THEN X=39:REM LEFT...
1610 Y=Y+1:IF Y>24 THEN Y=0:REM AND DOWN
1620 GOTO 100
1700 GOTO 100:REM NO ACTION
1800 X=X+1:IF X>40 THEN X=0:REM RT..
1810 GOTO 100
1900 X=X+1:IF X>40 THEN X=0:REM RT..
1910 Y=Y-1:IF Y<0 THEN Y=24:REM ..AND UP
1920 GOTO 100
2000 X=X+1:IF X>40 THEN X=0:REM RT..
2010 Y=Y+1:IF Y>24 THEN Y=0:REM AND DOWN
2020 GOTO 160
```

PRINCIPLES OF THE PROGRAM

Most of the graphics principles that are used in this program were explained pretty thoroughly in Chapter 10. And you may recall others that were mentioned in the text that accompanied the program BALLBOUNCE.BAS.

Here is how the JOYSTIX program works.

In Line 3, an important constant called BASE is defined. The value of this constant, 1024 ($400 in hexadecimal notation), is the default starting address of the Commodore 64's low-resolution screen map. Then in Lines 40 and 50, two variables (called X and Y) are set up for use as screen coordinates. Their initial values are defined as INT(40/2) for X and INT(25/2) for Y. Since the Commodore's low-resolution screen measures 40 columns wide by 25 lines high, these values will cause a dot to be printed at midscreen when the program begins.

Line 60 sets the border and background screen colors for the JOYSTIX screen display. Then, in Lines 70 and 80, the value 7 is stored in every byte of RAM in the memory block that extends from Memory Register 55296 ($D800) to 56295 ($DBE7). This block-fill operation ensures that all of the characters which we will be printing on the screen will be yellow. As you may recall from the BALLBOUNCE program in Chapter 1, the block of RAM that extends from $D800 to $DBE7 can be thought of as a color overlay that can be placed over the Commodore 64's screen map. When a color is stored in a pixel on this overlay, then the corresponding pixel on the Commodore screen map will be displayed in the chosen color. Since the number 7 is the Commodore 64's code for the color yellow (for a list of all color codes, see the table in Appendix D of the *Commodore 64 Programmer's Reference Guide*), storing a 7 in every byte of memory from $D800 through $DBE7 will make every character that appears on the screen yellow; hence, what you will get in the JOYSTIX program is yellow dots.

A JOYSTICK-READING ROUTINE

The next section of the JOYSTIX.BAS program is the one that reads the game controller in Port A. The Commodore 64 has a pair of joystick ports that are often referred to as Port A and Port B. The status of Port A can be determined by peeking into an 8-bit memory register located at Memory Address 56320. The status of Port B can be determined by peeking into a similar memory register at Memory Address 56321.

Each of the two joysticks that can be plugged into the Commodore 64 contains five on/off switches. Four of these switches correspond to the four primary directions in which a joystick can be pushed: up, down, left, and right. If the joystick is moved diagonally, two switches can be activated simultaneously, and can be read in combination. In this way, diagonal movements of the joystick can be detected.

The fifth switch inside a joystick is used to determine whether the joystick's trigger button is pressed or not pressed.

The JOYSTIX.BAS program is designed to read a joystick that is plugged into Port A of the Commodore 64 console. The program reads the joystick by peeking into Memory Address 56320. Table 11-2 shows all possible values that can be found in that location, and their meanings.

Table 11-2 COMMODORE 64 JOYSTICK VALUES

SWITCH VALUE	BINARY VALUE	MEANING
0	0000 0000	No action
1	0000 0001	Up
2	0000 0010	Down
3	0000 0011	None
4	0000 0100	Left
5	0000 0101	Left + up
6	0000 0110	Left + down
7	0000 0111	None
8	0000 1000	Right
9	0000 1001	Right + up
10	0000 1010	Right + down
11	0000 1011	None
12	0000 1100	None
13	0000 1101	None
14	0000 1110	None
15	0000 1111	None
16	0001 0000	Trigger button pressed
17	0001 0001	Trigger + up
18	0001 0010	Trigger + down
19	0001 0011	None
20	0001 0100	Trigger + left
21	0001 0101	Trigger + left + up
22	0001 0110	Trigger + left + down
23	0001 0111	None
24	0001 1000	Trigger + right
25	0001 1001	Trigger + right + up
26	0001 1010	Trigger + right + down
26	0001 1010	None

After the flashing dot is placed in the center of the screen in the JOYSTIX.BAS program, a loop is set up to determine the value of the joystick's direction switches. But before the direction of the joystick is read, a test is conducted to determine whether the trigger button is being pressed. If the trigger button is pressed, then a small circle is drawn on the screen at the current joystick position. If the trigger button is not being pressed, no circle is drawn, and a space is printed on the screen to erase any circle that may have been previously printed in that location.

The result of all of this plotting, drawing, and erasing of tiny circles is quite entertaining. By pressing a joystick trigger and moving the joystick around, you can draw patterns of circles all over your computer screen. By moving the joystick and not pressing the trigger button, you can move your cursor without drawing any patterns, and you can also erase any patterns that may lie in your path.

WHEN BASIC IS ENOUGH

The JOYSTIX program works so well in BASIC that there is absolutely no reason for translating it into Assembly language. In fact, if it were converted to Assembly language, it would run too fast. At one point, I did translate it into Assembly language, and the results were totally unsatisfactory. Each time the joystick was moved, long strings of dots would instantly appear on the screen, instead of moving across the screen gently and controllably, as they do in the BASIC version of the program.

BASIC is not such a great language, though, for writing *high-resolution* graphics programs. High-resolution programs have to be written using an extremely complicated technique called *bit-mapping,* and BASIC is far too slow to handle bit-mapping routines efficiently. This next program in this chapter, called BLACKBOARD, will clearly demonstrate how unbearably slow a high-resolution program written in BASIC can be, but as you sit and wait for the program to crawl through its paces, please do not get too impatient. Later on, you will get an opportunity to type, assemble, and execute an Assembly language version of exactly the same program—and I guarantee that you will notice the change.

First, though, type this program and run it. You will then see very clearly why BASIC is not considered the best language for writing high-resolution graphics programs, and why fast-action arcade games are almost always written in Assembly language.

```
A HIGH-RESOLUTION BASIC PROGRAM

10 REM *** "BLACKBOARD.BAS" *******
20 BASE=2*4096:POKE 53272,PEEK(53272)OR8:REM PUT
    HIGH-RES MAP AT 8192
30 POKE 53265,PEEK(53265)OR32:REM ENTER HIGH-RES BIT-
    MAP MODE
40 FOR I=BASE TO BASE+7999:POKE I,0:NEXT:REM CLEAR
    BIT MAP
50 FOR I=1024 TO 2023:POKE I,16:NEXT I:REM BLACK
    BACKGROUND, WHITE LINE
60 GOTO 200
80 REM ****** PLOT ROUTINE **********
90 CHAR=INT(HPSN/8)
100 ROW=INT(VPSN/8)
110 LINE=VPSN AND 7
```

```
120 BYTE=BASE+ROW*320+8*CHAR+LINE
130 BIT=7-(HPSN AND 7)
140 POKE BYTE,PEEK(BYTE) OR (2BIT)
150 RETURN
200 REM **** DRAW VERTICAL LINE *******
220 FOR VPSN=0 TO 199:REM PLOT LINE FROM TOP TO
    BOTTOM OF SCREEN
225 FOR HPSN=159 TO 160
230 GOSUB 80
240 NEXT HPSN:NEXT VPSN
245 REM *** DRAW HORIZONTAL LINE ******
250 VPSN=100:REM HALFWAY DOWN SCREEN
260 FOR HPSN=0 TO 319:REM PLOT LINE FROM LEFT SIDE TO
    RIGHT SIDE OF SCREEN
270 GOSUB 80
280 NEXT HPSN
290 GOTO 290
```

HOW BLACKBOARD.BAS WORKS

Now let's take a look at how the BLACKBOARD.BAS program works.

The Commodore 64, as pointed out in previous chapters, has two primary display modes: a text mode and a high-resolution graphics mode. (There is also a multi-color text mode, and there is a multi-color graphics mode, but we will not be discussing either of those modes in this chapter.)

When the C64 is in its text mode, it displays 25 lines of 40 typed characters each, or a total of 1,000 characters, on its monitor screen. Each of these characters is made up of eight bytes of binary data.

In its high-resolution mode, the Commodore produces a screen display 320 dots (or pixels) wide and 200 pixels high. That's a total of 64,000 separate dots, each of which requires one bit of memory. So it takes 8,000 bytes of memory to produce a high-resolution screen display.

When the C64 is in its text mode, the characters that it uses to create its text display are stored in its memory in the form of binary data. It takes eight bits of data to create one character. When the eight bits of data that form a character's image are displayed on the screen, they are arranged as shown at the top of the next page.

When your Commodore is in its text mode and you type a character on the screen, a code number representing that character is stored in the block of RAM that has been designated as *screen memory*. (Sometimes, as you remember from Chapter 10, this block of memory is called a *screen map*.) Each time the C64's VIC-II graphics chip creates a screen display (and it does that 60 times every second), it fetches each character code that is stored in the screen memory, and uses it as a pointer to another block of memory called *character*

BINARY NOTATION	HEXADECIMAL NOTATION	APPEARANCE
00000000	00	
00011000	18	`   XX   `
00111100	3C	`  XXXX  `
01100110	66	` XX  XX `
01100110	66	` XX  XX `
01111110	7E	` XXXXXX `
01100110	66	` XX  XX `
00000000	00	

generator ROM. In the block of memory called character generator ROM, a 64-bit image of every character in the Commodore 64's character set is stored in the form of a series of eight bytes. And those eight bytes are what the VIC-II chip uses to create the characters which it displays on the Commodore 64's video screen.

SETTING UP A BIT-MAPPED DISPLAY

When your computer is in one of its bit-mapped modes, it does not use the preprogrammed characters stored in character-generator ROM. Instead, each individual dot on its video screen map is represented by one bit of data in the block of RAM that is used as a high-resolution screen map. So, if you know the exact position of a dot on the screen, you can turn that dot off or on by simply setting or clearing its corresponding bit in video memory. In this way, you can control every dot on the screen.

This would make bit-mapping a very simple matter if a dot could be plotted on the Commodore's screen using simple X/Y coordinates. Unfortunately, that is not the way high-resolution bit-mapping works on the Commodore 64. The dots that make up the Commodore's screen display do not run straight across and down the screen as they do in text mode. Instead, they are arranged just like they would be if they were *dots* in text characters—in 8-dot by 8-dot matrixes. These matrixes are placed on the screen in a 40-column by 25-line configuration, just as if they were standard text characters. This arrangement produces a high-resolution screen display that measures 320 dots (pixels) wide by 200 dots (pixels) high. That is a total of 64,000 dots, each one of which can be individually turned off and on.

Table 11-3 shows where a Commodore 64 gets the data that it uses for the first two rows of data on a high-resolution screen.

This zigzag layout makes it easy to mix text and bit-mapped graphics on the Commodore 64, since text and high-resolution graphics are laid out on the screen in exactly the same way. Unfortunately, it also makes the job of bit-mapping the C64 screen rather complicated. To map a dot on a Commodore 64 high-resolution display, you have to use a fairly complex mathematical formula. First you have to figure out where the dot lies on a 320-square by 200-square grid, using a pair of variables (which I will call X and Y) for the grid's column (X) and row (Y) coordinates. Then, since the C64 screen map is subdivided in 8-

Table 11-3

	COLUMN 1	COLUMN 2	COLUMN 3	. . . COLUMN 40
LINE 1	Byte 0	Byte 8	Byte 16	Byte 312
LINE 2	Byte 1	Byte 9	Byte 17	Byte 313
LINE 3	Byte 2	Byte 10	Byte 18	Byte 314
LINE 4	Byte 3	Byte 11	Byte 19	Byte 315
LINE 5	Byte 4	Byte 12	Byte 20	Byte 316
LINE 6	Byte 5	Byte 13	Byte 21	Byte 317
LINE 7	Byte 6	Byte 14	Byte 22	Byte 318
LINE 8	Byte 7	Byte 15	Byte 23	Byte 319
LINE 9	Byte 320	Byte 328	Byte 336	Byte 632
LINE 10	Byte 321	Byte 329	Byte 337	Byte 633
LINE 11	Byte 322	Byte 330	Byte 338	Byte 634
LINE 12	Byte 323	Byte 331	Byte 339	Byte 635
LINE 13	Byte 324	Byte 332	Byte 340	Byte 636
LINE 14	Byte 325	Byte 333	Byte 341	Byte 637
LINE 15	Byte 326	Byte 334	Byte 342	Byte 638
LINE 16	Byte 327	Byte 335	Byte 343	Byte 639
etc.				

dot by 8-dot matrixes, you have to break the screen map down into 8-dot by 8-dot subdivisions by dividing each coordinate by 8. Here is the way that is done in the BLACKBOARD.BAS program:

```
90 CHAR=INT(HPSN/8)
100 ROW=INT(VPSN/8)
```

Next, you have to figure out your dot's coordinates inside the 8-by-8 dot matrix in which it lies. You can do that with three more instructions such as these:

```
110 LINE=VPSN AND 7
120 BYTE=BASE+ROW*320+CHAR*8+LINE
```

Finally, you can turn on the bit you have selected with a line such as this:

```
140 POKE BYTE,PEEK(BYTE) OR (2^BIT)
```

This formula takes a long time to work out in BASIC, and that is why high-resolution graphics programs written in BASIC run so slowly. Fortunately, as you will soon see, the calculation takes much less time in Assembly language.

HIGH-RESOLUTION COLOR GRAPHICS

Another thing to remember when you use Commodore 64 high-res graphics is that the colors used in bit-mapped C64 graphics do not usually come from the 1,000-byte block of color RAM that begins at Memory Location 55296 ($D800 hex). Instead, they come from a screen memory map with a default address of 1024 ($400 in hexadecimal notation). When your computer is in its 320-dot by 200-dot bit-mapped mode, the values stored in what is ordinarily its screen memory do not equate to screen codes for ASCII characters. Instead, the upper four bits of each location in screen memory define the color of any bit that is set to 1 in a corresponding 8-dot by 8-dot area on your computer's video display. The lower four bits define the color of any bit that is set to a 0 in that same 8-by-8 block on the screen.

It is not too difficult to put the C64 into either of its high-resolution graphics modes. All you have to do is use a series of instructions like these:

```
20 BASE=2*4096:POKE 53272,PEEK(53272)OR8
30 POKE 53265,PEEK(53265)OR32
```

The first of these instructions—BASE=2*4096—informs your computer's VIC-II chip that you are going to be using a bit map that starts at Memory Address 8192 ($2000 in hexadecimal notation). Then, in the second instruction on Line 20—POKE 53272,PEEK(53272)OR8—the VIC chip is told where to place its screen map and where to find the data that it will need to display a bit-mapped screen. As you may recall from Chapter 10, Memory Address 53272 ($D018 in hexadecimal notation) is a memory register often referred to as VMCSB. When the C64 is in bit-mapped mode, the lower four bits of the VMCSB register are used to specify screen colors, and the upper four bits are used to point to the location of the bit map that will be used for the display.

In the instruction in Line 30, the VIC chip is finally instructed to go into high-resolution mode. This instruction is issued by setting Bit 4 of an important memory register that is often referred to as the SCROLY register. SCROLY is a multipurpose register that resides at Memory Address 53265 ($D011 in hexadecimal notation). One function of the SCROLY register is to enable fine scrolling—but we will not go into that in this chapter. For now, just remember that the SCROLY register is also used to determine whether the Commodore 64 will generate a text screen or a high-resolution display. If Bit 4 of the SCROLY register is set, the C64 will generate a high-resolution bit-mapped screen. If Bit 4 is clear, the computer will produce a text display.

Line 60 of the BLACKBOARD.BAS program is nothing but a jump to Line 200. In Lines 200 through 240, a vertical line is drawn down the center of the screen using a bit-mapping plot subroutine that appears in Lines 80 through 250. This subroutine employs the plotting formula described earlier in this chapter to print white dots on a black background on the screen.

The line that is drawn down the screen in Lines 220 through 240 is two dots wide. That is because it takes a two-dot width to form a good solid line on a Commodore 64 high-resolution screen display; because of video-display technicalities, a vertical line that is one one dot wide comes out not strong and white, but pale and gray. In the BLACKBOARD program, therefore, a two-step loop is used to make the vertical line on the screen two dots wide. This loop appears in Lines 225 to 240.

After the vertical line is drawn, a horizontal line is mapped across the screen in Lines 245 through 280. Horizontal lines that are one dot high look fine on a C64 screen display, so a two-dot loop is not needed in this horizontal line routine.

The BLACKBOARD program ends with an infinite loop at Line 290.

12 Drawing Pictures in High-Resolution Graphics

Some Advanced Features of Assembly Language

Now we are ready to take a look at how much better the BLACKBOARD program works in Assembly language. Here is what the program looks like when it is translated into Assembly language source code created with a Merlin 64 assembler:

THE BLACKBOARD PROGRAM IN ASSEMBLY LANGUAGE

```
*
* BLACKBOARD
*
 ORG $8000
*
COLOR EQU $10
BASE EQU $2000
SCROLY EQU $D011
VMCSB EQU $D018
COLMAP EQU $0400
*
HMAX EQU 320
VMAX EQU 200
HMID EQU 160
VMID EQU 100
*
SCRLEN EQU 8000
MAPLEN EQU 1000
*
TEMPA EQU $FB
TEMPB EQU TEMPA+2
*
TABPTR EQU TEMPA
TABSIZ EQU $9000
*
HPSN EQU TABSIZ+2
```

```
VPSN EQU HPSN+2
CHAR EQU VPSN+1
ROW EQU CHAR+1
LINE EQU ROW+1
BYTE EQU LINE+1
BITT EQU BYTE+2
*
MPRL EQU BITT+1
MPRH EQU MPRL+1
MPDL EQU MPRH+1
MPDH EQU MPDL+1
PRODL EQU MPDH+1
PRODH EQU PRODL+1
*
FILVAL EQU PRODH+1
JSV EQU FILVAL+1

*CIAPRA EQU $DC00

 JMP START
*
* BLOCK FILL ROUTINE
*
BLKFIL LDA FILVAL
 LDX TABSIZ+1
 BEQ PARTPG
 LDY #0
FULLPG STA (TABPTR),Y
 INY
 BNE FULLPG
 INC TABPTR+1
 DEX
 BNE FULLPG
PARTPG LDX TABSIZ
 BEQ FINI
 LDY #0
PARTLP STA (TABPTR),Y
 INY
 DEX
 BNE PARTLP
FINI RTS
*
* 16-BIT MULTIPLICATION ROUTINE
*
MULT16 LDA #0
 STA PRODL
 STA PRODH
 LDX #17
 CLC
MULT ROR PRODH
 ROR PRODL
 ROR MPRH
 ROR MPRL
 BCC CTDOWN
 CLC
 LDA MPDL
 ADC PRODL
 STA PRODL
 LDA MPDH
 ADC PRODH
 STA PRODH
CTDOWN DEX
 BNE MULT
 RTS
*
* PLOT ROUTINE
*
* ROW=VPSN/8 (8-BIT DIVIDE)
*
PLOT LDA VPSN
 LSR A
 LSR A
 LSR A
 STA ROW
*
* CHAR=HPSN/8 (16-BIT DIVIDE)
*
 LDA HPSN
 STA TEMPA
 LDA HPSN+1
 STA TEMPA+1
 LDX #3
DLOOP LSR TEMPA+1
 ROR TEMPA
 DEX
 BNE DLOOP
 LDA TEMPA
 STA CHAR
*
*
 LDA VPSN
 AND #7
 STA LINE
*
* BITT=7-(HPSN AND 7)
*
```

```
 LDA HPSN
 AND #7
 STA BITT
 SEC
 LDA #7
 SBC BITT
 STA BITT
*
* BYTE=BASE+ROW*HMAX+8*CHAR+LINE
*
* FIRST MULTIPLY ROW * HMAX
*
 LDA ROW
 STA MPRL
 LDA #0
 STA MPRH
 LDA #<HMAX
 STA MPDL
 LDA #>HMAX
 STA MPDH
 JSR MULT16
 LDA MPRL
 STA TEMPA
 LDA MPRL+1
 STA TEMPA+1
*
* ADD PRODUCT TO BASE
*
 CLC
 LDA #<BASE
 ADC TEMPA
 STA TEMPA
 LDA #>BASE
 ADC TEMPA+1
 STA TEMPA+1
*
* MULTIPLY 8 * CHAR
*
 LDA #8
 STA MPRL
 LDA #0
 STA MPRH
 LDA CHAR
 STA MPDL
 LDA #0
 STA MPDH
 JSR MULT16
 LDA MPRL
```

```
172  STA TEMPB
173  LDA MPRH
174  STA TEMPB+1
175 *
176 * ADD LINE
177 *
178  CLC
179  LDA TEMPB
180  ADC LINE
181  STA TEMPB
182  LDA TEMPB+1
183  ADC 0
184  STA TEMPB+1
185 *
186 * BYTE = TEMPA + TEMPB
187 *
188  CLC
189  LDA TEMPA
190  ADC TEMPB
191  STA TEMPB
192  LDA TEMPA+1
193  ADC TEMPB+1
194  STA TEMPB+1
195 *
196 * POKE BYTE,PEEK(BYTE)OR2^BIT
197 *
198  LDX BITT
199  INX
200  LDA #0
201  SEC
202 SQUARE ROL
203  DEX
204  BNE SQUARE
205  LDY #0
206  ORA (TEMPB),Y
207  STA (TEMPB),Y
208  RTS
209 *
210 * MAIN ROUTINE STARTS HERE
211 *
212 * FIRST DEFINE BIT MAP AND ENABLE
213 * HIGH-RESOLUTION GRAPHICS
214
215 START LDA #$18
216  STA VMCSB
217 *
218  LDA SCROLY
219  ORA #32
```

```
 STA SCROLY
*
* SELECT GRAPHICS BANK 1
*
 LDA $DD02
 ORA #$03
 STA $DD02
*
 LDA $DD00
 ORA #$03
 STA $DD00
*
* CLEAR BIT MAP
*
 LDA #0
 STA FILVAL
 LDA #<BASE
 STA TABPTR
 LDA #>BASE
 STA TABPTR+1240  LDA #<SCRLEN
 STA TABSIZ
 LDA #>SCRLEN
 STA TABSIZ+1
 JSR BLKFIL
*
* SET BKG AND LINE COLORS
*
 LDA #COLOR
 STA FILVAL
 LDA #<COLMAP
 STA TABPTR
 LDA #>COLMAP
 STA TABPTR+1
 LDA #<MAPLEN
 STA TABSIZ
 LDA #>MAPLEN
 STA TABSIZ+1
 JSR BLKFIL
*
* DRAW HORIZONTAL LINE
*
 LDA #VMID
 STA VPSN
 LDA #0
 STA HPSN
 STA HPSN+1
AGIN JSR PLOT
 INC HPSN
```

