APPLE II GS MACHINE LANGUAGE FOR BEGINNERS

Roger Wagner

The understandable guide to learning 65816 machine language on the Apple II GS.
Dedication
Once upon a time, a man was called upon to write a book to teach others the things he had spent a lifetime learning. His wife and child waited patiently for the dream to become real. I know one of my father's greatest joys was the accomplishment of that book. The wheel has turned, and now I dedicate this book to the most patient people I know, and the three most important and loved women in my life: my mother, Bea, my wife, Pam, and my daughter, Marta.
# Contents

Foreword ................................................................. vii
Introduction ......................................................... ix

1. Applesoft BASIC and Beyond ................................. 1
2. Real Machine Language Programming ..................... 21
3. The Apple II GS Mini-Assembler ............................ 37
4. Assembling a Program with *Merlin 8/16* ............... 57
5. Assembling a Program with *APW* .......................... 79
6. Loops and Counters ............................................ 99
7. Comparisons in Assembly Language ....................... 123
8. Simple Math ................................................... 141
9. Logical and Shift Operators .................................. 157
10. Addressing Modes and Improved Printing ................. 177
11. Data Storage and Program Control ......................... 205
12. Adding Machine Language Programs ........................ 223
13. ProDOS ...................................................... 245
14. ProDOS 16 .................................................. 269
15. A Look at Memory Use on the Apple II GS ............... 287
16. The Apple II GS Toolbox .................................... 309
17. The Memory Manager and Miscellaneous Tools ........... 335
18. QuickDraw and the Event Manager ......................... 365
19. The Window and Menu Managers ............................ 401
20. A Drawing Program for the Apple II GS ................... 445

Appendices ........................................................... 471
A. 65816 Instructions ............................................. 473
B. The Apple II GS Monitor ..................................... 589
C. ProDOS 16 File Dump Utility ............................... 593
D. Suggested Reading ............................................ 599
E. ASCII Character Chart ........................................ 601

Index ................................................................. 605
Disk Coupon .......................................................... 611
Foreword

This is the definitive guide to programming in 65816 machine language on the Apple IIGS. It's a thorough introduction to machine language programming on the newest Apple II, a comprehensive tutorial for beginning machine language programmers, and an invaluable reference guide for the most experienced software developer.

Roger Wagner's name is well known to Apple users and programmers. He is a popular columnist, a software developer with his own publishing company, and the author of the bestselling Assembly Lines: The Book. He has used his years of Apple II experience to put together COMPUTE!'s Apple IIGS Machine Language for Beginners, a book for both experienced and beginning machine language programmers.

Clearly written, with dozens of practical examples to show the way, this book includes more than the basics of machine language programming—it's a specific reference and guide to programming in 65816 machine language on the Apple IIGS. For instance, you'll find complete chapters on using the built-in assembler and on how to write and assemble source code with two popular assemblers, the Merlin 8/16 and the APW.

Other topics covered include ProDOS 8 and 16, Menu Manager, the Toolbox, the Memory Manager, QuickDraw, the Event Manager, the Window Manager, and how to add machine language to Applesoft BASIC programs. COMPUTE!'s Apple IIGS Machine Language for Beginners also contains exhaustive appendices that you can refer to in an instant. Among the appendices is a comprehensive 65816 instruction set.

Written for Apple IIGS programmers of every level of experience, COMPUTE!'s Apple IIGS Machine Language for Beginners is the most complete guide and reference to 65816 programming around. If you want to learn how to program in machine language on the IIGS, or if you're already creating software masterpieces in machine language, you need COMPUTE!'s Apple II Machine Language for Beginners.

All the programs in this book are ready to type in and use. There is also a disk available from COMPUTE! Books which includes all the source code from the book. An assembler is required to use the disk. To purchase the disk, use the coupon in the back of the book.
Some BASIC programmers believe that machine language is some very difficult and obscure language used only by advanced programmers. As it happens, it’s just different, and if you have successfully used Applesoft BASIC to write some of your own programs, there’s no reason why you should have any difficulty learning to write machine language programs.