```
269  BNE NEXT
270  INC HPSN+1
271 NEXT LDA HPSN+1
272  CMP #>HMAX
273  BCC AGIN
274  LDA HPSN
275  CMP #<HMAX
276  BCC AGIN
277 *
278 * DRAW VERTICAL LINE
279 *
280  LDA #0
281  STA VPSN
282 POINT LDA #<HMID
283  STA HPSN
284  LDA #>HMID
285  STA HPSN+1
286  JSR PLOT
287  INC HPSN
288  BNE SKIP
289  INC HPSN+1
290 SKIP JSR PLOT
291  LDX VPSN
292  INX
293  STX VPSN
294  CPX #VMAX
295  BCC POINT
296 INF JMP INF
```

SOME BASIC DIFFERENCES

There are two obvious differences between BLACKBOARD.BAS and its Assembly language counterpart. One is that the Assembly language version of the program is much longer, and the other is that it runs much faster.

In most other respects, the BASIC and Assembly language versions of the BLACKBOARD program are quite a bit alike. There are a few features of the Assembly language program that may be worth pointing out, however.

One such feature is the subroutine labeled BLKFIL. It starts at Line 50, and it is the first routine in the program. (It is not the first routine that executes, however; look at Line 46, and you will see that the first thing the BLACKBOARD program does is jump to Line 215, which is labeled START. That is where execution of the program begins. BLKFIL—and the other routines that precede Line 215—are all subroutines that are designed to be called from the main body of the program.)

Now let's take a close look at the BLKFIL routine that begins at Line 50. In the BLACKBOARD program, this subroutine is used to clear a bit map and a color map, and to fill both of those memory blocks with the same values they were stuffed with in the BLACKBOARD.BAS program. But the BLKFIL routine—unlike the FOR/NEXT loops that were used for the same purpose in the BLACKBOARD.BAS program—operates at lightning speed. It can clear an 8K memory block so fast that you may miss the whole thing if you blink your eyes.

One reason the routine executes so fast is that it is designed to move data one "page" at a time. In 6502 Assembly language, a 256-byte chunk of memory is often referred to as a page. Since the hexadecimal equivalent of 256 is $0100, the address of any hexadecimal "page" in a computer's memory can be defined in two parts—a high-order byte and a low-order byte. The BLKFIL routine in the BLACKBOARD.S program manages memory pages very efficiently because it fills memory blocks a page at a time. After the high-order byte of the address of a page is defined, the routine fills every byte on that page with a desired value. Only then does it go to the next page. Once all full pages have been filled with data, it fills the remaining partial page. Because of this technique the routine can use one-byte addresses rather than two-byte addresses, speeding up its block-filling mission considerably.

In Lines 69 through 90, you will see the most sophisticated 16-bit multiplication routine that has appeared in this book so far. This routine can multiply two unsigned 16-bit numbers and can handle a product up to 32 bits long. When the routine ends, the low half of the product is stored in the variables labeled MPRL and MPRH, and the high half of the product is stored in PRODL and PRODH. This multiplication subroutine is used twice in the BLACKBOARD program: once in Lines 134 to 148, and again in Lines 160 to 174. Neither of these routines requires the use of a 32-bit product, so neither routine makes use of the variables PRODL and PRODH, but if you ever do need a multiplication routine that can handle a 32-bit product, here is one that fills the bill.

The 16/32-bit multiplication routine in the BLACKBOARD program is followed by a plotting routine that is much longer, but also runs much faster, than the plotting routine that accomplished the same task in the BLACKBOARD.BAS program.

One more point: while you are typing BLACKBOARD.S on your computer keyboard, you may notice that a couple of the equates in the program's symbol table do not appear in the main body of the program. Do not be concerned about that; we will be expanding the program later on, and these equates will be used then.

After the main body of BLACKBOARD.S begins (at Line 199), the program works exactly like its BASIC predecessor. It clears the bit map that starts at $2000, sets background and dot colors (you can change those if you like), and then draws a pair of crosshairs on your Commodore screen. Don't blink, or you might miss all of the action. I think you will agree that there is simply no comparison between the BLACKBOARD.S program's execution speed and that of its BASIC predecessor.

ONE MORE PROGRAM

Now we have come to the really fun part of this chapter—an Assembly language program that combines the best features of the JOYSTIX.BAS program and the BLACKBOARD.BAS routine. This program is called SKETCHER, and it is an electronic version of those plastic, carbon-filled Etch-a-Sketch drawing screens that you may remember from your childhood. The SKETCHER program is actually nothing but an expanded version of the BLACKBOARD.S program. So, if you have typed and assembled the BLACKBOARD.S program, you can easily expand your BLACKBOARD.S source-code listing into a SKETCHER program. Here is all you will have to do.

First, change Line 2 of the BLACKBOARD program to read:

```
2 * SKETCHER
```

Then replace Lines 259 through 296 of the BLACKBOARD program with Lines 259 through 424 as follows:

```
259 *
260 * PRINT DOT AT MIDSCREEN
261 *
262  LDA #VMID
263  STA VPSN
264  LDA #<HMID
265  STA HPSN
266  LDA #>HMID
267  STA HPSN+1
268  JSR PRINT
269 *
270 * READ JOYSTICK
271 *
272 * FIRST CHECK TRIGGER BUTTON
273 *
274 READJS LDA CIAPRA
275  AND #$10
276  BEQ START
277 *
278 * NOW READ JOYSTICK
279 *
280  LDA #$0F
281  PHA
282  AND CIAPRA
283  STA JSV
284  PLA
285  SEC
286  SBC JSV
287  STA JSV
288 *
289  TAX
290  BEQ READJS
291  LDA RELADS-1,X
292  STA MODREL+1
```

```
MODREL BNE *
MODR1
*
NIL1 JMP READJS
*
* ROUTINES TO MOVE JOYSTICK
*
UP JSR MOVEUP
 JMP DOIT
*
DOWN JSR MOVEDN
 JMP DOIT
*
LEFT LDX HPSN
 LDY HPSN+1
 TXA
 BNE DECLB
 DEY
DECLB DEX
 STX HPSN
 STY HPSN+1
 JMP DOIT
*
UPANDL JSR MOVEUP
 JMP LEFT
*
DNANDL JSR MOVEDN
 JMP LEFT
*
NIL2 JMP READJS
*
RIGHT LDX HPSN
 LDY HPSN+1
 INX
 BNE NOINCR
 INY
NOINCR STX HPSN
 STY HPSN+1
 JMP DOIT
*
UPANDR JSR MOVEUP
 JMP RIGHT
*
DNANDR JSR MOVEDN
 JMP RIGHT
*
* SUBROUTINES TO MOVE UP & DOWN
*
```

```
MOVEUP LDX VPSN
 DEX
 STX VPSN
 RTS
*
MOVEDN LDX VPSN
 INX
 STX VPSN
 RTS
*
* "DOIT" SUBROUTINE
*
DOIT JSR PRINT
 JMP READJS
*
* MORE SUBROUTINES START HERE
*
* MAKE SURE DOT IS WITHIN RANGE
*
CHECK LDA VPSN
 BEQ RAISE
 CMP #VMAX-1
 BCS LOWER
 JMP HCHECK
RAISE INC VPSN
 JMP HCHECK
LOWER LDA #VMAX-1
 STA VPSN
*
HCHECK BIT HPSN+1
 BPL OKLOW
 LDA #1
 STA HPSN
 LDA #0
 STA HPSN+1
 RTS
*
OKLOW LDA #<HMAX-2
 CMP HPSN
 LDA #>HMAX-2
 SBC HPSN+1
 BCC TOOHI
 RTS
*
TOOHI LDA #<HMAX-2
 STA HPSN
 LDA #>HMAX-2
 STA HPSN+1
 RTS
*
* PRINT DOT ON SCREEN
*
PRINT JSR CHECK
 JSR PLOT
*
 LDA HPSN
 PHA
 LDA HPSN+1
 PHA
*
 LDA HPSN
 BNE SKIP
 DEC HPSN+1
SKIP DEC HPSN
 JSR CHECK
 JSR PLOT
*
 PLA
 STA HPSN+1
 PLA
 STA HPSN
 RTS
*
RELADS DFB UP-MODR1
 DFB DOWN-MODR1
 DFB NIL1-MODR1
 DFB LEFT-MODR1
 DFB UPANDL-MODR1
 DFB DNANDL-MODR1
 DFB NIL2-MODR1
 DFB RIGHT-MODR1
 DFB UPANDR-MODR1
 DFB DNANDR-MODR1
*
```

HOW IT WORKS

When you have assembled and run the SKETCHER program, you will see that it works slightly differently from the JOYSTIX.BAS program. It allows you to draw on the screen using a game controller, just as the JOYSTIX program does, but its method of reading the controller's trigger button is slightly different. In the JOYSTIX program, the trigger button is used to prevent the printing of a dot on the screen. In the SKETCHER program, depressing the joystick trigger completely erases the screen display.

If you understand how the BLACKBOARD.S and JOYSTIX.BAS programs work, there is not a lot that is new to explain in the SKETCHER program.

It employs the same techniques that were used for plotting on the screen in BLACKBOARD.S, and it reads joysticks in much the same way that they were read in JOYSTIX.BAS. But there are also a couple of new features that may be worth noting.

One of these features is an error-checking subroutine in Lines 358 through 389. This subroutine is quite straightforward, but it is also quite important, since its job is to prevent what is being printed on the screen from extending beyond the boundaries of RAM space that have been designated as screen memory. Error-checking routines such as this one are very important in Assembly language programming, since they keep screen data from overshooting its boundaries and winding up in memory blocks where it has no right to be. When data gets out of hand and goes bounding off into never-never land, it can bring a program to a crashing halt, and you do not want that to happen in your Assembly language programs.

Another important feature of the SKETCHER program is a short routine that appears in Lines 291 to 296. This segment of code is a *relative address modification* routine, and it takes advantage of a strange and wonderful capability of Assembly language and some other programming languages: the ability of a program to modify itself.

The relative address modification routine in the SKETCHER program serves exactly the same purpose as an ON . . . GOTO routine in BASIC. It reads a numeric value—in this case, a value provided by a joystick—and then branches to a routine that has been assigned a corresponding value in a program. This is a fairly sophisticated programming technique, even in BASIC. So, before we examine how it works in Assembly language, let's take a brief look at a somewhat simpler sort of address-modification program. The short segment of code that follows is not a relative address modification, like the one in the SKETCHER program, but a *direct* address modification subroutine that is even more often used in Assembly language programs. Once you understand the principle of direct address modification, it will be easier to grasp the relative address modification technique used in the SKETCHER program.

A SIMPLE ADDRESS-MODIFICATION ROUTINE

LINE NR.	SOURCE-CODE LISTING Label	Code	MEM. ADR.	MACHINE LANGUAGE
100	ADDRESS	LDA VALUE	8040	AD A7 02
101		INC ADDRESS+1	8043	EE 41 80
102		BNE NEXT	8046	D0 03
103		INC ADDRESS+2	8048	EE 42 80
104	NEXT	RTS	804B	60

The above example lists both the source code and the object code of a short address-modification routine, along with the addresses of the memory registers into which the machine code is stored. You should be able to get a clear picture of how the program works by looking at its object code: the machine-language part of the listing.

Look carefully at the routine's object code, and you will see that when the subroutine is first called, the accumulator is loaded with a value which I have labeled—logically enough—VALUE. As you can see in the object-code listing of Line 100, that value is fetched from Memory Register $02A7.

In the next three lines of the routine, something quite extraordinary occurs. As you may be able to recognize by now, the instructions in Lines 101 through 103 are a standard set of instructions for incrementing a 16-bit number, but what number is being incremented here? Well, look again at the object-code part of the program, and you will see that the value which is incremented is whatever 16-bit value happens to be stored in Memory Registers $8041 and $8042, and what value is that? Why, it is the value that follows the mnemonic LDA in Line 100 of the program.

Now take a *very* close look at the object-code listing of this routine, and you will see that the routine has now rewritten itself. The next time the routine is called, Line 100 will load the accumulator not with the value stored in Memory Register $02A7, but by that value *plus one*—and that value will continue to be incremented by one *every time the routine is called.*

Address modification is a very powerful programming technique that is used quite often in high-performance Assembly-language programs. Routines that use address modification are compact and fast-running, and they do not require the use of zero-page memory, which is always in short supply. So a good knowledge of the principles of address modification can be of great value to the Assembly language programmer.

RELATIVE ADDRESS MODIFICATION

Now let's take a look at *relative* address modification: the kind used in the SKETCHER program. As already mentioned, Assembly language programs use relative address modification in much the same way that BASIC programs use ON . . . GOTO routines. In SKETCHER, relative address modification is used to make the program branch to an UP, DOWN, LEFT or RIGHT routine—or a combination thereof—depending on the direction of a joystick.

The address-modification routine in the SKETCHER makes use of a data table that has been placed at the end of the program, in Lines 414 through 424. As you can see, this data table is labeled RELADS (which stands for "relative address"). But notice that the values of the bytes in the RELADS table are not defined as specific values. Instead, each value in the table is defined as the result of a calculation, specifically, the difference between the value in the table and a given line in the SKETCHER program.

Look carefully at the definitions of the bytes in the RELADS table, and you will see that each value in the table has been defined as being equal to the address of one specific joystick-movement routine, *minus* the value of the address of Line 338 of the SKETCHER program, which is labeled MODREL (an abbreviation, in a backwards sort of way, for "relative modification").

Now examine Lines 290 through 297, and you will be able to see how an Assembly language program can actually rewrite part of itself while it is running, using the technique of relative address modification.

In Line 290, when the address-modification routine begins, the direction switch of a game controller has just been read, and the value thus obtained has been stored in the 6510 chip's X register. If the controller's trigger button is currently being pressed, the screen is cleared and the joystick is read again, but if the trigger button has not been pressed, the accumulator is loaded with an 8-bit value that is designed to point to a specific address: namely, the address of one of the joystick-movement routines in Lines 298 through 350 of the SKETCHER program.

Next, examine Lines 291 and 292. In Line 291, an eight-bit value pointing to the desired address is loaded into the accumulator. Then, in Line 292, that value is stored in a given memory address. How is that memory address obtained? Well, it has to be calculated using the label/offset combination MODREL+1.

And just where is MODREL+1? Well, look at Line 293 and you will find the answer:

```
293 MODREL BNE *
```

Now the label of that line, as you can see, is MODREL. From a machine-language point of view, one could also say that MODREL is the label of one specific memory register: the register that holds the machine-language equivalent of the Assembly-label mnemonic BNE. So, if you wanted to assign a label to the address of the Assembly language mnemonic BNE in Line 293 of the SKETCHER program, that label would have to be MODREL.

If the mnemonic BNE is located at the machine language address labeled MODREL, then what is at the machine-language address labeled MODREL+1?

Well, in the source-code listing of the SKETCHER program, MODREL+1 appears to be the address of an asterisk. This may look like a strange way to write a line of code—and it is. Fortunately, in Commodore Assembly language, an asterisk does have a meaning. It is a pseudo-op that is often used to refer to the current content of an assembler's program counter. So, when the SKETCHER program is first assembled, Memory Registers MODREL+1 and MODREL+2 hold nothing but a 16-bit value pointing to their own address.

When the SKETCHER program is executed, however, the contents of MODREL+1 and MODREL+2 are automatically changed. Take a quick look at Line 292, and you will see how. In that line, the contents of MODREL+1 are changed to the value stored in the accumulator—which is, in turn, the value of a specific byte in the data table labeled RELADS. And, as we have seen, each byte in that table is an 8-bit pointer that can be used to calculate the address of a specific joystick-movement routine.

Relative address modification is quite a sophisticated concept, so do not be surprised if it all seems a little foggy at first, but what it boils down to is this. When the SKETCHER program reaches Line 292, a value pointing to the address of a joystick-movement routine is stuffed into an address designated as MODREL + 1. When that happens, the value of the asterisk in Line 293 is replaced by a 16-bit value pointing to the address a specific byte in the RELADS table in Lines 414 through 423. The byte thus obtained is then used to calculate the final address of the desired joystick-movement routine.

If you do not quite understand all of that quite yet, please do not worry. Just type, assemble, and run the SKETCHER program, and observe what the address-modification routine actually does. Once you understand *what* it does, understanding *how* it does it will be much less of a problem.

You have seen only two kinds of address-modification routines in this chapter, but many other kinds of address-modification and data-modification techniques are also used in Assembly language. Advanced Assembly language programmers like them because they are compact and fast-running, and because they do not require the use of Page Zero, where space is always at a premium.

13 Customizing a Character Set

Copying Characters from ROM into RAM

Your Commodore has a terrific built-in character set. From the Commodore 64 keyboard, you can access 512 individual characters, including uppercase ASCII characters, lowercase ASCII characters, reverse-video characters, a host of special characters, and one of the finest sets of keyboard addressable graphics characters in the microcomputer industry.

But sooner or later, if you are like most Commodore Assembly language programmers, you will want to modify your computer's built-in graphics set in one way or another. You may want to add a few special characters that are not on the C64 keyboard. You may want to create a new set of graphics characters for a game you are designing. Or you may want to design a custom type font so you can print fancy text on your computer screen or a dot-matrix printer.

Well, you can do all of these things—and many, many more—if you know how to alter your Commodore's built-in character set. And altering a character set is no problem at all for a good Assembly language programmer.

"ROM-IMAGE" CHARACTERS

As you may recall from Chapter 10, the VIC-II graphics chip in the Commodore 64 generates characters from 4K of character data stored in ROM Addresses $D000 through $DFFF. But the VIC-II chip does not "see" your computer's character data where it actually is. Instead, like a man looking at a mirage, the VIC chip usually looks for—and finds—character data at one of two "ROM image" locations. If the VIC chip has been instructed to get graphics data from Bank 0, it will find a ROM image of character data at Memory Registers $1000 through $2000. If it is using Graphics Bank 2, it will see character data at Memory Registers $9000 through $A000.

As you also may remember from Chapter 10, however, the addresses of these two memory blocks can be changed quite easily. By altering the lower nibble of the VIC-II Memory Control Register, or VMCSB register, at Memory Address $D018, you can tell your computer's VIC-II chip to look for character data in any one of 21 different blocks of memory. All 21 of those memory blocks are listed in the table called "RAM Character Set Starting Addresses" in Chapter 10.

Since you can tell the VIC-II chip exactly where to look for character data, it is easy to alter the Commodore 64's built-in character set, and then use it in its altered form. All you have to do is follow these three steps:

1. Copy your computer's built-in character data from ROM into RAM.
2. Modify the character set that now resides in RAM in any way you wish.
3. Tell your computer's VIC-II chip where the modified set is, so that it can retrieve character data from that character set instead of from the "ROM-image" charcter sets at $1000 (in Bank 0) or $9000 (in Bank 2).

ONE SMALL PROBLEM

That is about all there is to customizing a character set—except for one small hitch, which was mentioned in passing in Chapter 10. The hitch is that Memory Addresses $D000 through $DFFF have two different functions in the Commodore 64. This block of ROM is shared by the C64's VIC-II graphics chip and the I/O drivers that are part of the computer's operating system. During the VIC-II's screen-refresh cycle, when the chip needs access to character data so that it can create a screen display, Addresses $D000 through $DFFF are used to hold the character-generator data needed by the VIC processor. But as soon as the VIC-II has completed its screen-refreshing operation, the character data required by the VIC-II chip is bank-switched out of this character block, and a set of registers which the Commodore 64 needs for the operation of I/O devices are switched in. The C64 operating system then takes care of certain I/O housekeeping chores. When those are done, character data is switched back into the $D000-$DFFF memory block. And so on.

Obviously, all of this switching in and out of character data could cause serious problems during the process of copying a character set from ROM to RAM. If the character data being copied were switched right out of your computer's memory during the copying operation, the whole operation could be ruined.

AN EASY SOLUTION

Fortunately, there is an easy way to prevent this kind of disaster from happening. As you may remember from Chapter 10, a special memory register in the Commodore 64—Register $0001, often called the R6510 register—can be used to determine whether character data or I/O data is switched into ROM at Addresses $D000 through $DFFF. If Bit 2 of the R6510 register is set, then I/O data will be stored in Registers $D000 through $DFFF. If Bit 2 of the R6510 register is clear, then the $D000-$DFFF memory block will hold character-generator data.

Another special memory register—the C2DDRA register, at Memory Address $DD02—is often used in conjunction with the R6510 register. The C2DDRA is a "data-direction" register that is used to determine the direction of data flow to and from I/O devices. If Bits 0 and 1 of the C2DDRA register are set, then any data that appears on lines going to peripheral devices will be regarded as output data, not input data, and that is the way things should be during a character-copying operation. Otherwise, data generated by an I/O device might be accepted as input data, and might interfere with the RAM-copying process.

As an additional safety measure, the keyboard of the Commodore 64 can be turned off while a character set is being copied into RAM. You can turn off the C64 keyboard by storing the value $FE (binary 1111 1110) into Register CIACRE (VIC-II Control Register A), at Memory Address $DC0E.

Now here is a short program that will copy all 512 characters of the Commodore 64 character set from ROM into RAM. It does not alter any of the characters; it just copies them, and tells the VIC-II chip where to find them. Examine the program, and you will see that it does everything explained so far in this chapter, and then some.

First, the program makes sure that all I/O lines are designated as outputs, and then it makes sure that the C64 will be left in its uppercase mode until the copying operation is complete. Next, a sufficient amount of free RAM is set aside to hold the copied character set (this step is needed only if the characters that are being copied will be used by a BASIC program). The C64 keyboard is then turned off, and character ROM is switched into Memory Addresses $D000 through $DFFF.

After all of that is done, the C64 character set is copied into RAM using a standard block-move algorithm. Then I/O is switched back in, the keyboard is turned back on, and the VIC chip is told (via the VMCSB register) where its character-generator data can now be found.

COPYING A CHARACTER SET FROM ROM TO RAM

```
*
* MOVECHRS
*
 ORG $8000
*
R6510 EQU $0001
NEWADR EQU $3000
CHRBAS EQU $D000
CIACRE EQU $DC0E
C2DDRA EQU $DD02
VMCSB EQU $D018
CHROUT EQU $FFD2
*
FRETOP EQU $0034
MEMSIZ EQU $0038
*
```

```
TABLEN EQU $1000
MVSRCE EQU $FB
MVDEST EQU MVSRCE+2
*
LENPTR EQU $200
*
* SET CIA BITS TO OUTPUTS
*
 LDA C2DDRA
 ORA #3
 STA C2DDRA
*
* USE UPPER-CASE CHARACTER SET
*
 LDA #142
 JSR CHROUT
*
* CLEAR RAM FOR CHR MEMORY
*
 LDA #48
 STA FRETOP
 STA MEMSIZ
*
* TURN OFF KB INTERRUPT TIMER
*
 LDA CIACRE
 AND #$FE
 STA CIACRE
*
* SWITCH I/O OFF, CHAR ROM ON
*
 LDA R6510
 AND #$FB
 STA R6510
*
* COPY CHARACTERS INTO RAM
*
 LDA #<CHRBAS
 STA MVSRCE
 LDA #>CHRBAS
 STA MVSRCE+1
*
 LDA #<NEWADR
 STA MVDEST
 LDA #>NEWADR
 STA MVDEST+1
*
 LDA #<TABLEN
 STA LENPTR
 LDA #>TABLEN
 STA LENPTR+1
*
* START MOVE
*
 LDY #0
 LDX LENPTR+1
 BEQ MVPART
MVPAGE LDA (MVSRCE),Y
 STA (MVDEST),Y
 INY
 BNE MVPAGE
 INC MVSRCE+1
 INC MVDEST+1
 DEX
 BNE MVPAGE
MVPART LDX LENPTR
 BEQ MVEXIT
MVLAST LDA (MVSRCE),Y
 STA (MVDEST),Y
 INY
 DEX
 BNE MVLAST
MVEXIT
*
* SWITCH I/O BACK IN
*
 LDA R6510
 ORA #4
 STA R6510
*
* TURN KEYBOARD BACK ON
*
 LDA CIACRE
 ORA #1
 STA CIACRE
*
* SET VIC MEMORY CONTROL REGISTER
*
 CLC
 LDA VMCSB
 AND #$F0
 ADC #$0C
 STA VMCSB
*
 RTS
*
```

```
 END
*
```

WHAT NEXT?

Once a character set has been copied from ROM to RAM, any character that it contains can be modified in any way desired. Here is another program—actually an expanded version of the previous program—that demonstrates how a character can be altered once it has been moved into RAM. It makes use of a short and simple data-moving routine to turn the letter Z into a man waving his arms. Type the program, assemble it, and run it—and then start typing on your computer keyboard. If everything goes as it should, every Z that you type will show up on your screen not as a Z, but as a little man.

MODIFYING A CHARACTER

```
*
* MYCHRS
*
 ORG $8000
*
R6510 EQU $0001
NEWADR EQU $3000
CHRBAS EQU $D000
CIACRE EQU $DC0E
C2DDRA EQU $DD02
VMCSB EQU $D018
*
FRETOP EQU $0034
MEMSIZ EQU $0038
*
TABLEN EQU $1000
MVSRCE EQU $FB
MVDEST EQU MVSRCE+2
CHRADR EQU MVDEST
*
LENPTR EQU $200
RAMCHR EQU LENPTR+2
*
 JMP START
*
SHAPE HEX 18,DB,42,7E,18,7E,66,E7 ;A MAN WAVING HIS ARMS
* SET CIA BITS TO OUTPUTS
*
START LDA C2DDRA
 ORA #3
 STA C2DDRA
```

```
*
* CLEAR RAM FOR CHR MEMORY
*
 LDA #48
 STA FRETOP
 STA MEMSIZ
*
* TURN OFF KB INTERRUPT TIMER
*
 LDA CIACRE
 AND #$FE
 STA CIACRE
*
* SWITCH I/O OFF, CHAR ROM ON
*
 LDA R6510
 AND #$FB
 STA R6510
*
* COPY CHARACTERS INTO RAM
*
 LDA #<CHRBAS
 STA MVSRCE
 LDA #>CHRBAS
 STA MVSRCE+1
*
 LDA #<NEWADR
 STA MVDEST
 LDA #>NEWADR
 STA MVDEST+1
*
 LDA #<TABLEN
 STA LENPTR
 LDA #>TABLEN
 STA LENPTR+1
*
* START MOVE
*
 LDY #0
 LDX LENPTR+1
 BEQ MVPART
MVPAGE LDA (MVSRCE),Y
 STA (MVDEST),Y
 INY
 BNE MVPAGE
 INC MVSRCE+1
 INC MVDEST+1
 DEX
 BNE MVPAGE
MVPART LDX LENPTR
 BEQ MVEXIT
MVLAST LDA (MVSRCE),Y
 STA (MVDEST),Y
 INY
 DEX
 BNE MVLAST
MVEXIT
*
* SWITCH I/O BACK IN
*
 LDA R6510
 ORA #4
 STA R6510
*
* TURN TIMER BACK ON
*
 LDA CIACRE
 ORA #1
 STA CIACRE
*
* SET VIC MEMORY CONTROL REGISTER
*
 CLC
 LDA VMCSB
 AND #$F0
 ADC #$0C
 STA VMCSB
*
* NOW WE ALTER A CHARACTER (Z)
*
 LDA #26 ;Z
 STA RAMCHR
*
* CALCULATE RAMCHR'S ADDRESS
*
 LDA #0
 STA RAMCHR+1
 LDA RAMCHR
 CLC
 ASL A
 ROL RAMCHR+1
 ASL A
 ROL RAMCHR+1
 ASL A
 ROL RAMCHR+1
 STA RAMCHR
```

```
128 *
129  CLC
130  LDA RAMCHR
131  ADC #<NEWADR
132  STA CHRADR
133  LDA RAMCHR+1
134  ADC #>NEWADR
135  STA CHRADR+1
136 *
137 * NOW WE CHANGE THE CHARACTER
138 *
139  LDY #0
140 DOSHAPE LDA SHAPE,Y
141  STA (CHRADR),Y
142  INY
143  CPY #9
144  BCC DOSHAPE
145 *
146  RTS
147 *
148  END
149 *
```

MIXING TEXT AND HIGH-RESOLUTION GRAPHICS

Once you have copied a character set into RAM, another interesting thing you can do with it is use it to print text characters on a high-resolution, bit-mapped screen. Here is a program that does just that; it copies a character set into RAM, and then employs a bit-mapping routine to print a character from that set (namely, an "A") on a high-resolution screen. One possible use of this program, therefore, is to mix text and high-resolution graphics.

As you will quickly see, this program—which I call SHOWCHRS—contains a number of new routines, along with a few other routines that we have encountered in previous programs.

Here is how the SHOWCHRS program works. First it copies a character set from ROM to RAM, and then it plots a character on a high-resolution screen using the same kind of high-resolution plotting routine that you encountered in the BLACKBOARD and SKETCHER programs in Chapter 11. So, if you have been typing and running the Assembly language programs in this book, you can now save yourself some typing. Just use your assembler's MERGE or APPEND utility (if it has one) to tack a few old, familiar programs together. Then you can use your assembler's editor to make whatever modifications may be needed to fashion some of your old programs into this new one.

One of the program's new routines is the one in Lines 225 through 246. This routine calculates the starting address of the data needed to form a character by going through a series of mathematical operations. The routine takes the character's ASCII code, multiplies it by eight (since it takes eight bytes to draw a character), and then adds the result of this calculation to the starting address of a character set that has been copied into ROM. The final result of this process is the starting address of the RAM data needed to generate the character.

Another new routine is the one that extends from Lines 367 to 440. This routine uses a nested loop to define the shape of a character, employing the same kind of plotting subroutine that was used in an earlier chapter to draw a character on a bit-mapped screen.

One point worth noting in the program's character-printing routine is the series of stack-manipulation instructions in Lines 404 through 416. These instructions are used to save the contents of the 6510 chip's X and Y registers on the stack while dots are being plotted on the screen. The contents of the X and Y registers have to be saved while the plotting subroutine is in use because both the shape-defining and dot-plotting routines in the SHOWCHRS program make use of these registers. So the contents of the registers have to be saved each time a dot-plotting routine is called, then restored each time a dot-plotting routine is completed.

```
THE A's HAVE IT

  1 *
  2 * SHOWCHAR
  3 *
  4  ORG $8000
  5 *
  6 COLOR EQU $10
  7 COLMAP EQU $8400
  8 BASE EQU $A000
  9 SCROLY EQU $D011
 10 CI2PRA EQU $DD00
 11 C2DDRA EQU $DD02
 12 VMCSB EQU $D018
 13 *
 14 HMAX EQU 320
 15 HMID EQU 160-4 *
 16 VMID EQU 100-4 *
 17 *
 18 SCRLEN EQU 8000
 19 MAPLEN EQU 1000
 20 *
 21 TEMPA EQU $FB
 22 TEMPB EQU TEMPA+2
 23 *
 24 TABPTR EQU TEMPA
 25 TABSIZ EQU $02A7
 26 *
 27 HPSN EQU TABSIZ+2
```

```
VPSN EQU HPSN+2
CHAR EQU VPSN+1
ROW EQU CHAR+1
LINE EQU ROW+1
BYTE EQU LINE+1
BITT EQU BYTE+2
*
MPRL EQU BITT+1
MPRH EQU MPRL+1
MPDL EQU MPRH+1
MPDH EQU MPDL+1
PRODL EQU MPDH+1
PRODH EQU PRODL+1
*
FILVAL EQU PRODH+1
*
R6510 EQU $0001
NEWADR EQU $8800
CHRBAS EQU $D000
CIACRE EQU $DC0E
*
TABLEN EQU $800
*
MVSRCE EQU $61
MVDEST EQU MVSRCE+2
BYTPTR EQU MVDEST+2
*
LENPTR EQU $9000
CHCODE EQU LENPTR+2
HPTR EQU CHCODE+2
ONEBYT EQU HPTR+1
COUNT EQU ONEBYT+2
*
 JMP START
*
* BLOCK FILL ROUTINE
*
BLKFIL LDA FILVAL
 LDX TABSIZ+1
 BEQ PARTPG
 LDY #0
FULLPG STA (TABPTR),Y
 INY
 BNE FULLPG
 INC TABPTR+1
 DEX
 BNE FULLPG
PARTPG LDX TABSIZ
```

```
 BEQ FINI
 LDY #0
PARTLP STA (TABPTR),Y
 INY
 DEX
 BNE PARTLP
FINI RTS
*
* 16-BIT MULTIPLICATION ROUTINE
*
MULT16 LDA #0
 STA PRODL
 STA PRODH
 LDX #17
 CLC
MULT ROR PRODH
 ROR PRODL
 ROR MPRH
 ROR MPRL
 BCC CTDOWN
 CLC
 LDA MPDL
 ADC PRODL
 STA PRODL
 LDA MPDH
 ADC PRODH
 STA PRODH
CTDOWN DEX
 BNE MULT
 RTS
*
* PLOT ROUTINE
*
* ROW=VPSN/8 (8-BIT DIVIDE)
*
PLOT LDA VPSN
 LSR A
 LSR A
 LSR A
 STA ROW
*
* CHAR=HPSN/8 (16-BIT DIVIDE)
*
 LDA HPSN
 STA TEMPA
 LDA HPSN+1
 STA TEMPA+1
 LDX #3
```

```
124 DLOOP LSR TEMPA+1
125  ROR TEMPA
126  DEX
127  BNE DLOOP
128  LDA TEMPA
129  STA CHAR
130 *
131 * LINE=VPSN AND 7
132 *
133  LDA VPSN
134  AND #7
135  STA LINE
136 *
137 * BITT=7-(HPSN AND 7)
138 *
139  LDA HPSN
140  AND #7
141  STA BITT
142  SEC
143  LDA #7
144  SBC BITT
145  STA BITT
146 *
147 * BYTE=BASE+ROW*HMAX+8*CHAR+LINE
148 *
149 * FIRST MULTIPLY ROW * HMAX
150 *
151  LDA ROW
152  STA MPRL
153  LDA #0
154  STA MPRH
155  LDA #<HMAX
156  STA MPDL
157  LDA #>HMAX
158  STA MPDH
159  JSR MULT16
160  LDA MPRL
161  STA TEMPA
162  LDA MPRL+1
163  STA TEMPA+1
164 *
165 * ADD PRODUCT TO BASE
166 *
167  CLC
168  LDA #<BASE
169  ADC TEMPA
170  STA TEMPA
171  LDA #>BASE
```

```
 ADC TEMPA+1
 STA TEMPA+1
*
* MULTIPLY 8 * CHAR
*
 LDA #8
 STA MPRL
 LDA #0
 STA MPRH
 LDA CHAR
 STA MPDL
 LDA #0
 STA MPDH
 JSR MULT16
 LDA MPRL
 STA TEMPB
 LDA MPRH
 STA TEMPB+1
*
* ADD LINE
*
 CLC
 LDA TEMPB
 ADC LINE
 STA TEMPB
 LDA TEMPB+1
 ADC 0
 STA TEMPB+1
*
* BYTE = TEMPA + TEMPB
*
 CLC
 LDA TEMPA
 ADC TEMPB
 STA TEMPB
 LDA TEMPA+1
 ADC TEMPB+1
 STA TEMPB+1
*
* POKE BYTE,PEEK(BYTE)OR2^BIT
*
 LDX BITT
 INX
 LDA #0
 SEC
SQUARE ROL
 DEX
 BNE SQUARE
 LDY #0
 ORA (TEMPB),Y
 STA (TEMPB),Y
 RTS
*
* CALCULATE CHCODE'S ADDRESS
*
GETADR LDA #0
 STA CHCODE+1
 LDA CHCODE
 CLC
 ASL A
 ROL CHCODE+1
 ASL A
 ROL CHCODE+1
 ASL A
 ROL CHCODE+1
 STA CHCODE
*
 CLC
 LDA CHCODE
 ADC #<NEWADR
 STA BYTPTR
 LDA CHCODE+1
 ADC #>NEWADR
 STA BYTPTR+1
 RTS
*
*
* MAIN ROUTINE STARTS HERE
*
START LDA VMCSB
 ORA #8
 STA VMCSB
*
 LDA SCROLY
 ORA #32
 STA SCROLY
*
* USE BANK 2
*
 LDA C2DDRA
 ORA #3
 STA C2DDRA
*
 LDA CI2PRA
 AND #252
 ORA #1 ;BANK 2
 STA CI2PRA
*
* CLEAR BIT MAP
*
 LDA #0
 STA FILVAL
 LDA #<BASE
 STA TABPTR
 LDA #>BASE
 STA TABPTR+1
 LDA #<SCRLEN
 STA TABSIZ
 LDA #>SCRLEN
 STA TABSIZ+1
 JSR BLKFIL
*
* SET BKG AND LINE COLORS
*
 LDA #COLOR
 STA FILVAL
 LDA #<COLMAP
 STA TABPTR
 LDA #>COLMAP
 STA TABPTR+1
 LDA #<MAPLEN
 STA TABSIZ
 LDA #>MAPLEN
 STA TABSIZ+1
 JSR BLKFIL
*
* TURN OFF KB INTERRUPT TIMER
*
MVCHRS LDA CIACRE
 AND #$FE
 STA CIACRE
*
* SWITCH BASIC OUT
*
 LDA R6510
 AND #$FE
 STA R6510
*
* SWITCH I/O OFF, CHAR ROM ON
*
 LDA R6510
 AND #$FB
 STA R6510
*
```

```
316 * COPY CHARACTERS INTO RAM
317 *
318  LDA #<CHRBAS
319  STA MVSRCE
320  LDA #>CHRBAS
321  STA MVSRCE+1
322 *
323  LDA #<NEWADR
324  STA MVDEST
325  LDA #>NEWADR
326  STA MVDEST+1
327 *
328  LDA #<TABLEN
329  STA LENPTR
330  LDA #>TABLEN
331  STA LENPTR+1
332 *
333 * START MOVE
334 *
335  LDY #0
336  LDX LENPTR+1
337  BEQ MVPART
338 MVPAGE LDA (MVSRCE),Y
339  STA (MVDEST),Y
340  INY
341  BNE MVPAGE
342  INC MVSRCE+1
343  INC MVDEST+1
344  DEX
345  BNE MVPAGE
346 MVPART LDX LENPTR
347  BEQ MVEXIT
348 MVLAST LDA (MVSRCE),Y
349  STA (MVDEST),Y
350  INY
351  DEX
352  BNE MVLAST
353 MVEXIT
354 *
355 * SWITCH I/O BACK IN
356 *
357  LDA R6510
358  ORA #4
359  STA R6510
360 *
361 * TURN TIMER BACK ON
362 *
363  LDA CIACRE
```

```
 ORA #1
 STA CIACRE
*
* DRAW A CHARACTER
*
 LDA #<HMID
 STA HPSN
 STA HPTR
 LDA #>HMID
 STA HPSN+1
 STA HPTR+1
 LDA #VMID
 STA VPSN
*
 LDA #1 ;'A'
 STA CHCODE
 JSR GETADR
*
* A NESTED LOOP:
*
* (X IS THE OUTSIDE LOOP)
*
 LDX #8
SETBIT LDY #0
 LDA (BYTPTR),Y
 STA ONEBYT
*
* THE INSIDE LOOP:
*
* (Y IS ZERO AT START)
*
RSHIFT LDA ONEBYT
 ASL A
 STA ONEBYT
 BCC NOSHOW
*
* DISPLAY BIT
*
* SAVE X AND Y REGISTERS
*
 TXA
 PHA
 TYA
 PHA
*
 JSR PLOT
*
* RETRIEVE X AND Y REGISTERS
```