This book will meet you at your current understanding of Applesoft BASIC and will introduce you to the fundamental principles of machine language programming. Your knowledge of BASIC will provide a valuable foundation for learning, and you’ll even notice some similarities between BASIC and machine language programs. This is no accident. Part of BASIC’s original purpose was to be an educational tool in the teaching of machine language programming.

Using the BASIC environment as a starting point, you’ll learn about memory organization and use, and you’ll learn how to add simple machine language routines to your existing BASIC programs. Each new concept will be introduced as an extension to ideas already presented, making the learning process as absolutely clear and simple as possible. By the time you’re done, you’ll be able to write your own Apple IIGS machine language programs—ones completely independent of BASIC—and you’ll be able to use many advanced Apple IIGS features such as super hi-res graphics, the mouse, menus, and windows.

Because the Apple IIGS is an amazingly complex machine, it’s impossible for one book to provide every last detail of its operation. There are very large books written on just some of the many topics covered in this book. For example, the 700-page Apple IIGS Technical Reference, by Michael Fischer, is about the IIGS Tools only; and the 600-page Programming the 65816, by David Eyes and Ron Lichty, is an encyclopedic reference dealing with just the 65816 instructions. The goal of this book is not to provide the last word about each 65816 instruction, or on every GS Tool call. Instead, the goal is to provide information that will build the foundation essential for your own further efforts.

When you’ve completed this book, you’ll have gained the knowledge and confidence necessary to write your own programs. The many examples of executable programs included will provide valuable subroutines and procedures that you can incorporate in your own programs.
Introduction

What You Need

Like any creative process, having the right tools and materials can make all the difference in the world. We’re going to make some assumptions about what materials you have as you read this book.

Obviously, having an Apple IIGS on hand while you work through the examples is important. You’ll also need to have a disk drive to save your programs, although it doesn’t matter whether you have a floppy disk of any particular brand or capacity—in fact, even a hard disk is OK. Apple Computer’s ProDOS will be used in those chapters that deal with file operations.

When a person writes a letter or report on a computer, a word processor is the right tool for the job. When writing machine language programming, a good assembler is what you need. An assembler is essentially a word processor specifically designed for writing machine language programs. It also does many other things, so you’ll need one to follow the examples in this book. Most of the examples use the Merlin 8/16 assembler, but there is also information on the APW (Apple Programmer’s Workshop) assembler. I prefer the Merlin 8/16 in terms of ease of use, speed of assembly, and minimum explanation required for a beginning programmer; but you may use any assembler capable of assembling 65816 instructions.

It’s also important, especially as you do more programming, to have a solid reference library. For beginners, this book has all the information necessary to complete the examples, but you may wish to look at books listed in the bibliography for further reading material.

Program Library

In the early days of programming, programmers had to write programs that were completely self-contained, ones in which every operation of the program was covered by instructions the programmer had specifically written into the program.

It soon became apparent, however, that time could be saved in program development if the most frequently used routines were available in a library that could be incorporated as the program was being written. That way, things like disk access, printing to the screen, reading the keyboard, and so forth, didn’t have to be rewritten every time a new program was started—programmers no longer have to “reinvent the wheel.”

In the Apple IIGS, a great number of useful routines and operations are not only already written for you, but are also built right into the machine,
much the same way that Applesoft BASIC is already there. Thus, your library is right in the machine. You don’t have to add hundreds of lines of program instructions from a disk file when you write a program; a simple call to a subroutine in the computer does the trick. Your program is smaller, development time is shorter, and writing a program is in general much more satisfying.

One of the most valuable lessons you can learn from this book, beyond the simple commands that make up machine language, is how to use the routines already existing in the machine that make writing your programs much easier. You should also make a point of saving every program you write, so you can build your own collection of program operations for use in later programs as you create them.