```
*
 PLA
 TAY
 PLA
 TAX
*
NOSHOW INC HPSN
 BNE LEAP
 INC HPSN+1
*
LEAP INY
 CPY #8
 BCC RSHIFT
*
 INC VPSN
*
 LDA HPTR
 STA HPSN
 LDA HPTR+1
 STA HPSN+1
*
 INC BYTPTR
 BNE OKMSB
 INC BYTPTR+1
OKMSB
*
 DEX
 BNE SETBIT
*
INF JMP INF
*
 END
```

HEADLINE CHARACTERS—THE EASY WAY

If you like to write programs in high-resolution graphics, you can see how a program like the one above might come in handy. Now that you have it on a disk, you can use it at any time you like to print text anywhere on a high-resolution screen.

Now we are going to make the SHOWCHRS program even better. With a few minor changes, you can make the program print headline-size characters—twice as wide and twice as high as ordinary text characters—on a high-resolution screen. Best of all, you can do that using the Commodore 64's standard built-in character set.

I call the expanded SHOWCHRS program BIGCHRS. With the BIGCHRS program, you can print giant-size text in bit-mapped graphics—

without going through the trouble of designing a single character of your own.

The BIGCHRS program works just like SHOWCHRS, except that it prints characters that are twice as wide and twice as high as normal text characters. It accomplishes this feat by expanding each dot in a standard text character to four dots—two going across the screen, and two going down—with some tricky X and Y loops that I will let you figure out for yourself.

The BIGCRS program is based on the SHOWCHRS program. So you can save yourself some typing by loading SHOWCHRS and then carefully editing it to look like BIGCHRS.S, in Appendix B.

Once you have the BIGCHRS program typed, assembled, and executed, you will be ready to expand it into a program that prints headlines in large type on a computer screen. In the next chapter, you will learn how to do just that—and you will also learn how to program sprite graphics in Assembly language.

14 Programming Sprites in Assembly Language

Animating Sprites on the Screen

Once you know how to print large characters on a high-resolution screen, it is not difficult to write programs that will print multi-character messages in large type. The next (and last) program in this chapter will do just that, and it will do it using principles that you are probably very familiar with by now. If you assembled and ran the message-printing programs in earlier chapters of this book—the programs called RESPONSE and THE NAME GAME—then you already know how to print messages on a computer screen. As the following program will demonstrate, the same kinds of techniques that were used to print the messages in those programs can also be used to print messages in big type.

But the program we will be looking at next is not just a patchwork of old routines. It also contains something new: sprite graphics.

If you have ever worked with sprites in BASIC, you will probably be pleased to learn that it is actually easier to program sprites using Assembly language than it is to create and use them in BASIC. That is because sprites are programmed using many kinds of bit and byte manipulations that are much easier to manage using binary and hexadecimal numbers than they are using decimal numbers. By the time you finish this chapter, you will see why.

Sprites, as you may know, are graphics characters that can be created, colored, and animated quite easily, and can be moved around completely independently of anything else on a computer screen. Using ordinary programming techniques, up to eight sprites can be displayed on a screen at once. These eight sprites are usually numbered 0 through 7.

Sprites are made of tiny dots, just like programmable text characters. Like programmable characters, they can be created using standard bit-mapping techniques, but sprites are larger than text characters. A sprite can be up to 24 horizontal screen dots wide and up to 21 vertical screen dots dots high.

A sprite can be displayed in any of the 16 colors that are available to the VIC-II chip. It is also possible to create multicolored sprites. Instructions for programming multicolored sprites will not be provided, but can be found in the *Commodore 64 Programmer's Reference Guide* and a number of other books listed in the bibliography appendix in this book.

Sprites can also be expanded to twice their normal width and twice their normal height, or four times their standard size. The sprite used in this chapter will be an expanded one.

HOW SPRITES ARE DRAWN

As previously pointed out, a sprite can measure up to 21 dots (or bytes) wide, and up to 24 dots (or bits) high, or a total of 504 dots. Figure 14-1 shows a sprite bit map.

A sprite bit can also be pictured as a byte map—a matrix that measures three bytes wide by 21 bytes high, for a total of 63 bytes. Actually, the bytes that make up a sprite are in consecutive order in RAM, starting with the byte in the upper left-hand corner and ending with the one in the lower right-hand corner, the 63rd byte. But when a sprite appears on the screen, it looks more like Figure 14-2.

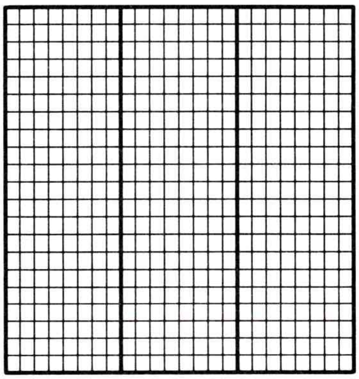

Figure 14-1 Sprite Bit Map

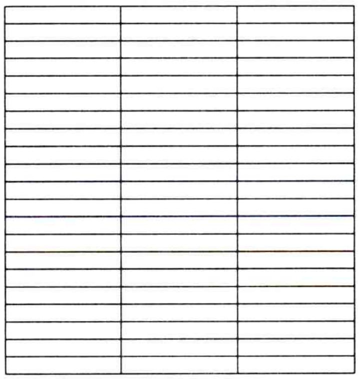

Figure 14-2 Sprite Byte Map

HOW SPRITES ARE PROGRAMMED

Although it takes only 63 bytes to form a sprite, each sprite consumes 64 bytes in RAM. The 64th byte of each sprite map is used to mark the end of its location in memory.

Sprites can be placed anywhere in free RAM, and a special pointer is provided to mark the location for each sprite. Each sprite pointer is one byte long, so it takes eight bytes of RAM to hold the eight pointers that are needed to address the Commodore 64's eight sprites. These eight pointers are always the last eight bytes of whatever block of RAM has been designated as screen memory. When the location of screen memory is moved, the addresses of the C64's eight sprite pointers also change. But it is always easy to find them, since they always take up the last eight bytes of whatever block of RAM is being used as screen memory.

A one-byte value is all that is ever needed to define the starting address of a sprite map, since sprites always fall into whatever 16K bank of memory is currently accessible to the VIC-II chip. That means that a sprite pointer is actually an offset that must be added to the starting address of the graphics bank currently in use to determine the starting address of the bit map that is to be used to form the sprite.

When the Commodore 64 is first turned on, its VIC-II chip is set to retrieve graphics information from Bank 0, and to get its screen map from Memory Registers $0400 through $0800 (1024 through 2048 in decimal notation). At power-up time, therefore, the default address of the first sprite pointer, or Sprite Pointer 0, is $07FB (1020 in decimal notation), and the next eight bytes in RAM are the pointers for Sprites 1 through 7. So the default addresses of the pointers for the C64's eight sprite pointers are Memory Addresses $07FB through $07FF—the last eight bytes in the block of RAM designated as screen memory.

To find the data that it needs to display a sprite, all the Commodore 64 has to do is look at the 8-bit value stored in the appropriate sprite pointer. When that value is added to the address of the graphics bank currently in use, the result will be the address of the bit map that must be used to define the sprite.

TURNING SPRITES ON AND OFF

Before a sprite can be displayed, it must be turned on. Sprites are turned on and off with a sprite enable register (abbreviated SPENA) situated at Memory Address $D015. Each bit of the SPENA register is associated with one sprite; Bit 0 is used to turn Sprite 0 on and off, Bit 1 is used to control Sprite 1, and so on. If the bit associated with a sprite is set, then the sprite is enabled. If the bit is not set, then the sprite is not enabled and cannot be used.

POSITIONING SPRITES

Each of the C64's eight sprites has two position registers: an X position register that is used to determine its horizontal placement on the screen, and a Y position register that is used to determine its vertical position. These registers are abbreviated SP0X through SP7X and SP0Y through SP7Y. In addition, there is a special most significant X position register (abbreviated MSIGX) that is used to designate the horizontal positions of all eight sprites. This register is needed because a sprite can be placed in 512 possible horizontal screen positions—too many positions for an eight-bit register to keep track of. If a sprite is to be placed in a position that can be stored as a value in an 8-bit register—that is, in a position with a value of less than 255—then the MSIGX register is not used, but if the horizontal position of a sprite has a value of more than 255, a bit in the MSIGX register is set. Each bit of the MSIGX register equates to the number of a Sprite; Bit 0 is used for Sprite 0, Bit 1 is used for Sprite 1, and so on.

There is no MSIGY register because there is no need for one. A sprite can be placed in only 256 vertical positions, so only one 8-bit register per sprite is needed to handle the vertical positioning of sprites on the C64's screen.

When you store values in a horizontal or vertical position sprite register, that value is used to determine the position of the *upper left-hand corner* of the sprite, but storing a value in a horizontal or vertical position register does not ensure that a sprite will be displayed on the screen. Of the 512 possible horizontal positions of a sprite, only Positions 24 through 343 are visible on the screen. Of the 255 vertical positions that are available, only Positions 50 through 249 are actually visible on the screen. It is therefore quite easy to make a sprite disappear; all you have to do is store the value of an offscreen position in its horizontal or vertical position register.

Table 14-1 lists the locations of all of the sprite position registers used by the Commodore 64.

Table 14-1 SPRITE POSITION REGISTERS

HEXADECIMAL ADDRESS	POSITION REGISTER	HEXADECIMAL ADDRESS	POSITION REGISTER
D000	SP0X	D008	SP4X
D001	SP0Y	D009	SP4Y
D002	SP1X	D00A	SP5X
D003	SP1Y	D00B	SP5Y
D004	SP2X	D00C	SP6X
D005	SP2Y	D00D	SP6Y
D006	SP3X	D00E	SP7X
D007	SP3Y	D00F	SP7Y

D010—MSIGX (Most Significant X Position Register)

SELECTING COLORS FOR SPRITES

In addition to its two position registers, each sprite also has a color register. The color register for Sprite 0 is at Memory Address $D027, and the addresses of the color registers for the other seven sprites follow in consecutive order. The color address for Sprite 7 is therefore at Memory Address $D02E.

To select the color of a sprite, all you have to do is store the standard value of one of the Commodore 64's 16 colors in that sprite's color register. Every bit that is set on the sprite's bit map will then be displayed in the selected color. Every dot that has a value of 0 will be transparent, and will not cover up anything that is beneath it on the screen.

EXPANDING SPRITES

As previously mentioned, a sprite normally measures 24 horizontal screen dots wide by 21 vertical screen dots high, but by using two special registers called XXPAND and YXPAND, a sprite can be expanded to twice its normal width, twice its normal height, or both. The XXPAND register is at Memory Address $D01D, and the YXPAND register is at $D017. Each bit in each register corresponds to a sprite number, with Bit 0 controlling the size of Sprite 0, Bit 1 controlling the size of Sprite 1, and so on.

ON WITH THE PROGRAM

Now we are ready to take a look at an Assembly language program that makes use of high-resolution graphics, an alternate character set, a large-type printing routine, and an animated, expanded sprite routine. The program copies a character set from ROM into RAM and then prints a message on the screen in large type. It then clears a bit map for Sprite 0, copies some data into the bit map from the character set in RAM, and places an expanded sprite in an area out of viewing range at the top of the screen. Next, the sprite descends into viewing range, and maintains a slow descent until it reaches a predetermined position. Then it stops, and becomes part of the message displayed on the screen.

Here is the program:

```
DESCENT OF A SPRITE

*
* SPRITE
*
 ORG $9000 *
*
COLOR EQU $E0
*
TABLEN EQU $800
MAPLEN EQU 1000
SCRLEN EQU 8000
SP0ADR EQU $8000
COLMAP EQU $8400
NEWADR EQU $8800
*
```

```
SPRPTR EQU $87F8
SPENA EQU $D015
SP0COL EQU $D027
SP0X EQU $D000
SP0Y EQU $D001
MSIGX EQU $D010
YXPAND EQU $D017
XXPAND EQU $D01D
*
HMAX EQU 320
VMID EQU 100-8
*
R6510 EQU $0001
BASE EQU $A000
CHRBAS EQU $D000
SCROLY EQU $D011
VMCSB EQU $D018
BORDER EQU $D020
CIACRE EQU $DC0E
CI2PRA EQU $DD00
C2DDRA EQU $DD02
*
TEMPA EQU $FB
TEMPB EQU TEMPA+2
TABPTR EQU TEMPA
*
MVSRCE EQU $61
MVDEST EQU MVSRCE+2
BYTPTR EQU MVDEST+2
*
TABSIZ EQU $02A7
*
HPSN EQU TABSIZ+2
VPSN EQU HPSN+2
CHAR EQU VPSN+1
ROW EQU CHAR+1
LINE EQU ROW+1
BYTE EQU LINE+1
BITT EQU BYTE+2
*
MPRL EQU BITT+1
MPRH EQU MPRL+1
MPDL EQU MPRH+1
MPDH EQU MPDL+1
PRODL EQU MPDH+1
PRODH EQU PRODL+1
FILVAL EQU PRODH+1
LENPTR EQU FILVAL+1
```

```
CHCODE EQU LENPTR+2
HPTR EQU CHCODE+2
VPTR EQU HPTR+2
ONEBYT EQU VPTR+1
COUNT EQU ONEBYT+2
LTTR EQU COUNT+1
*
 JMP START
*
TEXT DFB 9,32,32,32,13,25,32,3
 DFB 15,13,13,15,4,15,18,5
 DFB 32,54,52,0
*
* BLOCK FILL ROUTINE
*
BLKFIL LDA FILVAL
 LDX TABSIZ+1
 BEQ PARTPG
 LDY #0
FULLPG STA (TABPTR),Y
 INY
 BNE FULLPG
 INC TABPTR+1
 DEX
 BNE FULLPG
PARTPG LDX TABSIZ
 BEQ FINI
 LDY #0
PARTLP STA (TABPTR),Y
 INY
 DEX
 BNE PARTLP
FINI RTS
*
* 16-BIT MULTIPLICATION ROUTINE
*
MULT16 LDA #0
 STA PRODL
 STA PRODH
 LDX #17
 CLC
MULT ROR PRODH
 ROR PRODL
 ROR MPRH
 ROR MPRL
 BCC CTDOWN
 CLC
 LDA MPDL
 ADC PRODL
 STA PRODL
 LDA MPDH
 ADC PRODH
 STA PRODH
CTDOWN DEX
 BNE MULT
 RTS
*
* PLOT ROUTINE
*
* ROW=VPSN/8 (8-BIT DIVIDE)
*
PLOT LDA VPSN
 LSR A
 LSR A
 LSR A
 STA ROW
*
* CHAR=HPSN/8 (16-BIT DIVIDE)
*
 LDA HPSN
 STA TEMPA
 LDA HPSN+1
 STA TEMPA+1
 LDX #3
DLOOP LSR TEMPA+1
 ROR TEMPA
 DEX
 BNE DLOOP
 LDA TEMPA
 STA CHAR
*
* LINE=VPSN AND 7
*
 LDA VPSN
 AND #7
 STA LINE
*
* BITT=7-(HPSN AND 7)
*
 LDA HPSN
 AND #7
 STA BITT
 SEC
 LDA #7
 SBC BITT
 STA BITT
```

```
*
* BYTE=BASE+ROW*HMAX+8*CHAR+LINE
*
* FIRST MULTIPLY ROW * HMAX
*
 LDA ROW
 STA MPRL
 LDA #0
 STA MPRH
 LDA #<HMAX
 STA MPDL
 LDA #>HMAX
 STA MPDH
 JSR MULT16
 LDA MPRL
 STA TEMPA
 LDA MPRL+1
 STA TEMPA+1
*
* ADD PRODUCT TO BASE
*
 CLC
 LDA #<BASE
 ADC TEMPA
 STA TEMPA
 LDA #>BASE
 ADC TEMPA+1
 STA TEMPA+1
*
* MULTIPLY 8 * CHAR
*
 LDA #8
 STA MPRL
 LDA #0
 STA MPRH
 LDA CHAR
 STA MPDL
 LDA #0
 STA MPDH
 JSR MULT16
 LDA MPRL
 STA TEMPB
 LDA MPRH
 STA TEMPB+1
*
* ADD LINE
*
 CLC
 LDA TEMPB
 ADC LINE
 STA TEMPB
 LDA TEMPB+1
 ADC 0
 STA TEMPB+1
*
* BYTE = TEMPA + TEMPB
*
 CLC
 LDA TEMPA
 ADC TEMPB
 STA TEMPB
 LDA TEMPA+1
 ADC TEMPB+1
 STA TEMPB+1
*
* POKE BYTE,PEEK(BYTE)OR2^BIT
*
 LDX BITT
 INX
 LDA #0
 SEC
SQUARE ROL
 DEX
 BNE SQUARE
 LDY #0
 ORA (TEMPB),Y
 STA (TEMPB),Y
 RTS
*
* CALCULATE CHCODE'S ADDRESS
*
GETADR LDA #0
 STA CHCODE+1
 LDA CHCODE
 CLC
 ASL A
 ROL CHCODE+1
 ASL A
 ROL CHCODE+1
 ASL A
 ROL CHCODE+1
 STA CHCODE
*
 CLC
 LDA CHCODE
 ADC #<NEWADR
 STA BYTPTR
 LDA CHCODE+1
 ADC #>NEWADR
 STA BYTPTR+1
 RTS
*
* DRAW A CHARACTER
*
DRAWCH LDA LTTR
 STA CHCODE
 JSR GETADR
*
* A NESTED LOOP:
*
* (X IS THE OUTSIDE LOOP)
*
 LDX #8
*
* SET UP COUNTER FOR 2 VERT LINES
*
SETLIN LDA #2
 STA COUNT
*
DRAWLN LDY #0
 LDA (BYTPTR),Y
 STA ONEBYT
*
* THE INSIDE LOOP:
*
* (Y IS ZERO AT START)
*
RSHIFT LDA ONEBYT
 ASL A
 STA ONEBYT
 BCS SHOW
*
 INC HPSN
 BNE ITSOK
 INC HPSN+1
ITSOK JMP NOSHOW
*
* DISPLAY BIT
*
* SAVE X AND Y REGISTERS
*
SHOW TXA
 PHA
 TYA
 PHA
*
 JSR PLOT
*
* NOW DO IT AGAIN
*
 INC HPSN
 BNE NOINC
 INC HPSN+1
*
NOINC JSR PLOT
*
* RETRIEVE X AND Y REGISTERS
*
 PLA
 TAY
 PLA
 TAX
*
NOSHOW INC HPSN
 BNE LEAP
 INC HPSN+1
*
LEAP INY
 CPY #8
 BCC RSHIFT
*
 INC VPSN
*
 LDA HPTR
 STA HPSN
 LDA HPTR+1
 STA HPSN+1
*
* 2 VERT LINES DONE YET?
*
 DEC COUNT
 BNE DRAWLN
*
 INC BYTPTR
 BNE OKMSB
 INC BYTPTR+1
OKMSB DEX
 BNE SETLIN
 RTS
*
* MAIN ROUTINE STARTS HERE
*
```

```
351 START LDA VMCSB
352  ORA #8
353  STA VMCSB
354 *
355  LDA SCROLY
356  ORA #32
357  STA SCROLY
358 *
359 * USE BANK 2
360 *
361  LDA C2DDRA
362  ORA #3
363  STA C2DDRA
364 *
365  LDA CI2PRA
366  AND #252
367  ORA #1 ;BANK 2
368  STA CI2PRA
369 *
370 * CLEAR BIT MAP
371 *
372  LDA #0
373  STA FILVAL
374  LDA #<BASE
375  STA TABPTR
376  LDA #>BASE
377  STA TABPTR+1
378  LDA #<SCRLEN
379  STA TABSIZ
380  LDA #>SCRLEN
381  STA TABSIZ+1
382  JSR BLKFIL
383 *
384 * SET LINE, BKG AND BORDER COLORS
385 *
386  LDA #COLOR
387  STA FILVAL
388  LDA #<COLMAP
389  STA TABPTR
390  LDA #>COLMAP
391  STA TABPTR+1
392  LDA #<MAPLEN
393  STA TABSIZ
394  LDA #>MAPLEN
395  STA TABSIZ+1
396  JSR BLKFIL
397  LDA #13 ;GREEN
398  STA BORDER
```

```
*
* MOVE CHARACTER SET INTO RAM
*
* THIS ROUTINE TURNS BASIC OFF
*
 LDA R6510
 AND #$FE
 STA R6510
*
* TURN OFF KB INTERRUPT TIMER
*
 LDA CIACRE
 AND #$FE
 STA CIACRE
*
* SWITCH I/O OUT, CHAR ROM IN
*
 LDA R6510
 AND #$FB
 STA R6510
*
* COPY CHARACTERS INTO RAM
*
 LDA #<CHRBAS
 STA MVSRCE
 LDA #>CHRBAS
 STA MVSRCE+1
*
 LDA #<NEWADR
 STA MVDEST
 LDA #>NEWADR
 STA MVDEST+1
*
 LDA #<TABLEN
 STA LENPTR
 LDA #>TABLEN
 STA LENPTR+1
*
* START MOVE
*
 LDY #0
 LDX LENPTR+1
 BEQ MVPART
MVPAGE LDA (MVSRCE),Y
 STA (MVDEST),Y
 INY
 BNE MVPAGE
 INC MVSRCE+1
 INC MVDEST+1
 DEX
 BNE MVPAGE
MVPART LDX LENPTR
 BEQ MVEXIT
MVLAST LDA (MVSRCE),Y
 STA (MVDEST),Y
 INY
 DEX
 BNE MVLAST
MVEXIT
*
* SWITCH I/O BACK IN
*
 LDA R6510
 ORA #4
 STA R6510
*
* TURN TIMER BACK ON
*
 LDA CIACRE
 ORA #1
 STA CIACRE
*
* POSITION MESSAGE ON SCREEN
*
 LDA #8 *
 STA HPSN
 STA HPTR
 LDA #0 *
 STA HPSN+1
 STA HPTR+1
 LDA #VMID
 STA VPSN
 STA VPTR
*
* PRINT LINE OF LARGE TYPE
*
 LDX #0
DISP LDA TEXT,X
 CMP #0 ;EOF
 BEQ DONE
 STA LTTR
 TXA
 PHA
 JSR DRAWCH
 PLA
 TAX
```

```
*
* ADVANCE CURSOR
*
 CLC
 LDA HPTR
 ADC #16
 STA HPTR
 STA HPSN
 LDA HPTR+1
 ADC #0
 STA HPTR+1
 STA HPSN+1
 LDA VPTR
 STA VPSN
*
* PRINT NEXT LETTER
*
 INX
 JMP DISP
*
DONE
*
* DISPLAY SPRITE #0
*
* DEFINE SPRITE
*
* CLEAR SPRITE MAP
*
 LDA #0
 STA FILVAL
 LDA #<SP0ADR
 STA TABPTR
 LDA #>SP0ADR
 STA TABPTR+1
 LDA #64
 STA TABSIZ
 LDA #0
 STA TABSIZ+1
 JSR BLKFIL
*
* (COPY HEART FROM C64 CHR SET)
*
 LDA SP0ADR
 STA TEMPA
 LDA #83 ;HEART
 STA CHCODE
 JSR GETADR
*
```

```
543  LDY #0
544  LDX #8
545 *
546 DEFSP0 LDA (BYTPTR),Y
547  STA (TEMPA),Y
548 *
549  INC BYTPTR
550  INC TEMPA
551  INC TEMPA
552  INC TEMPA
553 *
554  DEX
555  BNE DEFSP0
556 *
557 * STORE SPRITE'S ADDRESS IN PTR
558 * (ADDRESS IS $8000 -- NO OFFSET)
559 *
560  LDA #0
561  STA SPRPTR
562 *
563 * EXPAND SPRITE (VERT & HORZ)
564 *
565  LDA #1
566  STA XXPAND
567  STA YXPAND
568 *
569 * TURN ON SPRITE #0
570 *
571  LDA #1
572  STA SPENA
573 *
574 * MAKE SPRITE RED
575 *
576  LDA #10 ;RED
577  STA SP0COL
578 *
579 * POSITION SPRITE ON SCREEN
580 *
581  LDA #62
582  STA SP0X
583  LDA #0
584  STA MSIGX
585  LDA #34
586  STA SP0Y
587 *
588 * MOVE SPRITE DOWN SCREEN
589 *
590 DROP INC SP0Y
```