**Just for the Fun of It**

One last warning before you start: Don’t be discouraged if you’re not an overnight expert in assembly language programming.

Many people are discouraged when learning a new skill because they expect instant results. Perhaps you know someone who got interested in art or music, and then became disappointed when their first attempts weren’t beautiful works of art or they weren’t able to play their favorite melody by just picking up the instrument.

Even some schools lose sight of the fact that not everything has to be justified by monetary or productivity standards. They teach only word processing and spreadsheets because these are the “practical” uses for a computer, and downplay the value of programming or using the computer for fun. Yet these same schools have no problem offering shop and art classes, even though very few students will go on to become professional carpenters or artists.

The key to your own success is to set achievable goals, and to do things that you find personally rewarding. Don’t worry that the first program you write isn’t likely to be immediately publishable as a commercial product, or that it isn’t big enough to do something “really important.” By starting with a small project like drawing a single line on the screen, you’ll have chosen a project that will teach you something, and in completing it, you’ll gain the confidence and sense of satisfaction that will make you want to start your next project. As your experience grows, one day you’ll suddenly realize that you’ve learned how to write all sorts of great programs and that programming has become more fun than work.

The bottom line is this: The most important rule in programming is that you enjoy it. As in learning any subject, motivation and taking the time to experiment are more important than genius or prior experience. If you have the former, everyone will think you have the latter.
Chapter 1

Applesoft BASIC
and Beyond
A good starting point for learning machine language is to take a look at something you’re already familiar with, namely Applesoft BASIC. As it happens, Applesoft BASIC is a machine language program. When your program executes the command PRINT “HELLO”, it’s a machine language program that puts the characters on the screen. It is also possible to call custom-made machine language routines from within an Applesoft BASIC program. If this is all true, why learn machine language in the first place?

What Is Machine Language and Why Bother Learning It?

As discussed earlier, the primary reason for learning machine language is that you’ll find the subject interesting and fun to experiment with. There are some more tangible reasons, though, that will make learning machine language more useful than, say, learning ancient Latin. One of the main reasons is speed. The Apple II GS can execute about 100,000 machine language instructions per second, far more than Applesoft BASIC can execute, even when it’s running in fast mode.

Another reason to learn machine language is flexibility. Applesoft BASIC is an artificial environment that stands between you and the most fundamental level of your computer. When you type PRINT “HELLO” in a BASIC program, you really don’t have any idea what causes the letters to appear on the screen. In machine language, you do. In addition, BASIC is limited to the different kinds of information, called data structures, that it can easily deal with. For example, in BASIC you can have string variables with lengths up to 255 characters. Suppose you want to store a paragraph with 1000 characters in it—Applesoft BASIC has no direct variable defined for paragraph. Or suppose you want your program to handle graphics objects like points, rectangles, ovals, and so forth. Again, no variable types exist for these.

In machine language, there is not specifically any data type; you still have to create these. But you do have the flexibility to create any kind of data structures you want.
On the Apple IIgs, another good reason to program in machine language is to have access to all those features, such as super hi-res graphics, that are built into the machine, but are not directly accessible from Applesoft BASIC. With machine language, these are relatively easy to use. In fact, an interesting first application of your machine language programming skills can be using them to create a bridge between Applesoft BASIC and the IIgs Tools.

Finally, one of the best reasons to learn machine language is to gain a better idea of how a computer actually works. In machine language, you're dealing with the most fundamental levels of operation in the computer. You'll see how the microprocessor actually runs a program, how memory is used, how the hardware interacts with a program, and more. This knowledge will also make you more flexible as the hardware evolves and changes in the future, and it provides an excellent foundation should you ever wish to program on other computers.

A World of Languages

Programming is just a word to describe the process of telling your computer what you want it to do. In fact, this is the real power of a computer as a modern device. The beauty of a computer is that you can tell it how to help you solve problems that are unique to your own life, on your own terms, and at your convenience.

How hard is it to program a computer? Not as hard as you might think. The only requirement is a language with which you can communicate with your computer. It doesn't understand English directly, but it's also not always necessary to use BASIC, PASCAL, or other formal computer languages. Some of the most successful commercial programs for the computer are just well-disguised programming languages.

As an example, consider the classic spreadsheet program. It's always fun to hear someone say "Oh, I don't program. I just use programs like the Acme Spreadsheet." With a little thought, you realize that using a spreadsheet is much like programming. The user creates a set of numbers, variables and equations that are used in a predictable order by the computer. Some more advanced spreadsheet programs have enhancements like search and sort functions, IF-THEN testing, and more. It's programming all right—just in a different language than we usually think of.

Any language consists of a collection of words with specific meanings. In programming, these words are usually commands that make the computer do something in particular, such as add numbers, clear the screen, and so on.

In Applesoft BASIC, there are just over 100 different commands to learn, expressions like PRINT, VTAB, INPUT. Once you've learned the commands,
you can put them together in a certain order and create a program.

In machine language, there are still only about 100 commands to learn, and many fall into groups of similar functions that make learning them easier still.

**How It Really Works**

**Lesson #1 starts with Wagner's Paradox:**

"Everything complex can be broken down into simple elements (and nothing is as simple as it seems)."

Although sometimes it feels like life is an endless case of discovering the fine print, it’s true that most things in life are quite simple when you consider the fundamental things that make them up. The computer is an excellent example.

Even though the engineering required to build a computer is awe-inspiring, the underlying principle of their operation is almost trivial.

The heart of your Apple IIGS is something called the 65816 microprocessor. Another important part is its memory—thousands of places in the computer where a simple number can be stored. Each memory location can store an arbitrary number value in the range of 0 to 255. Number values larger than 255 must use a combination of bytes to store the number value. That’s why, in Applesoft BASIC, you get an “ILLEGAL QUANTITY ERROR” if you try to use the POKE statement with a value larger than 255, such as in POKE 768,1000. (Try this if you’ve never done it.)

The 65816, like its cousins in other computers, works on the idea of scanning through memory, one location at a time, and performing some action depending on what number value it finds stored there.

If it finds a 27 in one place, perhaps it will add some numbers; if it finds a 32, it will subtract them. In the 65816, as was mentioned earlier, there are about 100 general operations that the microprocessor will do, depending on what number it finds in a given memory location. A program is created by putting the possible commands in a certain meaningful order to tell the computer how to do a certain task.

Obviously, the more memory a computer has, the more instructions (and information) it can hold. In the world of Applesoft BASIC, which was designed for an earlier microprocessor called the 6502, there are about 65000 memory locations in which various numbers can be stored. Each memory location is called a *byte of memory* by most programmers. When you hear someone talk about 1 *megabyte*, they’re talking about one million separate memory locations.
in the computer. The 65816, a newer descendant of that original 6502, can ad-
dress up to 16 million bytes of memory (16 megabytes).

In Applesoft BASIC, however, which was created for the Apple IIs, we
can only talk to 64K at a given time. (There are actually 65,536 locations in the
computer. In the metric system, K is used as an abbreviation for thousand, but
in computer jargon K refers to 1024 bytes.)

**BASIC vs. Machine Language**

The 65816 can directly interpret the numbers stored in memory as a program—
a sequence of instructions. This sequence of number-instructions is called ma-
chine language. Machine language, or ML, is the actual series of numbers in
memory that the microprocessor can directly act on. There are no handy words
like PRINT or HOME to make life easier for the human creating the program.
Machine language programming in its literal sense is the process of placing
numbers in memory, one at a time, to create a program that the computer can
understand and carry out. *Assembly language*, as we’ll see shortly, is actually
what most people mean when they say they program in machine language. As-
sembly language itself is an extension beyond true machine language—but
more on that later.