```
*
* DELAY LOOP
*
 LDX #$FF
XLOOP LDY #$10
YLOOP DEY
 BNE YLOOP
 DEX
 BNE XLOOP
*
 LDA SPOY
 CMP #142
 BNE DROP
*
INF JMP INF
*
 END
```

15 Commodore 64/128 Music and Sound

An Introduction to Interrupt Operations

One of the best features of your Commodore computer is its incredible ability to synthesize music and sounds. Despite their low prices, the Commodore 64 and the Commodore 128 have sound and music generating capabilities that rival those of music synthesizers used by professional musicians. In this chapter, you will learn how to turn your Commodore's keyboard into a small music-synthesizer keyboard that can produce an almost limitless variety of sounds.

You can use either BASIC or Assembly language to program sound on your Commodore. But Assembly language is much better than BASIC for writing music and sound routines. Here are some of the reasons why:

- Sound is programmed on the Commodore 64 by manipulating specific bits in specific memory registers—a job that is slow and clumsy in BASIC, but fast and easy in Assembly language.
- Timing is often critical in sound and music programming, so the speed of Assembly language is especially important in programs that deal with music and sound.
- The length of a note cannot be determined very precisely in BASIC, but musical timing can be controlled with pinpoint precision in Assembly language. In fact, by using a programming tool called an interrupt, you can make the lengths of musical notes, rests, and phrases completely independent of everything else in an Assembly language program. By using interrupts, you can add music and sound to an Assembly language program with perfect synchronization—and you can be certain that your soundtrack will always run at the same speed, no matter how many other features are then added to the program.

To understand how the Commodore 64/128 music synthesizer works, it is necessary to know a few fundamental principles of sound reproduction. I will outline some of those principles now. Then we will be ready to put your Commodore through its paces as a music synthesizer.

THE FOUR CHARACTERISTICS OF SOUND

When you hear a sound being played on a musical instrument, there are really four characteristics which are combined to create the sound that you perceive. These four characteristics are:

1. *Volume,* or loudness.
2. *Frequency,* or pitch.
3. *Timbre,* or sound quality.
4. *Dynamic range,* or the difference in level between the loudest sound that can be heard and the softest sound that can be heard during a given period of time. This period of time can range between the time it takes to play a single note and the length of a much longer listening experience, such as a musical performance or a complete musical recording.

In the Commodore 64/128, there is a special microprocessor that can be programmed to control the volume, frequency, timbre, and dynamic range of sounds. It is this processor—called the 6581 SID (Sound Interface Device)—which gives the Commodore 64/128 its incredible sound-synthesizing capabilities.

SID'S THREE VOICES

The SID chip that is built into your Commodore has three separate voices, and each of these voices can be independently programmed. This means that your Commodore can play music in three-part harmony—or, if you prefer, you can use one voice for melody, one for percussion, and one for bass. If you like, you can use the SID chip to generate noises instead of music—and, if you wish, you can program each of SID's three voices to produce a different sound. You can even program SID to synthesize sounds that are recognizable as speech—if you are a good enough programmer.

In a moment, we will take a look at how the SID chip can be used to program three voices: that is, how it can control the *volume, frequency,* and *timbre* of three independent sources of sound. But first let's see where the SID chip is situated in your computer's memory, and how it is designed to be programmed.

WHERE TO FIND IT

In the Commodore 64, Memory Registers $D400 through $D7FF (54272 to 55295 in decimal notation) are used to address the SID chip. These 1,024 memory registers can be divided as shown in Table 15-1.

Table 15-1 MEMORY BLOCKS USED BY THE 6581 SID CHIP

$D400 through $D406	Registers for Voice 1
$D407 through $D40D	Registers for Voice 2
$D40E through $D414	Registers for Voice 3
$D415 through $D418	Sound filter and volume controls
$D419 through $D41A	Game paddle registers (not used for sound)
$D41B through $D41C	Read-only sound registers (Used in advanced synthesis operations)
$D41D through $D41F	Not connected
$D420 through $D7FF	Images of other registers; not used

ANOTHER USEFUL TABLE

Table 15-2 is a memory map of the SID chip's sound-related registers. The functions of most of the registers listed in this table will be explained later in this chapter. The functions of those registers not covered in this chapter can be found in the *Commodore 64 Programmer's Reference Guide,* and in most books devoted specifically to Commodore 64 music and sound.

As you can see from these tables, Registers $D400 through $D418 are the only SID registers that are ordinarily used in basic-level to intermediate-level SID programming. And the largest block of memory in the table—the section that extends from $D400 through $D414—can be broken down further into three subsections: one for Voice 1, one for Voice 2, and one for Voice 3. Later in this chapter, the functions of all of the registers in the block that extends from $D400 to $D414 will be covered in more detail. Meanwhile, let's take an overall look at how the SID chip's registers are used to program the volume, frequency, timbre, and dynamic range of the three voices of the Commodore 64.

VOLUME

For some reason, the designers of the Commodore 64 made it impossible to control the volume of the SID chip's three voices individually. Instead, the loudness of the overall sound produced by the SID register is determined by the value that is placed in the lower three bits (Bits 0 through 3) of Memory Register $D418. This register is sometimes known as the SIGVOL register.

To control the volume of all sounds produced by the SID chip, all you have to do is place a number ranging from $0 to $F in the lower nibble of the SIGVOL register. The larger the value of this nibble, the louder the sound which the SID chip produces. If the value of the nibble is $0, no sound will be generated. In most applications, the volume nibble of the SIGVOL register is kept at $F, its maximum setting.

Bits 4 through 6 of the SIGVOL register are used to control three sound filters that are built into the SID chip: a low-pass filter, a bandpass filter, and a high-pass filter. The uses of these filters will be explained later in this chapter.

Bit 7 of the SIGVOL register can be used to disconnect the output of Voice 3 of the SID chip. Voice 3 is disconnected by setting this bit to 1. When

Table 15-2 MEMORY MAP OF THE 6581 SID CHIP REGISTERS

ADDRESS	LABEL	FUNCTION
$D400	FRELO1	Voice 1 Frequency Control (low byte)
$D401	FREHI1	Voice 1 Frequency Control (high byte)
$D402	PWLO1	Voice 1 Pulse Waveform Width (low byte)
$D403	PWHI1	Voice 1 Pulse Waveform Width (high nibble)
$D404	VCREG1	Voice 1 Control Register
$D405	ATDCY1	Voice 1 Attack/Decay Register
$D406	SUREL1	Voice 1 Sustain/Release Control Register
$D407	FRELO2	Voice 2 Frequency Control (low byte)
$D408	FREHI2	Voice 2 Frequency Control (high byte)
$D409	PWLO2	Voice 2 Pulse Waveform Width (low byte)
$D40A	PWHI2	Voice 2 Pulse Waveform Width (high nibble)
$D40B	VCREG2	Voice 2 Control Register
$D40C	ATDCY2	Voice 2 Attack/Decay Register
$D40D	SUREL2	Voice 2 Sustain/Release Control Register
$D40E	FRELO3	Voice 3 Frequency Control (low byte)
$D40F	FREHI3	Voice 3 Frequency Control (high byte)
$D410	PWLO3	Voice 3 Pulse Waveform Width (low byte)
$D411	PWHI3	Voice 3 Pulse Waveform Width (high nibble)
$D412	VCREG3	Voice 3 Control Register
$D413	ATDCY3	Voice 3 Attack/Decay Register
$D414	SUREL3	Voice 3 Sustain/Release Control Register
$D415	CUTLO	Filter Cutoff Frequency (low nibble)
$D416	CUTHI	Filter Cutoff Frequency (high byte)
$D417	RESON	Filter Resonance Control Register
$D418	SIGVOL	Volume and Filter Select Register

voice 3 is disconnected, an oscillator with which Voice 3 is equipped can be used for modulating the sound of the other two voices. Or the Voice 3 oscillator can be used for other purposes, such as generating random numbers, without affecting the output of sound.

When the filters controlled by Register $D418 are not being used, and when there is no need to disconnect Voice 3, then the SID chip's volume can be controlled by simply storing a value ranging from $0 to $F (or from 1 to 15 in decimal notation) in the SIGVOL register. But when Bits 5 through 7 of the SIGVOL register are in use, then masking operations must be used in order to implement a desired volume setting without affecting the register's other functions. Here is a routine that could be used to implement a volume setting of 15 ($F in hexadecimal notation) without disturbing the high-order nibble of the SIGVOL register:

```
LDA SIGVOL
AND $#F0
ORA #$0F
STA SIGVOL
```

FREQUENCY

The pitch of a musical note is determined by its frequency. Frequency is usually measured in Hertz (Hz), or cycles per second. The frequencies that can be produced by the Commodore 64's SID chip range from 0 Hz (very low) to 4,000 Hz (quite high).

The SID chip synthesizes the frequencies of sounds by carrying out a rather complex mathematical operation. First, it reads a pair of 8-bit values (one "low" value and one "high" value) that have been placed in a specific pair of frequency control registers. (The SID chip has six such registers—two for each voice—and the addresses of all of them are listed in Table 15-2.)

When a pair of frequency-control registers has been loaded with two 8-bit values, it combines them into a 16-bit value. It then divides that 16-bit value by a number that is derived from a certain frequency: specifically, the frequency of a system clock built into the Commodore 64. Finally, when all of these operations have been carried out, the SID chip is able to generate a note of the desired frequency.

That is quite an involved series of operations, but you do not really have to worry about how they all work in order to produce a note of a given frequency on the Commodore 64. All you have to do is place the proper values in the proper memory registers, and then set a certain bit in another register. All of the values you need to play eight octaves of notes on the Commodore 64 are listed in a table on pages 384 through 386 of the *Commodore 64 Programmer's Reference Guide*. In that table, you find two values (a "low" value and a "high" value) that must be placed in the SID chip's frequency control registers in order to produce each note that the Commodore 64 is capable of generating. But please remember that the values listed in this table are not actual frequencies; they are numbers that the SID chip uses to calculate frequencies that are to be generated.

TIMBRE

Timbre, or note quality, can be illustrated with the help of a structure called a waveform. The SID chip can generate four kinds of waves: a triangle wave, a sawtooth wave, a pulse wave, and a noise wave.

To understand the concept of waveforms, it is necessary to have a fundamental understanding of musical harmonics. So here is a brief crash course in music theory.

With the help of an electronic instrument, it is possible to generate a tone that has just one pure frequency. But when a note is played on a musical instrument, more than one frequency is usually produced. In addition to a primary frequency, or a fundamental, there is usually a set of secondary frequencies called harmonics. It is this total harmonic structure which determines the timbre of a sound.

When a tone containing only a fundamental frequency is viewed on an oscilloscope, the pattern that is produced on the screen is that of a pure sine wave. (When a flute is played, the waveform that it produces is very close to that of a pure sine wave.) The waveform of a sine wave is shown in Figure 15-1.

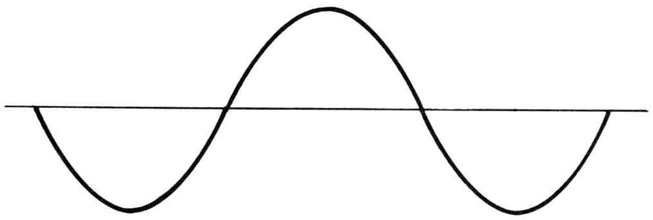

Figure 15-1 Sine Waveform

When harmonics are added to a tone, the result is a richer sound that produces what is sometimes called a triangle wave (see Figure 15-2). Triangle waveforms, or waves that are close to triangle waveforms, are produced by instruments including xylophones, organs, and accordians.

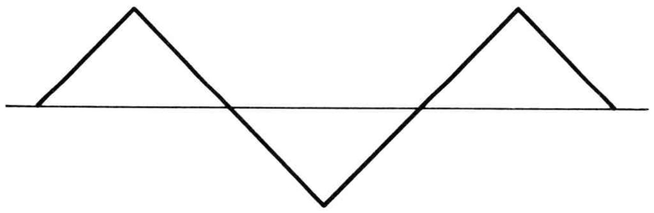

Figure 15-2 Triangle Waveform

When still more harmonics are added to a note, other kinds of waves are formed. Harpsichords and trumpets, for example, produce a type of wave that is sometimes called a sawtooth wave (see Figure 15-3). And a piano generates a squarish kind of wave called a square wave or a pulse wave (see Figure 15-4).

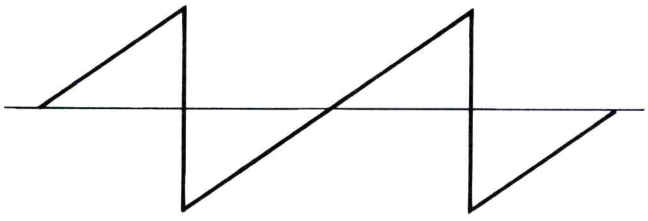

Figure 15-3 Sawtooth Waveform

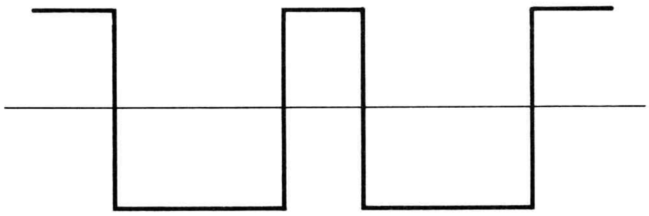

Figure 15-4 Pulse Waveform

PULSE WAVEFORM WIDTH CONTROLS

When the SID chip is called on to generate a pulse wave, it is necessary to use an additional control called a pulse waveform width control. As you can see in Figure 15-4, the pulses in a pulse waveform have a certain width, and are separated by gaps that may have a different width. The SID chip has six registers—two for each voice—that can be used to control the widths of pulse waveforms. A pulse wave generated by the SID chip has a 12-bit resolution, so only 12 bits in each pair of width-control registers are used: all eight bytes of each low-order register, plus the lower nibble of each high-order register.

The setting of each width-control register determines how long a pulse wave will stay at the high part of its cycle. The possible range of 12-bit values, ranging from 0 to 4,095, makes it possible for a square wave to stay in the high part of its cycle from 0% to 100% of the time, in 4,096 steps.

NOISE WAVEFORMS

Another kind of waveform that the SID chip can produce is a noise waveform. A noise waveform creates a random sound output that varies with a frequency proportionate to that of an oscillator built into Voice 1. Noise waveforms are often used to imitate the sounds of drums and even explosions and other nonmusical noises.

HOW TO SELECT A WAVEFORM

The SID chip has three registers—one for each voice—that can be used to determine the waveforms of sounds. These three registers, called control registers, are $D404 (for Voice 1), $D40B (for Voice 2), and $D412 (for Voice 3).

These three registers are multipurpose registers; only their high-order nibbles (Bits 4 through 7) are used for determining waveforms. The uses of the remaining bits will be discussed later in this chapter. Meanwhile, these are the bits that must be set to choose waveforms:

Bit 4: Triangle waveform
Bit 5: Sawtooth waveform
Bit 6: Pulse waveform
Bit 7: Random noise waveform

OTHER KINDS OF WAVES

Many other kinds of waves can be produced with the help of special filters. Three such filters—a low-pass filter, a high-pass filter and a band-pass filter—

are built into the Commodore 64. A low-pass filter masks out frequencies above a certain cut-off frequency, and attenuates the low frequencies that pass through. A high-pass filter masks out frequencies below a certain cut-off frequency, and attenuates the high frequencies that pass through. A band-pass filter cuts off frequencies that are outside a band near the center of the frequency spectrum, and attenuates the midrange frequencies that pass through.

As explained earlier in this chapter, under the section dealing with volume, SID Register $D418—the register that is used to control volume—is also used to control the SID chip's three sound filters.

For the sake of simplicity, the filters that are built into the SID chip will not be used in the program presented in this chapter. But you are encouraged to experiment with the filters when you run the program, since you may want to use them in programs which you design.

DYNAMIC RANGE

The dynamic range of a note—the difference in volume between its loudest sound level and its softest sound level—can be illustrated in many ways. To illustrate and control the dynamics of notes produced by the SID chip, engineers who designed your Commodore use a device called an ADSR envelope, or attack/decay/sustain/release envelope. An ADSR envelope illustrates four distinct stages in the life of a note: four phases which every note undergoes between the time it starts and the time it fades away. These four phases—called attack, decay, sustain, and release—are shown in the ADSR envelope illustrated in Figure 15-5.

The addresses of the SID registers that are used to create ADSR envelopes are listed in Table 15-2. As you can see by looking at this table, the SID chip has six registers—two for each voice—that are used to control the attack, decay, sustain, and release characteristics of notes. Each voice has one register that controls the attack and decay phases of notes, and another register that controls the sustain and release phases of notes.

Following are brief descriptions of the four note cycles illustrated in Figure 15-5.

PHASES 1 AND 2: ATTACK AND DECAY

Every note starts with an attack. The attack phase of a note is the length of time that it takes for the volume of the note to rise from a level of zero to the note's peak volume.

As soon as a note reaches its peak volume, it begins to decay. The decay phase of a note is the length of time that it takes for the note to decay from its peak volume to a predefined sustain volume.

As mentioned earlier, each of the SID chip's three voices has one register that controls both the attack and decay characteristics of notes which the

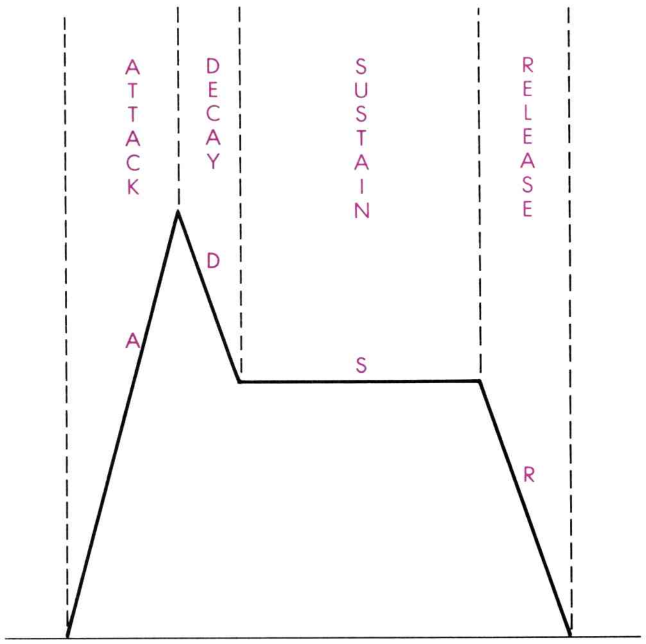

Figure 15-5 An ADSR Envelope

chip produces. The three SID registers that control attacks and decays are $D405 (for Voice 1), $D40C (for Voice 2), and $D413 (for Voice 3).

The high nibble of each of these registers (Bits 4 through 7) is used to set the duration of a note's attack cycle, and the low nibble of each register (Bits 0 through 3) is used to set the duration of a note's decay cycle. Each nibble can be set to a value ranging from $0 (for a duration of 2 milliseconds) to $F (for a duration of 8 seconds).

PHASES 3 AND 4: SUSTAIN AND DECAY

When the decay phase of a note ends, the note is usually sustained for a certain period of time at a certain volume. Then a release phase begins. During this final phase, the volume of the note drops from its sustain level back down to zero.

Each of the SID chip's three voices has one register that controls both the sustain and release characteristics of notes which the chip produces. The three SID registers that control the sustain and release phases of notes are $D406 (for Voice 1), $D40D (for Voice 2), and $D414 (for Voice 3).

The low nibble of each of these registers (Bits 0 through 3) is used to set the duration of a note's release cycle. Each of these "release" nibbles can be set to a value ranging from 0 (for 6 milliseconds) to 15 (for 24 seconds).

The high nibble (Bits 4 through 7) of each sustain/release register is used to control the sustain cycle of notes. But this nibble is not used to control the duration of the sustain cycle. Instead, it is used to control the volume that is to be maintained throughout the sustain cycle. The duration of a note's sustain cycle must be controlled with either a timing loop or some other kind of timer.

The value of the sustain nibble of a sustain/release register can range from 0 (for no volume) to 15 (equal to the note's peak volume).

THE SID CHIP'S CONTROL REGISTERS

Once you have determined a note's volume, frequency, waveform, and ADSR envelope, it is easy to instruct the SID chip to play the note. All you have to do is set one bit in one register: specifically, the gate bit in the control register for the SID voice that you are using.

The SID chip has three control registers: one for each voice. Their addresses are $D404 (for Voice 1), $D40B (for Voice 2), and $D412 (for Voice 3). These three registers were mentioned earlier in this chapter, since their high-order nibbles are used to select the waveforms that the SID chip generates. Now we are ready to talk about their low-order nibbles (Bits 0 through 3). The uses of these bits will now be described in reverse order, beginning with Bit 3.

Bit 3, the test bit of each SID control register, is used to disable the oscillator that is built into the voice that the register controls. When this oscillator is disabled, complex waveforms—even waveforms that synthesize speech—can be generated under software control.

Bit 2 of each SID control register is called a ring modulation bit. When this bit is set to 1, the triangle waveform of the voice controlled by the register is replaced with a ring-modulated combination of two oscillators, and can thus be used to simulate the sound of a bell or a gong.

Bit 1, a synchronization bit, can be used to synchronize the fundamental frequency of Oscillator 1 with the fundamental frequency of Oscillator 3, enabling the advanced programmer to create a wide range of complex harmonic structures using Voice 1.

Bit 0 is the main bit, or gate bit, of each SID control register. When you have selected a note's volume, frequency, waveform, and ADSR envelope, and have given the SID chip all the information it needs to play it, you can start the note by setting the gate bit of the proper SID control register. To stop the note—whether or not it has finished playing—all you have to do is clear the gate bit of the appropriate SID control register. Once you have cleared the gate bit, you can change the settings of any SID registers you wish. Then you can play another note—or create another sound—by setting the gate bit again. Or, if you prefer, you can play the same note or create the same sound over and over again, by repeatedly setting and clearing the gate bit while all other SID registers remain the same.

USING INTERRUPT ROUTINES

Shortly, you will have an opportunity to type, assemble and execute an Assembly language program that illustrates some of the music-synthesizing capabilities of the SID chip built into your Commodore. First, though, it might be helpful to discuss the concept of the interrupt, a very powerful programming technique that is often used in music and sound routines (as well as in many other kinds of high-performance programs).

An interrupt, often cryptically referred to as an IRQ, is a high-priority routine that interrupts other routines so that it can do its work. No matter what is happening when an interrupt is called, a computer will stop everything else it is doing in order to process the interrupt. An interrupt, in a manner of speaking, always goes to the head of the line and keeps other routines waiting while it does its job.

Assembly language programmers often use interrupts when they want to write time-critical routines. For example, one very important interrupt routine is built into the operating system of the Commodore 64. This routine, called a hardware interrupt routine, takes place 60 times a second, with quartz clockwork precision. During this interrupt, many vital and time-critical operations take place. For example, the computer's software clock is updated, the keyboard is read, and a cursor-blinking operation is performed. Every 1/60th of a second, when it is time for a hardware interrupt, the interrupt takes place and all other processing is temporarily halted. And not until the interrupt is completed does normal processing resume.

The hardware interrupt routine is very important to the Commodore Assembly language programmer because it can be customized with the help of a vector called the hardware interrupt vector. This vector is situated at Memory Addresses $0314 and $0315. It is often labeled the CINV vector in Commodore 64 programs.

Since the CINV vector is in a documented position in RAM, you can "steal" it at any time you like. That means you can make it point to any user-written routine instead of to the hardware interrupt vector that is built into your computer's operating system. Then, 60 times every second, with precise regularity, your own routine will be processed as an interrupt routine.

If you steal the CINV vector in this fashion, however, there are two potential problems that you will have to solve. Here is one of them. If you want your computer's operating system to continue to work normally, even though its CINV vector has been stolen, you will have to make sure that all of the operations that are normally carried out by the hardware interrupt vector still take place.

Fortunately, this is not a difficult task to manage. If you want to make sure that your own interrupt and the normal CINV interrupt both take place every one sixtieth of a second, all you have to do is take two simple steps:

1. Change the CINV vector to point to your own interrupt.

2. Then end your own interrupt with a jump to the address the CINV vector originally pointed to.

As mentioned above, there is one problem that must be faced each time the CINV vector is changed. Fortunately, this problem is also quite easy to solve.

Here is the problem. The CINV vector consists of two 8-bit memory registers which, in combination, always hold a 16-bit address. When the CINV vector is to be altered, therefore, it must be changed in two steps. First, the low byte of the address which the vector points to must be changed. Then the high byte must be changed.

Normally, this sort of operation is no problem to the Assembly language program. But the CINV vector is a very special sort of vector; its job is to direct your computer to an operation that is carried out 60 times every second. There is always a chance, therefore, that the CINV vector will become active after one of its bytes has been changed but before the other byte has been changed. If that happens, the CINV vector may point to an incorrect address when it is called, resulting in a program crash or a system failure.

THE SEI AND CLI INSTRUCTIONS

To prevent this kind of catastrophe from taking place, the 6502/6510 chip has two special instructions for dealing with interrupts. One of these instructions is SEI, which stands for "set interrupt disable." The other is CLI, which means "clear interrupt disable." When an SEI instruction is invoked during the processing of an Assembly language program, the interrupt disable flag of the processor status register is set, and no maskable interrupts (which is what the CINV interrupt is) can take place. When a CLI instruction is used during an Assembly language program, it has just the opposite effect; the interrupt disable flag of the P register is cleared, and maskable interrupts are enabled.

Since the instructions SEI and CLI can enable and disable interrupts so easily, they can be used to change the C64's CINV vector with complete safety. To make sure that a program does not crash during the altering of the CINV vector, all you have to do is use the instruction SEI before the vector is changed, and then use the instruction CLI after it is changed. When you take that simple precaution, you can be sure that no interrupts take place while the vector is being altered, and that the vector is cleanly and safely changed.

In the program which is the topic of this chapter—a program which I have named MUSIC.S—the CINV vector is altered to include a note-timing loop. That ensures that the musical notes produced by the program will always be precisely timed.

This is the section of the program in which the address pointed to by the CINV vector is changed. First, the OS address ordinarily pointed to by the CINV vector is stored in a pair of memory registers called USERADD and USERADD + 1.

Next, in Line 60, the instruction SEI is used to disable maskable interrupts. When that has been done, the address of a user-written routine (a note-timing loop) is stored in the address of the CINV vector. Then interrupts are re-enabled with a CLI instruction.