BASIC is sometimes called a high-level language. One way of looking at
this term is that BASIC commands, like PRINT, are closer to English than more
simplistic languages (like machine language). What it really means, though, is
that a new command has been created, behind the scenes, from the primary
100 commands at the machine language level, to do something fairly complex.

An interesting analogy might be to consider that all the words you know
are created with the 26 letters of the alphabet. As with human language, an in-
finite number of specific high-level computer commands can be created from
the 100 fundamental commands that are built into the computer.

In its native mode, just the 65816 and some memory, the computer
doesn’t really understand words like PRINT. That’s where the idea of Applesoft
BASIC as language in itself comes in.

When you type in RUN to start an Applesoft BASIC program, you’re ac-
tually triggering a machine language program in the computer that acts as a
middleman between the lines of BASIC that you typed in and those 100 com-
mands that the 65816 understands.

When your program says PRINT "HELLO", this intermediate program
looks up the word PRINT in a list of commands, and then executes a short,
built-in ML program that prints HELLO on the screen.

So, actually, Applesoft BASIC really is machine language. Each time a
statement in your Applesoft BASIC program is executed, the computer carries
out a short machine language program (or more properly, a subroutine).

The main reason Applesoft BASIC runs more slowly than a pure machine language program is that it takes the Applesoft BASIC interpreter (the middleman) time to decide which routine to execute for each command in the various lines of BASIC.

**Peeking at Maps and Addresses**

If you know how to write even a simple program in Applesoft BASIC, you’re well on your way to knowing how to write a program in machine language.

For starters, we mentioned earlier that there are 65,536 locations in the original Apple II computers to store the parts of a machine language program. What a coincidence—line numbers in BASIC cover the same range.

In BASIC you identify a place in your program with a line number. For example, to jump to a given routine, you might use the statement GOSUB 1000.

In machine language, each location is identified with an address. Computer people start counting with zero for the first item, so the addresses of those locations count 0, 1, 2, 3—up to 65,535. (65,535 + the 0 byte = 65536).

Just like the addresses of the houses on a street, each address of a location in memory lets you find one and only one spot in the computer to look at or store a number in.

A useful chart can be made to show a diagram of the various memory locations in your Apple, and to show what different parts of memory are used for. Such a chart is called a memory map. Figure 1-1 is a simple memory map that represents the memory that Applesoft BASIC uses in the Apple IIgs.

This memory map represents each single available memory location in the first 64K (65536) of memory. The zero byte is shown at the bottom, but this is arbitrary. It’s also important not to confuse the contents of a memory location with its address. We’ve already said that the value of a number stored in a byte of memory cannot be larger than 255. However, the address of a given byte can be virtually any number at all, as shown by the memory map.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>65535</td>
<td></td>
</tr>
<tr>
<td>65534</td>
<td></td>
</tr>
<tr>
<td>65533</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>32768</td>
<td></td>
</tr>
<tr>
<td>32767</td>
<td></td>
</tr>
<tr>
<td>32766</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>255</td>
<td></td>
</tr>
<tr>
<td>254</td>
<td></td>
</tr>
<tr>
<td>253</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

---

Figure 1-1.
A Simple Apple IIgs Memory Map
Another common way of drawing a memory map is horizontally, with low address values on the left, and the top of memory on the right, such as the one shown in Figure 1-2.

Figure 1-2. Horizontal Memory Map

In normal Applesoft BASIC, there are two commands in particular, POKE and PEEK, that let you look at, and usually change, the contents of any of the first 64K of memory in your computer. These commands are the first links to the world of machine language.

Many times, a machine language program will use the contents of some particular memory location to store a meaningful value or to look at the contents of a location to see what action should be taken.

The Applesoft BASIC PEEK command is used to determine the contents of a given memory location in the Apple.

By using the PEEK command to examine different memory locations, you can find out some useful things not normally accessible by the usual BASIC commands. For example, memory location 33 holds the width of the screen. Suppose you wanted to write a general-purpose program which would work on anybody's computer, in either 40 or 80 columns. By examining memory location 33, your program could tell which mode is active when it is run. For example:

10 TEXT: HOME
20 CW = PEEK(33): REM DETERMINE COLUMN WIDTH
30 T$ = "TITLE OF THIS PROGRAM"
40 HTAB CW/2 - LEN(T$)/2: REM CENTER TITLE
50 PRINT T$: REM PRINT CENTERED TITLE

This program will always print a centered title on the screen, no matter what the video display mode.

Another useful PEEK is at memory locations 218, 219, and 222. In an Applesoft BASIC program, errors can be trapped by the statement ONERR GOTO early in the program. Using this command, for example, Control-C can appear to be ignored by the running program. Usually, Control-C will stop a running program, but with ONERR, you can trap the error and continue program execution. Program 1-1 is a short example.
Program 1-1. ONERR Example 1

5 HOME
10 ONERR GOTO 100
20 X = 1
30 VTAB 1:HTAB 1: PRINT X
40 X = X + 1
50 IF X < 10000 THEN 30
60 END
100 EC = PEEK (222) : REM ERROR CODE
110 EL = PEEK (218) + 256 * PEEK (219) : REM ERR LINE #
120 IF EC = 255 THEN VTAB 12: HTAB 1: PRINT "I'M NOT DONE YET!"
130 VTAB 1:HTAB 1:RESUME

In this program, line 10 tells Applesoft BASIC to jump to line 100 if any error occurs. Whenever any error occurs in a running program, Applesoft BASIC always stores a code value for the error in location 222. It also stores the value for the line number.

Storing the line number, however, creates a new problem. We mentioned earlier that a single byte of memory could only store a value in the range of 0 to 255. Since line numbers can have any value from 0 to 65535, how can Applesoft BASIC store the number? It uses two bytes. The system is a little strange, though. First, Applesoft BASIC divides the line number the error occurred in by 256 (a number you'll see a lot in machine language programming). It stores the remainder in the first memory location—218—and then stores the result in the next memory location, 219.

For example, if you press Control-C while the sample program is on line 40, Applesoft BASIC stores 40 (the remainder) in location 218, and it stores 0 (40 divided by 256 = 0, remainder 40) in location 219.

If you had a large program, and an error occurred when the program was executing line 600, then locations 218 and 219 would hold the values 88 and 2, respectively (600 divided by 256 equals 2, remainder 88).

You may wonder why we divide by 256. The main reason is that dividing a large line number like 63000 by any other number produces results or remainders larger than 255, and we couldn't store the result or remainder in a single byte. Using 256 as the divisor makes everything work smoothly. And one byte can hold any one of 256 different numbers (0–255).

Now, back to the error handling routine. To reconstruct the line number that the error occurred on, the program needs to multiply the value at location 219 by 256, and then add the result to the remainder value at 218. Line 110 does this for purposes of illustration, although our program in particular doesn't use the result.
Chapter 1

Finally, line 120 tests the error code to see if the error was caused by pressing Control-C (255 is the error code for Control-C). Any other error, such as a syntax error, generates a different code.

When trying out this program press Control-C several times in a row. You’ll notice the value for X at a given point is sometimes printed below the error message when you press Control-C. This is because RESUME re-executes the last statement executed, not the last complete line.

**POKEing Around in Memory**

In Applesoft BASIC, a POKE is used to put a particular number value (always in the range of 0 to 255) into a particular memory location (in the range of 0 to 65535).

For example, the sample program just presented has a drawback: It ignores all errors, even typographical errors in the listing. If a syntax error due to a typing mistake occurs, the program gets stuck in an endless loop—resuming the line with the error, going to the ONERR routine, and resuming again. Try it—retype line 50 as

```
50 IF X PRINT 10000 THEN 30: REM DELIBERATE ERROR
```

Line 100 can tell the kind of error, but just how can we turn off the ONERR trap?