```
53 *
54 *       SET UP INTERRUPT
55 *
56          LDA  CINV
57          STA  USERADD
58          LDA  CINV+1
59          STA  USERADD+1
```

```
60          SEI
61          LDA   #<WAIT
62          STA   CINV
63          LDA   #>WAIT
64          STA   CINV+1
65          CLI
66 *
```

Now here is the note-timing routine that is added to the CINV vector by the above sequence of instructions:

```
111 *
112 WAIT        LDX     TIMER
113             DEX
114             BNE     RETURN
115             LDA     #64
116             STA     WF
117             LDX     #0
118 RETURN      STX     TIMER
119             JMP     (USERADD)
```

We will discuss how this timing routine works later on in this chapter. For now, it is sufficient to remember that the routine ends with the statement JMP (USERADD). That statement links the user-written timing loop in the MUSIC.S program with a jump to the address originally pointed to by the CINV vector.

ONE FINAL PROGRAM

Now we are ready to examine MUSIC.S, the last program in this book. Here it is. Type it, assemble it into object code (as MUSIC.O), and run it, and it will turn your Commodore into an electronic piano. When the MUSIC.O program has been loaded into your computer and executed, you will be able to use the keys on the A-row as white piano keys, and the keys on the Q-row as black keys, as shown in Figure 15-6.

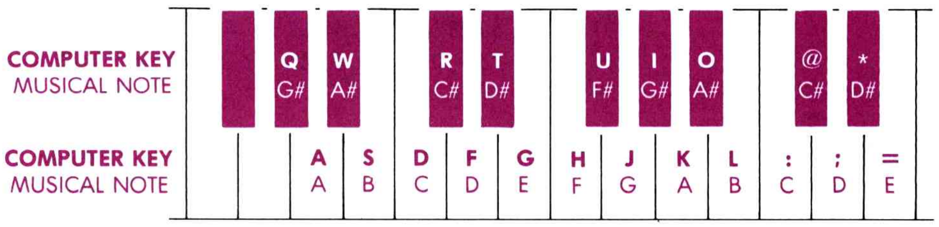

Figure 15-6 Keyboard Arrangement for the MUSIC.S Program

Now here is a listing of the MUSIC.S program:

```
MUSIC.S: A PIANO KEYBOARD PROGRAM
FOR THE COMMODORE 64

*
* MUSIC.S
*
               ORG     $8000
*
SFDX           EQU     $CB
*
CINV           EQU     $314
USERADD        EQU     $311
*
GETIN          EQU     $FFE4
*
SIGVOL         EQU     $D418
ATDCY1         EQU     $D405
PWHI1          EQU     $D403
PWLO1          EQU     $D402
SUREL1         EQU     $D406
FREHI1         EQU     $D401
FRELO1         EQU     $D400
VCREG1         EQU     $D404
*
TIMER          EQU     $FB
CHAR           EQU     TIMER+1
*
               JMP     INIT
*
KEYS           DFB     62,10,9,13,18,17,21,22
               DFB     26,29,30,34,33,37,38,42
               DFB     45,46,50,49,53
*
HIFREQ         DFB     13,14,14,15,16,17,18,19
               DFB     21,22,23,25,26,28,29,31
               DFB     33,35,37,39,42
*
LOFREQ         DFB     78,24,239,210,195,195,209,239
               DFB     31,96,181,30,156,49,223,165
               DFB     135,134,162,233,62
*
*
* CLEAR SOUND REGISTERS
*
*
INIT           LDA     #0
```

```
              LDX      #$18
CLOOP         STA      #$D400,X
              DEX
              BNE      CLOOP
*
* SET UP TIMER
*
              LDA      #60
              STA      TIMER
*
* SET UP INTERRUPT
*
              LDA      CINV
              STA      USERADD
              LDA      CINV+1
              STA      USERADD+1
              SEI
              LDA      #<WAIT
              STA      CINV
              LDA      #>WAIT
              STA      CINV+1
              CLI
*
* SET REGISTERS
*
              LDA      #15
              STA      SIGVOL
              LDA      #9
              STA      ATDCY1
              LDA      #0
              STA      SUREL1
              STA      PWHI1
              LDA      #255
              STA      PWLO1
              LDA      #64
              STA      VCREG1
*
GETKEY        LDA      SFDX
              CMP      #64
              BNE      SKIP
              LDA      #0
              STA      CHAR
              JMP      GETKEY
*
SKIP          LDX      #20
CHECK         CMP      KEYS,X
              BEQ      PLAY
              DEX
```

```
 92              BPL    CHECK
 93              JMP    GETKEY
 94 *
 95 PLAY         CMP    CHAR
 96              BNE    CONT
 97              JMP    GETKEY
 98 *
 99 CONT         STA    CHAR
100              LDA    #60
101              STA    TIMER
102              LDA    #64
103              STA    VCREG1
104              LDA    HIFREQ,X
105              STA    FREHI1
106              LDA    LOFREQ,X
107              STA    FRELO1
108              LDA    #65
109              STA    VCREG1
110              JMP    GETKEY
111 *
112 WAIT         LDX    TIMER
113              DEX
114              BNE    RETURN
115              LDA    #64
116              STA    VCREG1
117              LDX    #0
118 RETURN       STX    TIMER
119              JMP    (USERADD)
```

HOW THE MUSIC.S PROGRAM WORKS

The MUSIC.S program is fairly easy to follow. In Lines 43 through 47, it clears SID registers $D400 through $D418 with the help of a simple X loop. Next, a note timer is set and a note-timing routine is added to the Commodore hardware interrupt (CINV) vector. Then, in Lines 69 through 79, the major SID registers that control Voice 1 are loaded with values that will emulate the sound of a piano (these settings, plus settings that reproduce the sounds of several other instruments, are listed on page 164 of the *Commodore 64 User's Guide*).

The heart of the MUSIC.S program is a loop labeled GETKEY that occurs in Lines 81 through 86. This loop scans the Commodore keyboard repeatedly to see whether a key is pressed. If no key is pressed, the loop repeats until a keypress is detected. Once a key has been pressed, the program jumps to Line 88, where a routine labeled SKIP begins.

We will move on to the routine labeled SKIP in a moment. First, though, let's pause to take a closer look at how the GETKEY loop works.

In Line 81, the line in which the GETKEY loop starts, the accumulator is loaded with the value of a memory register labeled SFDX. This register, situated at Memory Address $CB, is used by an operating system routine called a key-scan interrupt routine. Sixty times every second, during the Commodore 64's hardware interrupt vector, this memory address is loaded with a special code number which the computer uses to determine whether a key is being pressed—and, if so, what key is being held down.

The code numbers that are used by the Commodore keyscan interrupt routine are neither ASCII code numbers nor Commodore screen code numbers. They are special key-code numbers that are listed in Table 15-3.

Table 15-3 KEY-CODE NUMBERS

KEY CODE	KEY	KEY CODE	KEY
0	Insert/Delete Key	33	I
1	Return Key	34	J
2	Cursor Right Key	35	0 (Zero)
3	Function Key f7	36	M
4	Function Key f1	37	K
5	Function Key f3	38	O (Letter)
6	Function Key f5	39	N
7	Cursor Down Key	40	+
8	3	41	P
9	W	42	L
10	A	43	-
11	4	44	.
12	Z	45	:
13	S	46	@
14	E	47	,
15	Not Used	48	Lira (British Pound Sign)
16	5	49	*
17	R	50	;
18	D	51	Clear/Home
19	6	52	Not Used
20	C	53	=
21	F	54	Up Arrow (Exponential Sign)
22	T	55	/
23	X	56	1
24	7	57	Left Arrow
25	Y	58	Not Used
26	G	59	2
27	8	60	Space Bar
28	B	61	Not Used
29	H	62	Q
30	U	63	Run/Stop
31	V	64	No Key Pressed
32	9		

WHY KEY CODES ARE USED

The MUSIC.S program makes use of key codes because key-code values, not ASCII-code values, are the kinds of values that are returned each time the Commodore 64 scans its keyboard to check for pressed keys. Each time a key is pressed, the Commodore operating system converts the key-code value of the depressed key into an ASCII-code value. Then that ASCII-code value is placed in a special "type-ahead" buffer so that it can be kept in memory long enough to be displayed on a screen.

That is a good system for printing text on a screen, but it is not an ideal system for a musical-keyboard program such as MUSIC.S. A type-ahead buffer is neither necessary nor desirable in a musical-keyboard program. And the Commodore 64's "debounce" feature, which causes a letter to be printed repeatedly on the screen after it has been held down for a short period of time, creates more problems than it solves when it is used in musical-keyboard programs.

Examine Lines 27 through 29 of the MUSIC.S program, and you will see a data table—labeled KEYS—of the key codes for all 21 keys that are used in the program. Immediately following this table, there are similar data tables showing the high-frequency and low-frequency code numbers of each note in the KEYS table. In each of these tables, the offset used for each note is identical. With the help of indirect addressing, therefore, the three tables can be used together to locate any valid note in the MUSIC.S program and to determine its proper frequency settings.

Now let's return to the GETKEY loop in the MUSIC.S program, the loop that begins at Line 81. Notice that the loop keeps recycling as long as the SFDX register has a value of 64. Refer to the key-code table that was presented previously, and you will see that a key-code value of 64 means that no key is being pressed. So, as long the SFDX register holds a value of 64, a value of 0 will be loaded over and over again into a special memory register that has been labeled CHAR, and the GETKEY loop in the MUSIC.S program will keep on repeating. (We will see in a few moments how the 0 that is stored in the CHAR register during this operation is used.)

As soon as a key is pressed, the value of SFDX will change from 64 to some other value, and the program will jump to the routine called SKIP that starts at Line 88. This routine uses an X loop to count down through the 21 key-code values that are valid in the MUSIC.S program. If a valid key is pressed, the program will jump to a PLAY routine which starts at Line 95. Otherwise, the program will keep looping back until a valid note is typed in.

When the PLAY routine begins, the first thing the program does is check the status of a variable labeled CHAR. This variable is used to determine whether a key has just been pressed or whether it is merely being held down. If a key has just been pressed, the CHAR register will hold a different value from the value of the key being pressed. If this is the case, the program will jump to a routine labeled CONT (for "continue") and a new note will be played. But if a key that has already initiated a note is still being pressed, the CHAR register will hold the same value as the value of the key being pressed, and a new note will not begin.

Because of a tricky feature of the GETKEY routine, the process I have just described will always work, even when the same key is pressed over and over again. In the GETKEY routine, in Lines 84 and 85, the value of CHAR is reset to zero every time a finger is lifted from a key. Because of this feature, a key that is pressed and held down will cause a note to sound only once. But if the key is released and then pressed down again, the note will play again.

The CONT routine, which extends from Line 99 through Line 110, is the routine during which notes are actually played. During this routine, the value 65 is stored in the Voice 1 Control Register, clearing that register's gate bit and turning off any note that may be playing. Then an interrupt-controlled note timer (labeled TIMER) is set to a value of 60, which corresponds to a playing time of one second. When this has been done, the high and low frequency codes that correspond to whatever note has been selected are stored in the appropriate SID registers. Then the value 65 is stored in the Voice 1 Control Register, setting that register's gate bit and starting a new note. Then an unconditional jump is made back to the GETKEY routine, so that a new note can be played. Meanwhile, the routine labeled WAIT—now a part of your computer's hardware interrupt vector—keeps ticking away, making sure that the ADSDR envelope of every note you play is correctly timed.

Once you have typed, assembled, and executed the MUSIC.S program, you may decide you would like to make it more complicated. As written, the program uses only one of the SID chip's voices—but it could easily be expanded into a program that uses all three. Then, to accompany your melody line, you could add harmony, a bass line, or even the sound of drums.

Since you have also learned how to write graphics programs in Assembly language, you could also improve the MUSIC.S program by adding some color graphics, creating something interesting to look at while the music plays. If you want to tackle a really challenging programming job, you may be able to expand the MUSIC.S program into one that can store and play back melodies that you have typed in on the keyboard. Then, by mixing Assembly language and BASIC, you may even be able to figure out how to store selections that you have played and recorded on a disk, so that you can reload them at any time you like, and can incorporate them into other programs.

Appendix A The 6510 Instruction Set

This appendix is a complete listing of the 6510 microprocessor instruction set—all of the instruction mnemonics used in Commodore 64 Assembly language programming. It does not include pseudo-operations (also known as pseudo-ops, or directives), which vary from assembler to assembler.

Here are the meanings of the abbreviations used in this appendix:

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

6510 MEMORY REGISTERS

A–Accumulator
X–X register
Y–Y register
M–Memory register

ADDRESSING MODES

A–Absolute addressing
AC–Accumulator addressing
Z–Zero-page addressing
IMM–Immediate addressing
IND–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
ZX–Zero-page X-indexed (Zero-page,X) addressing
ZY–Zero-page Y-indexed (Zero-page,Y) addressing

THE 6510 INSTRUCTION SET (6510 MNEMONICS)

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 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

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

ASL (Arithmetic shift left): Moves each bit in the accumulator or a specified memory location one position to the left. A zero is deposited into the 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 is clear, results in no operation if the carry flag is set. Destination of branch must be within a range of –126 to +129 memory addresses of the first instruction following the BCC instruction.

Flags affected: None

Registers affected: None

Addressing modes: R

BCS (Branch if carry set): Executes a branch if the carry flag is set, results in no operation if the carry flag is clear. Destination of branch must be within a range of –126 to +129 memory addresses of the first instruction following the BCS instruction.

Flags affected: None

Registers affected: None

Addressing modes: R

BEQ (Branch if equal): Executes a branch if the zero flag is set, results in no operation if the zero flag is clear. Can be used to jump to cause a branch if the result of a calculation is zero, or if two numbers are equal. Destination of branch must be within a range of –126 to +129 memory addresses of the first instruction following the BEQ instruction.

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 location. The contents of the accumulator are not affected, but three flags in the P register are.

If any bits in the accumulator and the value being tested match, the Z flag is cleared. If no match is found, the Z flag is set. Therefore, a BIT instruction followed by a BNE instruction can be used to detect a match, and a BIT instruction followed by a BEQ instruction can be used to detect a no-match condition.

In addition, 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 of the BIT instruction is often used in signed binary arithmetic. If a BIT operation results in the setting of the N flag, then the value being tested is negative. If the operation results in the setting of the V flag, that indicates a carry in signed-number math.

Flags affected: N, V, Z

Registers affected: None

Addressing modes: A, Z

BMI (Branch on minus): Executes a branch if the N flag is set, results in no operation if the N flag is clear. Destination of branch must be within a range of –126 to +129 memory addresses of the first instruction following the BMI instruction.

Flags affected: None

Registers affected: None

Addressing modes: R

BNE (Branch if not equal): Executes a branch if the zero flag is clear (that is, if the result of an operation is non-zero). Results in no operation if the zero flag is set. Can be used to jump to cause a branch if the result of a calculation is not zero, or if two numbers are not equal. Destination of branch must be within a range of –126 to +129 memory addresses of the first instruction following the BNE instruction.

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. Destination of branch must be within a range of –126 to +129 memory addresses of the first instruction following from the BPL instruction.

Flags affected: None

Registers affected: None

Addressing modes: R

BRK (Break): Halts the execution of a program, much like an interrupt would, and also stores the value of the program counter, plus two, on the hardware stack, along with the contents of the P register (which now has the B flag set). BRK is often used in debugging, and affects various debuggers in various ways. For more details, see your assembler and debugger's instruction manual.

Flags affected: B

Registers affected: None

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. This instruction is used primarily in operations involving signed numbers. Destination of branch must be within a range of –126 to +129 memory addresses of the first instruction following from the BVC instruction.

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. This instruction is used primarily in operations involving signed numbers. Destination of branch must be within a range of –126 to +129 memory addresses of the first instruction following from the BVS instruction.

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 mode): Puts the computer into binary mode (its default) so that binary operations (the kind most often used) can be carried out properly.

Flags affected: D

Registers affected: None

Addressing modes: IMP

CLI (Clear interrupt mask): Enables interrupts. Used in advanced Assembly language programming. For more details, see advanced Commodore Assembly languages texts and manuals.

Flags affected: I

Registers affected: None

Addressing modes: IMP

CLV (Clear overflow flag): Clears the P register's overflow flag by setting it to zero. This instruction is used primarily in operations involving signed numbers.

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. The result of the operation thus depends upon what branch instruction is used, and whether the value in the accumulator is less than, equal to, or more than the value being tested.

Flags affected: N, Z, C

Registers affected: None

Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX

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 result of the operation thus depends upon what branch instruction is used, and whether 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 result of the operation thus depends upon what branch instruction is used, and whether 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

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: 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

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: 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: X

Addressing modes: IMP

JMP (Jump to address): Causes program execution to jump to the address specified. The JMP instruction can be used with absolute addressing, and it is the only 6510 instruction that can be used with unindexed indirect addressing. A JMP instruction that uses indirect addressing is written in the format:

```
JMP ($0600)
```

If this statement were used in a program, the JMP instruction would cause program execution to jump to the address stored in Memory Addresses $0600 and $0601.

Flags affected: None

Registers affected: None

Addressing modes: A, 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: None

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 zero.

Flags affected: N, Z

Registers affected: A

Addressing modes: A, Z, IMM, AX, AY, IX, IY, ZX

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 zero.

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 the Z flag is set if the value loaded into the Y register is zero.

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 zero 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 one or more 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

PHA (Push accumulator): The contents of the accumulator are pushed on the stack. The accumulator and the P register are not affected.

Flags affected: None

Registers affected: None

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: None

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

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: None

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 status of both the program counter and the P register are restored in preparation for resuming the routine that was in progress when an interrupt occurred. All flags of the P register are restored to their original values. Interrupts are used in advanced Assembly language programming, and detailed information on interrupts is available in advanced Assembly language texts and Commodore reference manuals.

Flags affected: N, V, B, D, I, Z, C

Registers affected: None

Addressing modes: IMP

RTS (Return from subroutine): At the end of a subroutine, returns execution of a program to the next address after the JSR (jump to subroutine) instruction that caused the program to jump to the subroutine. At the end of an Assembly language program, the RTS instruction returns control of a Commodore computer to the device that was in control before the program began—usually the screen editor.

Flags affected: None

Registers affected: None

Address 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

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 6510 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.

Flags affected: D

Registers affected: None

Addressing modes: IMP

SEI (Set interrupt disable): Disables the interrupt response to an IRQ (maskable interrupt). Does not disable the response to an NMI (nonmaskable interrupt). Interrupts are used in advanced Assembly language programming, and are described in advanced Assembly language texts and Commodore reference manuals.

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

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 theY register are not affected.

Flags affected: None

Registers affected: M

Addressing modes: A, Z, 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: X

Addressing modes: IMP

TSX (Transfer stack to X register): The value in the stack pointer is deposited in the X register. The N and C flags are conditioned in accordance with the result of this operation. The value of the stack pointer is not changed.

Flags affected: N, C

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: None

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

Appendix B
Additional Programs

ADD8BIT was written using the Merlin 64 assembler. See pages 96-97 for an explanation of this program.

```
ADD8BIT.S
*
*ADD8BIT
*
         ORG   $8000
*
ADD8BIT  LDA   #0
         STA   $02AA
*
         CLD
         CLC
*
         LDA   $02A7
         ADC   $02A8
         BCS   ERROR
         STA   $02A9
         RTS
ERROR    LDA   #1
         STA   $02AA
         RTS
```

The following program, DATMOV, was written using the Merlin 64. See page 98 for a description.

```
DATMOV.S
*
* DATMOV
*
TXTBUF    EQU    $02A7
EOL       EQU    $0D
*
          ORG    $8000
*
          JMP    DATMOV
*
TEXT      HEX    54,41,4B,45,20,4D,45,20
          HEX    54,4F,20,59,4F,55,52,20
          HEX    4C,45,41,44,45,52,21,0D
*
DATMOV    LDX    #0
LOOP      LDA    TEXT,X
          STA    TXTBUF,X
          CMP    #EOL
          BEQ    FINI
          INX
          JMP    LOOP
FINI      RTS
          END
```

Following is a version of THE QUEST program, listed on page 110, that is written for the Merlin 64 assembler:

```
QUEST.S
*
* THE QUEST
*
          ORG    $8000
*
BUFLEN    EQU    23
CHROUT    EQU    $FFD2
*
          JMP    BEGIN
*
TEXT      DFB    87,72,69,82,69,32,73,83
          DFB    32,84,72,69,32,67,79,77
          DFB    77,79,68,79,82,69,63
*
BEGIN     LDX    #0
*
LOOP      LDA    TEXT,X
          JSR    CHROUT
          INX
          CPX    #BUFLEN
          BNE    LOOP
          RTS
```

The RESPONSE program is described on pages 119-122. This is the Merlin 64 version:

```
RESPONSE.S
*
* RESPONSE
*
          ORG    $8000
EOL       EQU    13
BUFLEN    EQU    24
FILLCH    EQU    $20
CHROUT    EQU    $FFD2
*
          JMP    START
*
TEXT      ASC    'YOU CAN FIND HIM IN 64K'
          DFB    13
*
* CLEAR TEXT BUFFER
*
START     LDA    #FILLCH
          LDX    #BUFLEN
STUFF     DEX
          STA    TXTBUF,X
          BNE    STUFF
*
* STORE MESSAGE IN BUFFER
*
          LDX    #0
LOOP1     LDA    TEXT,X
          STA    TXTBUF,X
          CMP    #EOL
          BEQ    PRINT
          INX
          CPX    #BUFLEN
          BCC    LOOP1
*
* PRINT MESSAGE
*
PRINT     LDX    #0
LOOP2     LDA    TXTBUF,X
          PHA
          JSR    CHROUT
          PLA
```

```
            CMP    #EOL
            BNE    NEXT
            JMP    FINI
NEXT        INX
            CPX    #BUFLEN
            BCC    LOOP2
*
FINI        RTS
*
TXTBUF      DS     BUFLEN
*
            END
```

Following is a version of the ODDTEST program, described on page 132, for the Merlin 64 assembler:

```
ODDTEST.S
*
* ODDTEST
*
VALUE1      EQU    $FB
VALUE2      EQU    $FC
FLGADR      EQU    $FD
*
            LDA    #10
            STA    VALUE1
            LDA    #0
            STA    FLGADR
*
            LDA    VALUE1
            LSR    A
            STA    VALUE2
*
            BCS    FLAG
            RTS
*
FLAG        ROL    FLGADR
            RTS
```

This is the Merlin 64 version of the PACKDATA program, which is listed on pages 137-138:

```
PACKDATA
*
* PACKDATA
*
            ORG    $8000
*
```

```
NYB1      EQU   $FB
NYB2      EQU   $FC
PKDBYT    EQU   $FD
*
          LDA   #$04
          STA   NYB1
          LDA   #$06
          STA   NYB2
*
          CLC
          LDA   NYB1
          ASL   A
          ASL   A
          ASL   A
          ASL   A
          ORA   NYB2
          STA   PKDBYT
          RTS
```

Following is the Merlin 64 version of the UNPACKIT program, listed on page 139:

```
UNPACKIT.S
*
* UNPACKIT
*
PKDBYT    EQU   $FB
LOWBYT    EQU   $FC
HIBYT     EQU   $FD
*
          ORG   $8000
*
          LDA   #255
          STA   PKDBYT
          LDA   #0
          STA   LOWBYT
          STA   HIBYT
*
          LDA   PKDBYT
          AND   #$0F
          STA   LOWBYT
          LDA   PKDBYT
          LSR   A
          LSR   A
          LSR   A
          LSR   A
          STA   HIBYT
          RTS
```

This is the Merlin 64 version of the ADDNCARRY program listed on page 144:

```
ADDNCARRY
* ADDNCARRY
          ORG    $8000
          CLD
          CLC
          LDA    #$8D
          ADC    #$FF
          STA    $FB
          RTS
```

Following is the ADD16 program, written for the Merlin 64 assembler. See page 146 for a description of this program.

```
ADD16.S
*
* ADD16
*
          ORG    $8000
*
          CLD
          CLC
          LDA    $FB
          ADC    $FD
          STA    $02A7
          LDA    $FC
          ADC    $FE
          STA    $02A8
          RTS
```

The SUB16 program, written on the Merlin 64, follows. See page 147 for a description of this program.

```
SUB16.S
*
* SUB16
*
          ORG    $8000
*
          CLD
          SEC
          LDA    $FD
          SBC    $FB
          STA    $02A7
          LDA    $FE
          SBC    $FC
          STA    $02A8
          RTS
```

The following program, MULT16, was written using the Merlin 64 assembler. Pages 148-149 contain a fully-commented version for the Commodore 64 assembler.

```
MULT16.S
*
* MULT16
*
MPR       EQU   $FD
MPD1      EQU   $FE
MPD2      EQU   $02A7
PRODL     EQU   $02A8
PRODH     EQU   $02A9
*
          ORG   $8000
*
* THE NUMBERS WE'LL MULTIPLY
*
          LDA   $250
          STA   MPR
          LDA   #2
          STA   MPD1
*
MULT      CLD
          CLC
          LDA   #0
          STA   MPD2
          STA   PRODH
          STA   PRODL
          LDX   #8
LOOP      LSR   MPR
          BCC   NOADD
          LDA   PRODH
          CLC
          ADC   MPD1
          STA   PRODH
          LDA   PRODL
          ADC   MPD2
          STA   PRODL
NOADD     ASL   MPD1
          ROL   MPD2
          DEX
          BNE   LOOP
          RTS
          END
```

The next program, MULT16B, is described on pages 150-151. This version is for the Merlin 64.

```
MULT16B.S
*
* MULT16B
*
PRODL    EQU    $FD
PRODH    EQU    $FE
MPR      EQU    $02A7
MPD      EQU    $02A8
*
         ORG    $8000
*
VALUES   LDA    #10
         STA    MPR
         LDA    #10
         STA    MPD
*
         LDA    #0
         STA    PRODH
         LDX    #8
LOOP     LSR    MPR
         BCC    NOADD
         CLC
         ADC    MPD
NOADD    ROR    A
         ROR    PRODH
         DEX
         BNE    LOOP
         STA    PRODL
         RTS
```

The DIV8/16 program is described on pages 151-153. This version was written using the Merlin 64.

```
DIV8/16.S
*
* DIV8/16
*
         ORG    $8000
*
DVDH     EQU    $FD
DVDL     EQU    $FE
QUOT     EQU    $02A7
DIVS     EQU    $02A8
RMDR     EQU    $02A9
*
         LDA    #$1C
         STA    DVDL
         LDA    #$02
```

```
             STA    DVDH
             LDA    #$05
             STA    DIVS
*
             LDA    DVDH
             LDX    #08
             SEC
             SBC    DIVS
DLOOP        PHP
             ROL    QUOT
             ASL    DVDL
             ROL    A
             PLP
             BCC    ADDIT
             SBC    DIVS
             JMP    NEXT
ADDIT        ADC    DIVS
NEXT         DEX
             BNE    DLOOP
             BCS    FINI
             ADC    DIVS
             CLC
FINI         ROL    QUOT
             STA    RMDR
             RTS
```

The following program, BIGCHRS, is based on the SHOWCHRS program listed on pages 228-237. Instead of typing the entire BIGCHRS program, load SHOWCHRS and edit it to look like the following:

```
BIGCHRS.S
*
* BIGCHRS
*
             ORG    $8000
*
COLOR        EQU    $10
COLMAP       EQU    $8400
BASE         EQU    $A000
VICTRL       EQU    $D011
CI2PRA       EQU    $DD00
CIADIR       EQU    $DD02
VICMEM       EQU    $D018
*
HMAX         EQU    320
HMID         EQU    160-4
VMID         EQU    100-4
*
```

```
SCRLEN   EQU   8000
MAPLEN   EQU   1000
*
TEMPA    EQU   $FB
TEMPB    EQU   TEMPA+2
*
TABPTR   EQU   TEMPA
TABSIZ   EQU   $02A7
*
HPSN     EQU   TABSIZ+2
VPSN     EQU   HPSN+2
CHAR     EQU   VPSN+1
ROW      EQU   CHAR+1
LINE     EQU   ROW+1
BYTE     EQU   LINE+1
BITT     EQU   BYTE+2
*
MPRL     EQU   BITT+1
MPRH     EQU   MPRL+1
MPDL     EQU   MPRH+1
MPDH     EQU   MPDL+1
PRODL    EQU   MPDH+1
PRODH    EQU   PRODL+1
*
FILVAL   EQU   PRODH+1
*
R6510    EQU   $0001
NEWADR   EQU   $8800
CHRBAS   EQU   $D000
CIACRE   EQU   $DC0E
*
TABLEN   EQU   $800
*
MVSRCE   EQU   $61
MVDEST   EQU   MVSRCE+2
BYTPTR   EQU   MVDEST+2
*
LENPTR   EQU   $9000
CHCODE   EQU   LENPTR+2
HPTR     EQU   CHCODE+2
ONEBYT   EQU   HPTR+1
COUNT    EQU   ONEBYT+2
*
         JMP   START
*
* CALCULATE CHCODE'S ADDRESS
*
GETADR   LDA   #0
```

```
         STA    CHCODE+1
         LDA    CHCODE
         CLC
         ASL    A
         ROL    CHCODE+1
         ASL    A
         ROL    CHCODE+1
         ASL    A
         ROL    CHCODE+1
         STA    CHCODE
*
         CLC
         LDA    CHCODE
         ADC    #<NEWADR
         STA    BYTPTR
         LDA    CHCODE+1
         ADC    #>NEWADR
         STA    BYTPTR+1
         RTS
*
* BLOCK FILL ROUTINE
*
BLKFIL   LDA    FILVAL
         LDX    TABSIZ+1
         BEQ    PARTPG
         LDY    #0
FULLPG   STA    (TABPTR),Y
         INY
         BNE    FULLPG
         INC    TABPTR+1
         DEX
         BNE    FULLPG
PARTPG   LDX    TABSIZ
         BEQ    FINI
         LDY    #0
PARTLP   STA    (TABPTR),Y
         INY
         DEX
         BNE    PARTLP
FINI     RTS
*
* 16-BIT MULTIPLICATION ROUTINE
*
MULT16   LDA    #0
         STA    PRODL
         STA    PRODH
         LDX    #16
SHIFT    ASL    PRODL
```

```
         ROL   PRODH
         ASL   MPRL
         ROL   MPRH
         BCC   NOADD
         CLC
         LDA   MPDL
         ADC   PRODL
         STA   PRODL
         LDA   MPDH
         ADC   PRODH
         STA   PRODH
NOADD    DEX
         BNE   SHIFT
         RTS
*
* PLOT ROUTINE
*
* ROW=VPSN/8 (8-BIT DIVIDE)
*
PLOT     LDA   VPSN
         LSR   A
         LSR   A
         LSR   A
         STA   ROW
*
* CHAR=HPSN/8 (16-BIT DIVIDE)
*
         LDA   HPSN
         STA   TEMPA
         LDA   HPSN+1
         STA   TEMPA+1
         LDX   #3
DLOOP    LSR   TEMPA+1
         ROR   TEMPA
         DEX
         BNE   DLOOP
         LDA   TEMPA
         STA   CHAR
*
* LINE=VPSN AND 7
*
         LDA   VPSN
         AND   #7
         STA   LINE
*
* BITT=7-(HPSN AND 7)
*
         LDA   HPSN
```

```
          AND   #7
          STA   TEMPA
          SEC
          LDA   #7
          SBC   TEMPA
          STA   BITT
*
* BYTE=BASE+ROW*HMAX+8*CHAR+LINE
*
* FIRST MULTIPLY ROW * HMAX
*
          LDA   ROW
          STA   MPRL
          LDA   #0
          STA   MPRH
          LDA   #<HMAX
          STA   MPDL
          LDA   #>HMAX
          STA   MPDH
          JSR   MULT16
*
* ADD PRODUCT TO BASE
*
          CLC
          LDA   #<BASE
          ADC   PRODL
          STA   TEMPA
          LDA   #>BASE
          ADC   PRODH
          STA   TEMPA+1
*
* MULTIPLY 8 * CHAR
*
          CLC
          LDA   #0
          STA   TEMPB+1
          LDX   #3
          LDA   CHAR
MULT8     ASL   A
          ROL   TEMPB+1
          DEX
          BNE   MULT8
*
* ADD LINE
*
          CLC
          ADC   LINE
          STA   TEMPB
```

```
         LDA   TEMPB+1
         ADC   LINE+1
         STA   TEMPB+1
*
* TEMPA + TEMPB = BYTE
*
         CLC
         LDA   TEMPA
         ADC   TEMPB
         STA   TEMPB
         LDA   TEMPA+1
         ADC   TEMPB+1
         STA   TEMPB+1
*
* POKE BYTE,PEEK(BYTE)OR2↑BIT
*
         LDX   BITT
         INX
         LDA   #0
         SEC
SQUARE   ROL
         DEX
         BNE   SQUARE
         LDY   #0
         ORA   (TEMPB),Y
         STA   (TEMPB),Y
         RTS
*
* MAIN ROUTINE STARTS HERE
*
START    LDA   VICMEM
         ORA   #8
         STA   VICMEM
*
         LDA   VICTRL
         ORA   #32
         STA   VICTRL
*
* USE BANK 2
*
         LDA   CIADIR
         ORA   #3
         STA   CIADIR
*
         LDA   CI2PRA
         AND   #252
         ORA   #1          ;BANK 2
         STA   CI2PRA
```

```
*
* CLEAR BIT MAP
*
         LDA   #0
         STA   FILVAL
         LDA   #<BASE
         STA   TABPTR
         LDA   #>BASE
         STA   TABPTR+1
         LDA   #<SCRLEN
         STA   TABSIZ
         LDA   #>SCRLEN
         STA   TABSIZ+1
         JSR   BLKFIL
*
* SET BKG AND LINE COLORS
*
         LDA   #COLOR
         STA   FILVAL
         LDA   #<COLMAP
         STA   TABPTR
         LDA   #>COLMAP
         STA   TABPTR+1
         LDA   #<MAPLEN
         STA   TABSIZ
         LDA   #>MAPLEN
         STA   TABSIZ+1
         JSR   BLKFIL
*
* TURN OFF KB INTERRUPT TIMER
*
MVCHRS   LDA   CIACRE
         AND   #$FE
         STA   CIACRE
*
* SWITCH BASIC OUT
*
         LDA   R6510
         AND   #$FE
         STA   R6510
*
* SWITCH I/O OFF, CHAR ROM ON
*
         LDA   R6510
         AND   #$FB
         STA   R6510
*
* COPY CHARACTERS INTO RAM
```

```
*
         LDA   #<CHRBAS
         STA   MVSRCE
         LDA   #>CHRBAS
         STA   MVSRCE+1
*
         LDA   #<NEWADR
         STA   MVDEST
         LDA   #>NEWADR
         STA   MVDEST+1
*
         LDA   #<TABLEN
         STA   LENPTR
         LDA   #>TABLEN
         STA   LENPTR+1
*
* START MOVE
*
         LDY   #0
         LDX   LENPTR+1
         BEQ   MVPART
MVPAGE   LDA   (MVSRCE),Y
         STA   (MVDEST),Y
         INY
         BNE   MVPAGE
         INC   MVSRCE+1
         INC   MVDEST+1
         DEX
         BNE   MVPAGE
MVPART   LDX   LENPTR
         BEQ   MVEXIT
MVLAST   LDA   (MVSRCE),Y
         STA   (MVDEST),Y
         INY
         DEX
         BNE   MVLAST
MVEXIT
*
* SWITCH I/O BACK IN
*
         LDA   R6510
         ORA   #4
         STA   R6510
*
* TURN TIMER BACK ON
*
         LDA   CIACRE
         ORA   #1
         STA   CIACRE
*
* DRAW A CHARACTER
*
         LDA   #<HMID
         STA   HPSN
         STA   HPTR
         LDA   #>HMID
         STA   HPSN+1
         STA   HPTR+1
         LDA   #VMID
         STA   VPSN
*
         LDA   #1         ;'A'
         STA   CHCODE
         JSR   GETADR
*
* A NESTED LOOP:
*
* (X IS THE OUTSIDE LOOP)
*
         LDX   #8
*
* SET UP COUNTER FOR 2 VERT LINES
*
SETLIN   LDA   #2
         STA   COUNT
*
DRAWLN   LDY   #0
         LDA   (BYTPTR),Y
         STA   ONEBYT
*
* THE INSIDE LOOP:
*
* (Y IS ZERO AT START)
*
RSHIFT   LDA   ONEBYT
         ASL   A
         STA   ONEBYT
         BCS   SHOW
*
         INC   HPSN
         BNE   ITSOK
         INC   HPSN+1
ITSOK    JMP   NOSHOW
*
* DISPLAY BIT
*
```

```
* SAVE X AND Y REGISTERS
*
SHOW     TXA
         PHA
         TYA
         PHA
*
         JSR   PLOT
*
* NOW DO IT AGAIN
*
         INC   HPSN
         BNE   NOINC
         INC   HPSN+1
*
NOINC    JSR   PLOT
*
* RETRIEVE X AND Y REGISTERS
*
         PLA
         TAY
         PLA
         TAX
*
NOSHOW   INC   HPSN
         BNE   LEAP
         INC   HPSN+1
*
LEAP     INY
         CPY   #8
         BCC   RSHIFT
*
         INC   VPSN
*
         LDA   HPTR
         STA   HPSN
         LDA   HPTR+1
         STA   HPSN+1
*
* 2 VERT LINES DONE YET?
*
         DEC   COUNT
         BNE   DRAWLN
*
         INC   BYTPTR
         BNE   OKMSB
         INC   BYTPTR+1
OKMSB    DEX
```