The answer is to change memory location 216. This location is set with a certain number value when the ONERR GOTO statement is executed, and Applesoft BASIC uses this value as a flag, or indicator, of when to trap errors. If our program could reset location 216 to 0 (the ONERR off value), errors other than Control-C would be properly handled—the program would stop and a message would be printed. While we’re at it, let’s add the instruction GOTO 30 to line 120 so as to re-execute the entire line when Control-C is pressed.

Program 1-2 is the revised listing.

```
5. HOME
10 ONERR GOTO 100
20 X = 1
30 VTAB 1:HTAB 1: PRINT X
40 X = X + 1
50 IF X < 10000 THEN 30
60 END

100 EC = PEEK (222) : REM ERROR CODE
110 EL = PEEK (218) + 256 * PEEK (219) : REM ERR LINE #
120 IF EC = 255 THEN VTAB 12: HTAB 1: PRINT "I'M NOT DONE YET!": GOTO 30
```
First, try this program as shown to verify that it works the way the first sample listing did. Then retype line 50 incorrectly as

```
50 IF X PRINT 10000 THEN 30 : REM DELIBERATE ERROR
```

and verify that the improved listing can distinguish between Control-C and other errors. By expanding the list of IF-THEN tests for different error codes, you can make your programs selectively respond to a wide variety of errors at different parts in your program.

**Using Applesoft BASIC's CALLs**

The other, and most important, link to machine language programming from Applesoft BASIC is the CALL command.

If you've ever used a CALL statement in Applesoft BASIC, you've already done the machine language equivalent to BASIC's GOSUB command. For example, a CALL 32768 from BASIC tells the computer to start running a machine language program at a certain location in memory (location number 32768). As long as there is a machine language program there—and that program eventually ends with the usual RETURN code (or more accurately, its ML equivalent)—then, when the routine is finished, program control will return to your Applesoft BASIC program (Figure 1-3).

---

**Figure 1-3. GOSUB vs. CALL**

```
Applesoft BASIC Program
(GOSUB XXX)

(RETURN)

Applesoft BASIC Program

(MACHINE LANGUAGE PROGRAM)
```

---

```
Applesoft BASIC Program
(CALL XXX)

(RETURN)
```

---
Chapter 1

For now, it’s important to understand that where a machine language program is located in the computer (its address) is as important as the line numbers in a BASIC Program.

**Instant Machine Language Programming**

One of the first, and best ways to start using machine language programming techniques in your own programs is to just use Applesoft BASIC’s CALL command to execute short machine language routines that either already exist in the computer or that you have written yourself.

We mentioned already that there were quite a number of preprogrammed routines built into your Apple computer that Applesoft BASIC uses for its own commands. Knowing some of these, like knowing some of the special memory locations to PEEK and POKE, can enhance your existing Applesoft BASIC programs.

For instance, did you wonder why the sample ONERR program didn’t use a FOR-NEXT loop? To find out, try running the program written using a FOR-NEXT loop (Program 1-3) and pressing Control-C.

Program 1-3. ONERR Example 3

5 HOME
10 ONERR GOTO 100
20 FOR X = 1 TO 10000
30 VTAB 1: HTAB 1: PRINT X
40 NEXT X
60 END
100 EC = PEEK (222) : REM ERROR CODE
110 EL = PEEK (218) + 256 * PEEK (219) : REM ERR LINE #
120 IF EC = 255 THEN VTAB 12: HTAB 1: PRINT “I’M NOT DONE YET!”: GOTO 30
125 POKE 216,0 : REM TURN OFF “ONERR”
130 VTAB 1: HTAB 1: RESUME