```
450          BNE   SETLIN
451 *
452 INF      JMP   INF
453 *
454          END
455 *
```

Bibliography

If you would like to learn more about 6502, 6510 and 8502 Assembly-language programming—or more about your Commodore computer—the following books might be helpful:

Andrews, Mark. *Atari Roots: A Guide to Atari Assembly Language.* Chatsworth CA: Datamost, 1984.

Camp, Robert. *Creating Arcade Games on the Commodore 64.* Greensboro NC: COMPUTE! Books, 1984.

Commodore Business Machines, Inc. *Commodore 64 Programmer's Reference Guide.* Wayne PA: Commodore, 1982. Distributed by Howard W. Sams & Co., Inc., Indianapolis.

Commodore Business Machines, Inc. *Commodore 64 User's Guide.* Wayne PA: Commodore, 1982.

COMPUTE! Books. *COMPUTE!'s First Book of Commodore 64 Sound and Graphics.* Greensboro NC: COMPUTE! Publications, 1983.

Findley, Robert. *6502 Software Gourmet Guide & Cookbook.* Rochelle Park NJ: Hayden, 1979.

Hyde, Randy. *Using 6502 Assembly Language.* Chatsworth CA: Datamost, 1982.

Leemon. *Mapping the Commodore 64.* Greensboro NC: COMPUTE! Books, 1984.

Leventhal, L. *6502 Assembly Language Programming.* Berkeley CA: Osborne/McGraw-Hill, 1979.

______, and Seville. *6502 Assembly Language Subroutines.* Berkeley CA: Osborne/McGraw-Hill, 1982.

Maurer, W. Douglas. *Apple Assembly Language.* Rockville MD: Computer Science Press, 1984.

Onosko, Tim. *Commodore 64: Getting the Most from It.* Bowie MD: Robert J. Brady/Prentice-Hall, 1983.

Platt, Charles. *Graphics Guide to the Commodore 64.* Berkeley CA: Sybex, 1984.

Saunders, William B. *The Elementary Commodore 64.* Chatsworth CA: Datamost, 1983.

Wagner, Roger. *Assembly Lines: The Book.* North Hollywood CA: Softalk Publishing, 1982.

Zaks, Rodney. *Programming the 6502.* Fourth Edition. Berkeley CA: Sybex, 1983.

______. *Advanced 6502 Programming.* Berkeley CA: Sybex, 1982.

Index

#, 8, 37, 54, 93, 94
$, 8, 19, 47, 48, 70. *See also* Hexadecimal numbers; Merlin assembler/editor
#$, 37
$q, 71
$r, 71
%, 19, 47. *See also* Binary numbers; Merlin assembler/editor
*, 47-49, 61, 215
*=, 59
. (period), 73, 75
:, 47, 48, 55, 70, 71
;, 59, 61
=, 79
<>, 127
@, 60
], 66, 67

A

A (Append), 47-48
A (Accumulator) register, 9, 36, 87
Absolute addressing, 95, 140
Absolute indexed addressing, 97-98
Accumulator addressing, 94-95
Accumulator (A) register, 9-10, 36-37, 87
ADC instruction, 37, 54, 101, 282
  8 addressing modes, 92
Addition
  and carry flag, 38-40, 54, 143-146
  4-bit, 143
  8-bit, 144
  program for. *See* ADDNRS program; ADDNCARRY program
  program with conditional branching, 96-97
  16-bit program, 146-147
ADDNCARRY program, 145
ADDNRS program
  and addressing modes, 92-93
  using Commodore 64 assembler, 57-65, 71-75
  using Merlin 64 assembler, 47-56, 70-71
Address bus, 35
Address modification routines, 213-214. *See also* Relative address modification
Addressing modes, 91-106, 282
  13 modes for 6510/8502, 92
ADRAY1, ADRAY2 vectors, 83-89
ADSR envelope, 268-270
ALU (Arithmetic logic unit), 35-37

AND operator, 134-135, 139, 140, 282
Animation, with joysticks, 189-193. *See also* Sprites
Arithmetic, 143-158
Arithmetic logic unit, 35-37
Artifical intelligence, 126
ASCII code, 56, 67, 71, 78, 97-98, 111, 219
ASL instruction, 130-131, 139, 282
ASM command, 55, 67, 70
ASSEMBLER64, 56-65
Assemblers, viii, 5, 45. *See also* Commodore 64 Macro; Merlin 64; Panther C64
  choosing, ix-x
  functions of, 10-11
Assembly language, 3-17, 153. *See also* Mnemonics
  capabilities, 3-4
  fields. *See* Fields
  programming with, 11, 45-67
  relationship to machine language, 7-8
  running programs in, 69-89
  sharing memory with machine-language programs, 181-183
AUTO command, 58

B

BALLBOUNCE program, 186
Bank-switching, 171, 172, 220
BASIC, 4-5, 171
  graphics programming, 186-192
  high resolution graphics, 193-194
  line numbers in, 50-51, 58-59
  number conversion, 29-31
  running machine-language from, 16-17, 69, 80-89
  sharing memory with machine-language, 179-181
  USR(X). *See* USR(X) function
BCC instruction, 113, 283
BCD (binary-coded decimal) mode arithmetic, 40-41, 156-157
BCS instruction, 113, 283
BEQ instruction, 113, 283
BIGCHRS program, 237-238, Appendix B
Binary-coded decimal (BCD) mode arithmetic, 40-41, 156-157
Binary numbers, 5-6, 19-22. *See also* %; 4-Bit; 8-Bit numbers; Signed numbers
  compared to decimal and hexadecimal, 23-25
  converting to/from decimal, 24-28
  converting to/from hexadecimal, 24-25, 28-29
Bit-mapped graphics display, 195-198, 227
Bit masking, 135, 136
BIT operator, 134, 140-141, 283
Bit shifting operations, 129-134
  and division, 151
  and multiplication, 148-151
Bit testing, 137, 138-139
Bits, 20, 129-141. *See also* 4-Bit, 8-Bit Numbers
  map, and sprites, 242
BLACKBOARD program
  Assembly, 201-207
  BASIC, 193-194
BMI instruction, 113, 141, 284
BNE instruction, 113, 119, 284
Boolean logic, 134-135
BOOT ALL program, 57, 60
BPL instruction, 113, 141, 284
Branching. *See* Conditional branching
BRANCHIT program, 115-118
Break flag, 39, 41
BRK instruction, 74, 284
BVC instruction, 113, 140, 285
BVS instruction, 114, 140, 285
BYTE directive, 111
Bytes, 20, 23
  high, low order, 208
  map of, and sprites, 242, 243

C

C2DDRA register, 221
Call address, 78
Carriage return, testing for, 98, 121
Carry bit. *See* Carry flag
Carry flag (C), 38-40, 112, 129-130, 133, 143-146
Central processing unit (CPU), 9, 12, 35
Character generator ROM, 171, 172, 194-195
Characters, 78, 168
  copying from ROM to RAM, 221-224
  customizing, 219-238
  headline-size, 237-238
  modification, 224-227
  moving, and starting addresses, 177-178
CHAREN bit, 171, 172
CHRIN routine, 126-127
CHROUT routine, 78-79, 111-112, 126
CIACRE register, 221
CINV vector, 271-272
CLC instruction, 40, 53, 285
CLD instruction, 41, 53, 285
CLI instruction, 40, 272, 285
CLV instruction, 42, 285
CMP instruction, 96, 112, 134, 286
COBOL, 4, 5
Color
  codes, 188
  high resolution, 197-198
  for sprites, 244-245
Color memory, 167, 168, 173, 187-189
  map of, 188
Comments field, Assembly programs, 52, 59-60

Commodore 64 Macro Assembler Development System, viii, ix, x-xi. *See* ASSEMBLER64; EDITOR64; HILOADER64; HILOADER64; LOLOADER64
  memory map, of, 182
  monitors. *See* Monitors
  programming with, 56-65
  running Assembly programs with, 71-75
Commodore 64 computer, vii, 12-13
Commodore 64 Programmer's Reference Guide, viii, ix, 78, 81, 126, 164, 166, 169, 177, 187, 191, 241, 265, 276
Commodore 64 User's Guide, 177, 187
Commodore 128 computer, vii
Comparison instructions, 96, 112-113
Compilers, 4-5, 11
Complements, number, 154-156
Condition flags, 39
  Break flag, 39, 41
  Decimal mode flag (D), 40-41
  Interrupt disable flag (I), 40
Conditional branching, 95-97, 112-119. *See also* Jump instructions
  how instructions are used, 118-119
  length limitations, 117
  mnemonic instructions, 113
Conversion of numbers, 23, 25-31
Counters, 38
  Program Counter (PC), 38, 52
CPU (Central processing unit), 9, 12, 35
CPX instruction, 96, 112, 119, 134, 286
CPY instruction, 96, 112, 134, 286

D

(Delete), 48
D6510, 169, 170
Data bus, 35
Data direction register, 169, 170
Debugging programs
  END and, 54-55
  monitor program and, 57, 69
  using BRK, 74
DEC instruction, 286
Decimal mode flag (D), 40-41
Decimal numbers, 19. *See also* BCD mode arithmetic
  compared to binary and hexadecimal, 23-25
  conversion to/from binary, 24-28
  conversion to/from hexadecimal, 23, 29-31
DEX instruction, 38, 287
DEY instruction, 38, 287
DFB directive, 111
DFC directive, 111
Disk drives, ix, xi
Division, using bit-shifting, 131-132, 134, 151-153
DOS (disk operating system), 13, 164, 166. *See also* Kernal
  running Assembly programs from, 76, 79-80
  and the stack, 105
DOS wedge, x, xii, 57, 60
Dynamic range in sounds, 268

E

EDITOR64, 56, 60, 61
8-Bit
  addition program. *See* ADDNRS
  binary numbers, 21-22, 129, 144
  memory and 16-bit numbers, 32
  number, values of bits in, 26
8502 microprocessor, viii, 12-14, 35-42
END/.END, 51, 54-55, 59
EOR operator, 134, 136-137, 287
EQU, 78-79, 208
Errors, 65, 213. *See also* Debugging programs

F

Fields in Assembly programs, 50-52. *See also* Label field; Line-number field; Op-code field; Operand field; Remarks field
Filters, and waveforms, 267-268
Flags. *See* Condition flags, Status flags
FLGADR (flag address), 132
Floating-point math, 82-84, 157-158. *See also* ADRAY1, ADRAY2 vectors
4-bit binary numbers, 21, 143
Frequency, and sound, 265

G

Game controller. *See* Joystick
GETKEY loop, 276, 279
GOSUB, 9, 51
GOTO instruction, 51, 144
Graphics. *See also* Character sets; High resolution graphics
  four types of data, 172-173
  and memory, 171-179

H

h command, 70, 71, 144
Hardware interrupt routines, 271
Hardware stack. *See* Stack
Harmonics, 265-266
Hex offset, 72

Hexadecimal numbers, 6, 8, 19-20, 22-23. *See also* $
  compared to binary and decimal, 23-25
  converting to/from binary, 28-29
  and decimal conversion, 23-24
  HI.TEST hex program, 6
  literal 37. *See* #$
High-level languages, 4-5
High resolution graphics, 137, 175, 185-198
  BASIC BLACKBOARD program, 193-194
  color, 197-198
  mixing with text, 227-237
HILOADER64, 56-57, 61, 71
HIRAM bit, 171
HI.TEST program
  assembly, 8
  assembly and machine compared, 7
  BASIC, 16-17
  binary code, 6
  hex code, 6
  origin directive, 15

I

I (Insert) 48
Immediate addressing, 93
Implicit (implied) addressing, 93
INC instruction, 287
Index register, X register as, 38
Indexed addressing, 97-98
Indexed indirect addressing, 99-100
Indirect addressing, 99-101
Indirect indexed addressing, 100-101
Input/output (I/O), 220
  devices, 12, 35, 36
  memory addresses, 168, 169, 170. *See also* R6510
Internal registers, 9, 35-42. *See also* Accumulator register; Processor status register; Program counter; Stack pointer; X register; Y register
Interpreters, 4-5, 10, 164
Interrupt disable flag (I), 40
Interrupt routines, 40, 261, 270-273
INX instruction, 38, 112, 287
INY instruction, 38, 288
IRQ. *See also* Interrupt routines

J

JMP, 51, 84, 101, 114, 288
Joystick, 212, 215
  controlling animation with, 189-193
  values of, 192
JOYSTICK program, 190
JSR instruction, 9, 51, 79, 86-87, 114, 288
Jump instructions, 86, 114-115. *See also* Conditional branching
Jump table, 77-78

K

Kernal, 77-79
  memory addresses, 169
Key codes, 277-279
Keyboard, 12, 35, 221
  for music program, 273
Keyscan interrupt routine, 277-279

L

l command, 71
L (List), 48-49
  of ADDNRS program, 50, 62-63
Labels in Assembly programs
  in Commodore 64 assembler, 59
  in Merlin 64 assembler, 51, 53
Languages, computer. *See* Assembly language; BASIC; COBOL; High-level languages; LOGO; Machine language; Pascal; PILOT
LDA instruction, 9, 37, 54, 99, 288
LDX instruction, 99, 288
LDY instruction, 289
Line numbers, 47, 49, 51
  Commodore 64 assembler, 58-59
  Merlin 64 assembler, 47, 49, 51
LOAD, 69
Loader programs, 56-57, 71 *See also* HILOADER64, LOLADER64
Logical operators, 133, 134
  AND, 134-135, 139, 140, 282
  BIT, 134, 140-141, 283
  EOR, 134, 135-137, 287
  ORA, 134, 135-136, 289
LOGO, 5
LOLOADER64, 56-57, 60, 71, 72
Loops, 80-81, 111-112, 119-121, 228
LORAM bit, 171
Low resolution screen, 191
LSR (logical shift right) instruction, 131-133, 139, 289

M

m command, 76, 144
Machine language, x, 4, 5-7
  assembler, xi
  compared to Assembly language, 7-8
  loader, xi
  running from BASIC, 7, 16-17, 69, 76, 80-81
  running programs in, 15-17
  sharing memory with Assembly programs, 181-183

sharing memory with BASIC programs, 179-181
Macros, x, xi
Maskable/nonmaskable interrupts, 40
Memory, 12, 35, 53, 129, 163-183. *See also* Addressing modes; Color memory; Memory addresses; RAM; ROM; Screen memory; Stack
blocks used by SID microprocessor, 263
and graphics, 171-179
map of, 165, 174
Memory addresses, 8, 14. *See also* Addressing modes; Kernal; Memory registers
$0000 through $0fff, 164-166
$0000, 163-164, 169. *See also* D6510
$0001, 163-164, 169. *See also* R6510
$0003-$0006. *See* ADRAY1 and ADRAY2
$0100 through $01FF, 166
$0200 through $03FF, 166-167
$0314,$0315. *See* CINV vector
$0400 through $07FF, 167
$0800 through $9FFF, 167
$A000 through $BFFF, 168
$C000 through $CFFF, 168
$CB. *See* SFDX register
$D000 through $D800, 168
$D011. *See* SCROLY register
$D015. *See* SPENA register
$D018. *See* VMCSB register
$D418. *See* SIGVOL
$D800 through $DBFF, 168
$DBFF through $FFF, 169
$DC0E. *See* CIACRE register
$DD00, and graphics data, 176
$DD02. *See* C2DDRA register
Memory banks, 174-177
Memory page, 208
Memory registers, 9, 14, 35, 281. *See also* D6510; FRETOP; Memory addresses; MEMSIZ; R6510; USERADD
map of SID, 264
SID control, 270
sprite position, 244
storing 16-bit numbers in 8-bit, 31-32
Merlin 64 assembler/editor, viii, ix, xi-xiii.
Editor mode, 47-49. *See also*:
Executive mode, 46-47. *See also* %
memory map of, 181-182
menu, 46-47
monitor. *See* Monitors
programming with, 45-46
running Assembly programs with, 70-71
Microprocessor unit (MPU), 12, 35
Mnemonics, 7, 8-10, 282-293
operation code, 51-52, 59
for the stack, 106
Monitors
Commodore 64, 57, 71-75
debugging with, 57, 69
machine-language, 57
Merlin 64, 47, 70-71
Panther C64, 75-76
to run machine-language, 69-76
MOS Technology, Inc., 12
MOVECHRS program, 221-224
MPU. *See* Microprocessor unit
Multiplication, 208
and bit-shifting, 130-131, 134, 148-151
Music, 272-279
MUSIC program, 273-276
MYCHRS program, 224-227

N

NAME GAME program, 122-127
Negative flag, 42, 140
Nibble, 20
NOP instruction, 289
Numbers, 19-32. *See also* BCD mode arithmetic; Binary numbers; Decimal Numbers; Hexadecimal numbers; Signed numbers
even and odd, 132-133
literal, 8, 37, 93. *See also* #

O

O (save object code), 56
Object code, 7, 10-11, 55-56, 64, 67
Offset values, 116-117
ON. . . .GOTO, 213, 214
One's complement, 154
Op-code. *See* Operation code
Operands, 52, 59, 114
and addressing modes, 92, 93, 94
Operating system. *See* DOS
Operation code mnemonics, 51, 59, 91
Operators, logical, 133, 134
AND, 134-135, 139, 140, 282
BIT, 134, 140-141, 283
EOR, 134, 136-137, 287
ORA, 134, 135-136, 289
ORA operator, 134, 135-136, 289
ORG, 51, 53, 76
Origin
directive, 14-15, 51, 53, 67, 76
line, 52, 59
Overflow flag (V), 42, 140
and signed numbers, 156

P

P Register. *See* Processor status register
Packing/unpacking data, 131, 133, 137-140
Page Zero, 94, 98-99, 164-166. *See also* Zero page addressing

Panther C64 assembler, viii, ix, xii, 65-67. *See also* Monitors
memory map of, 182
Panther Computer Corporation, viii
Pascal, 4, 5
Penguin math, 20-22
PHA instruction, 106, 289
PHP instruction, 106, 290
PILOT (language), 5
Pixel, 187
PLA instruction, 106, 290
PLP instruction, 106, 290
POKE statements, 16-17, 32
Positive/negative numbers. *See* Signed numbers
Printing
ADDNRS program, 55-56, 62
characters on a screen, 78, 111
PRINTIT program, 76-80
PRTR, 55
Processor status (P) register, 38-42, 281. *See also* Break flag; Carry flag; Decimal mode flag; Interrupt disable flag; Negative flag; Overflow flag; Zero flag
bits correlated to flags, 39
and the stack, 106-107
Program Counter (PC), 38, 52
Programmer calculator, 25
Programs. *See also* Appendix B
ADDNCARRY, 145
ADDNRS, 47-67, 92-93
ALLBOUNCE, 186
BIGCHRS, 237-238
BLACKBOARD
Assembly, 201-207
BASIC, 193-194
BRANCHIT, 115-118
divison, 152-153
HI.TEST
assembly, 7-8
BASIC, 16-17
binary, 6
hexadecimal, 6
machine, 7, 16-17
origin directive, 15
JOYSTIX, 190
MOVECHRS, 221-224
multiplication, 148-151
MUSIC, 274-276
MYCHRS, 224-227
NAME GAME, 122-127
number conversion, 27-28, 30-31
PRINTIT, 76-80
QUEST, 110-111, 119-122
RESPONSE, 119-121
16-bit addition, 146
16-bit subtraction, 147
showing conditional branching, 96-97
SHOWSHRS, 228-237
SKETCHER, 209-212
SPRITE, 245-258
USRX.BAS, 84, 88-89
USRX.S, 84-87
Pseudo-operation codes, 51, 54, 59, 111
Pulse waveform, 266-267

Q

Q (Quit), 55
QUEST program, 110-111, 119-122

R

R (Replace), 49
R6510, 169, 170, 171
Bit 2 and character generation, 220
RAM (random-access memory), 12, 13-14. *See also* Kernal
color memory, 173
for user programs, 166, 167, 168, 169
moving characters from ROM to, 221-224
storing 16-bit numbers in 8-bit, 31-32
Registers. *See* Memory registers
Relative address modification, 213, 214-216
Relative addressing, 95-96, 97. *See also* Conditional branching
RESPONSE program, 119-121
Roger Wagner Publishing, Inc., viii
ROL instruction, 132, 133-134, 145, 290
ROM (read-only memory), 12, 13
character generator, 172
moving characters to RAM from, 221-224
ROM image characters, 173-175, 219-220
ROR instruction, 133-134, 290
RTI instruction, 291
RTS instruction, 9, 51, 54, 86, 114, 291

S

Saving ADDNRS program, 57-58, 62-63, 67, 74
Sawtooth waveform, 266
SBC instruction, 291
Screen coordinates, 191, 195
Screen memory mapping, 167, 172-173, 187, 194
moving, and starting addresses, 178-179
Scrolling, 197
SCROLY register, 197
SEC instruction, 40, 291
Secondary routine, 51
SED instruction, 41, 292
SEI 40, 272, 292
Set bits, 140
SFDX register, 276-278
SHOWCHRS program, 228-237
SID microprocessor, 262-270
control registers, 270
memory blocks used by, 263
Signed numbers, 153-156

SIGVOL register, 263-265
Sine waveform, 266
16-bit numbers, 22-23. *See also* Hexadecimal numbers
  addition program, 146-147
  division, 151-153
  multiplication, 150-151, 208
  in PEEK, POKE commands, 31
  retrieving, storing in RAM, 32
  subtraction, 147
6502 Assembly Language Subroutines, 153
6502 microprocesssor, viii, 12
6510 microprocessor, viii, 12-13, 35-42
  diagram, 36
  instruction set. *See* Mnemonics
6581 SID (Sound Interface Device). *See* SID microprocessor
SKETCHER program, 209-212
Sound, characteristics of, 262
  Frequency, 265
  Timbre, 265-266
  Volume, 263-265
  Waveforms, 266-268
Sound generation, 261-279
Source code, 7, 11, 55-56, 63, 64
Sourceror, xi
Spacing when typing in programs, 51, 61-62, 75
SPENA register, 243
SPRITE program, 245-258
Sprites, 173, 241-258
  position registers, 244
STA instruction, 37, 54, 145, 292
Stack, 9, 86, 101-106, 228
  incrementing, decrementing, 102-103
  memory addresses, 166
  program to increment, decrement, 105-106
Stack pointer, 38, 102-103
Status flags, 38-42
  Carry flag (C), 38-40, 112, 129-130, 133, 143-146
  Negative flag, 42, 140
  Overflow flag (V), 42, 140. *See also* Overflow flag
  Zero flag (Z), 40, 112, 140
Status register. *See also* Processor status register
STX instruction, 99, 292
STY instruction, 292
Subroutine, 51, 114
Subtraction, 112
  and carry flag, 38-40
  16-bit, 147
SYS command, 16, 80, 81

T

TAX instruction, 87, 292
TAY instruction, 293
Texas Instruments Incorporated 25
Text buffer, 97-98, 121-122
Text
  mixing with high-resolution graphics, 227-237
  screen, 175, 194-195
Timbre, 265-266
Time critical routines, 271, 272, 276
Triangle waveform, 266
Truth tables, 135, 136
TSX instruction, 293
Two's complements, 155-156
TXA instruction, 293
TXS instruction, 293
TNA instruction, 293

U

Unindexed indirect addressing, 101
USERADD registers, 81-82, 89
USR(X) function, 81-83, 158
  program illustrating, 85-89

V

V flag. *See* Overflow flag
Vector, 77-78. *See also* ADRAY1, ADRAY2 vectors; CINV vector
VIC-II (video interface chip), 171-179, 220
  memory control register. *See* VMCSB
Video
  memory RAM, 167
  monitor, 12, 35
  refresh cycle, 171, 220
VLSI (very large scale integrated circuit), 12, 35
VMCSB register, 177-179, 197, 219
Volume, 263-265

W

W (write text file), 56
Waveforms, 266-268
  filters and, 267-268
  pulse, 266-267
  sawtooth, 266
  sine, 266
  triangle, 266
Word (16 bits), 20, 23

X

X register, 38, 87-88, 97, 99-101
  incrementing, decrementing, 111-112, 119, 151

Y

Y register, 38, 87, 97, 100-101
  incrementing, decrementing, 112

Z

Zero flag (Z), 40, 112, 140
Zero page addressing, 94, 140. *See also* Page Zero
Zero page,X addressing, 98
Zero page,Y addressing, 99

SAMS

Commodore 64™/128™ Assembly Language Programming

Here it is! The first complete book about Assembly language for both the Commodore 64 and 128 computers. You'll use this extensive collection of Assembly programs again and again. You'll also learn how to:

- Design your own character set
- Write joystick-controlled action games
- Draw on-screen, high-resolution graphics
- Create animated sprite graphics
- Convert numbers from one base to another
- Mix BASIC and machine languages
- Program music and sound

This book is more than just a reference book. More than a 6502 Assembly manual. It's a hands-on, step-by-step guide to programming with three of today's popular assemblers: the Commodore 64, Merlin 64™, and Panther C64™. So whether you own a Commodore 64 or 128, all you need is your computer and this book. And you can quickly master the intricacies of Assembly language programming. Sams makes it easy!

Mark Andrews started his writing career as audio/video columnist for the New York *Daily News*. For three years, he had a nationally-syndicated weekly newspaper column about personal computers and other consumer electronics products. Now his articles appear in hundreds of magazines, newspapers, and books. Mr. Andrews has written seven books about computers.

Howard W. Sams & Co.
A Division of Macmillan, Inc.
4300 West 62nd Street, Indianapolis, IN 46268 USA

0 81262 22444 1

$15.95/22444

ISBN: 0-672-22444-5

C64™/128™ Assembly Language Programming Andrews SAMS™

SAMS™

Book Mark

Sams books cover a wide range of technical topics. We are always interested in hearing from our readers regarding their informational needs. Please complete this questionnaire and return it to us with your suggestions. We appreciate your comments.

1. Which brand and model of computer do you use?

☐ Apple ____
☐ Commodore ____
☐ IBM ____
☐ Other (please specify) ____

2. Where do you use your computer?

☐ Home ☐ Work

3. Are you planning to buy a new computer?

☐ Yes ☐ No

If yes, what brand are you planning to buy? ____

4. Please specify the brand/type of software, operating systems or languages you use.

☐ Word Processing ____
☐ Spreadsheets ____
☐ Data Base Management ____
☐ Integrated Software ____
☐ Operating Systems ____
☐ Computer Languages ____

5. Are you interested in any of the following electronics or technical topics?

☐ Amateur radio
☐ Antennas and propagation
☐ Artificial intelligence/expert systems
☐ Audio
☐ Data communications/telecommunications
☐ Electronic projects
☐ Instrumentation and measurements
☐ Lasers
☐ Power engineering
☐ Robotics
☐ Satellite receivers

6. Are you interested in servicing and repair of any of the following (please specify)?

☐ VCRs ____
☐ Compact disc players ____
☐ Microwave ovens ____
☐ Television ____
☐ Computers ____
☐ Automotive electronics ____
☐ Mobile telephones ____
☐ Other ____

7. How many computer or electronics books did you buy in the last year?

☐ One or two ☐ Three or four
☐ Five or six ☐ More than six

8. What is the average price you paid per book?

☐ Less than $10 ☐ $10-$15
☐ $16-$20 ☐ $21-$25 ☐ $26+

9. What is your occupation?

☐ Manager
☐ Engineer
☐ Technician
☐ Programmer/analyst
☐ Student
☐ Other ____

10. Please specify your educational level.

☐ High school
☐ Technical school
☐ College graduate
☐ Postgraduate

11. Are there specific books you would like to see us publish? ____

Comments ____

Name ____
Address ____
City ____
State/Zip ____

22444

SAMS™

Book Mark

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 1076 INDIANAPOLIS, IND.

POSTAGE WILL BE PAID BY ADDRESSEE

HOWARD W. SAMS & CO., INC.
ATTN: Public Relations Department
P.O. BOX 7092
Indianapolis, IN 46206

A SPECIAL OFFER FOR NEW OWNERS!

Get the most out of your Commodore computer.
. . . With Commodore magazines.
And save 20% off the regular newsstand price.

Subscribe to *Commodore Power/Play* and *Commodore Microcomputers* magazines and you're on your way to realizing the full power and potential of your new Commodore computer.

Each issue brings you new ways to use and enjoy your computer. The first word on new software and hardware. Programming techniques for both beginners and advanced users. In-depth product reviews of the best software and hardware. The latest games or education and applications programs. Visits with other users who have discovered new and interesting ways to use their Commodore computers.

You'll find practical articles on linking up with user groups in your area. Telecommunications and using on-line services such as CompuServe. Computer music and art, and much, much more.

In addition, every issue contains programs you can type in yourself and use right away. There's entertainment and games or practical household and business applications programs in each issue!

Together, they're the perfect combination of pure fun and productivity!

And if you take advantage of this special offer—only for new computer owners—you can save as much as 20% off the regular newsstand price!

Subscribe now and get the most out of your Commodore computer. And save as much as 20%!

————————————DETACH AND MAIL TODAY————————————

Please sign me up for

☐ year(s) of *Power/Play* and *Microcomputers* (12 issues total per year) at $24/year (a savings of 20% off the regular newsstand price).

☐ year(s) of *Power/Play* only (entertainment and games—6 issues per year) at $15/year.

☐ year(s) of *Microcomputers* only (more in-depth information about practical ways to use your computer—6 issues per year) at $15/year.

ALL PRICES IN US CURRENCY. Canadian add $5.00 to each subscription to cover postage. Overseas: $25.00/6 issues (includes postage).

Name ____________________ Phone ____________

Address ______________________________

City ____________ State ____________ Zip ____________

Signature ______________________________

METHOD OF PAYMENT

☐ Enclosed is my check or money order for $__________ *(Make check or money order payable to COMMODORE PUBLICATIONS)*

☐ Bill me

SAM1

☐ Charge my VISA or MasterCard *(circle one)* Card number

☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐ Expiration Date__________

or call 800-345-8112 to order (in Penna. 800-662-2444)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY LABEL

FIRST CLASS PERMIT NO. 251 HOLMES, PA

POSTAGE WILL BE PAID BY ADDRESSEE

Commodore Publications
Magazine Subscription Department
Box 651
Holmes, PA 19043