When you press Control-C in this program, you’ll find that, although the code for handling the Control-C works OK, when the program tries to continue the FOR-NEXT loop, you get a NEXT WITHOUT FOR error. That’s because Applesoft BASIC forgets where it’s up to in the FOR-NEXT loop when the error occurs. Does this mean you can never have an ONERR trap with a GOTO in a program that uses FOR-NEXT loops? It does—if you don’t know a special CALL that you can do to fix the problem. Program 1-4 is the revised program.
Program 1-4. ONERR Example 4

5 HOME
10 ONERR GOTO 100
20 FOR X = 1 TO 10000
30 VTAB 1:HTAB 1: PRINT X
40 NEXT X
60 END

100 EC = PEEK (222) : REM ERROR CODE
110 EL = PEEK (218) + 256 * PEEK (219) : REM ERR LINE #
115 CALL 62248: REM FIX ONERR HANDLING
120 IF EC = 255 THEN VTAB 12: HTAB 1: PRINT "I'M NOT DONE YET!": GOTO 30
125 POKE 216,0: REM TURN OFF "ONERR"
130 VTAB 1: HTAB 1:RESUME

The added line, 115, does a CALL 62248 that restores Applesoft BASIC's memory (so to speak) about where in the FOR-NEXT loop it was when the error occurred. CALL 62248 calls a part of the Applesoft RESUME routine that fixes the internal Applesoft information about any pending FOR-NEXT loops. If the error doesn't occur in a FOR-NEXT loop, the CALL 62248 doesn't hurt anything.

There are a number of useful CALLs that you can do in an Applesoft BASIC program to routines already present in your computer. Here's a short list of some of those ML routines that you can call right from Applesoft BASIC.

<table>
<thead>
<tr>
<th>Address to Call</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>64538</td>
<td>Move cursor up.</td>
</tr>
<tr>
<td>64614</td>
<td>Move cursor down.</td>
</tr>
<tr>
<td>64528</td>
<td>Move cursor left.</td>
</tr>
<tr>
<td>64500</td>
<td>Move cursor right.</td>
</tr>
<tr>
<td>64780</td>
<td>Wait for a keypress.</td>
</tr>
<tr>
<td>64858</td>
<td>Wait for a RETURN keypress.</td>
</tr>
<tr>
<td>64668</td>
<td>Clear to end of line from cursor.</td>
</tr>
<tr>
<td>64578</td>
<td>Clear to bottom of screen from cursor.</td>
</tr>
</tbody>
</table>

Looking at the list, you'll notice all the addresses have rather high values. That's because Applesoft BASIC uses a large number of machine language routines starting at location 53248. The particular routines in this list start at 64500 and above, but you'll recall the ONERR fix was a CALL to location 62248. Figure 1-4 is a memory map that shows where Applesoft BASIC ROM is located.
The illustration indicates that the Applesoft BASIC routines are ROM routines. You’ve probably heard the terms RAM and ROM before. (ROM stands for Read Only Memory; RAM stands for Random Access Memory.) Your computer has both RAM and ROM in it. The difference is that RAM not only stores a number value, but the number value can be changed by writing a new value to that location at any time. That’s what a POKE in Applesoft BASIC does: It writes a new number value to a given memory location. ROM memory, on the other hand, can only be read. That is, you can PEEK to see what’s there, but a POKE will not change the contents.

As an example, try this short BASIC program:

```
10 PRINT PEEK (62000): REM SHOW WHAT’S AT LOC. 62000
20 POKE 62000,0 : REM TRY TO CHANGE IT
30 PRINT PEEK (62000): REM SEE IF IT CHANGED.
40 END
```

What you should find is that, no matter how hard you try to change the contents of memory location 62000, it always holds the same value.

The problem with RAM is that, generally speaking, when you turn off the computer’s power, the contents of RAM are erased; therefore, you can’t store anything permanent there. But there are some things, like Applesoft BASIC, that the computer designers wanted to keep in the machine while the power is off; these are stored in ROM.

As a side-note: If we explain this much further, the latter part of Wagner’s Paradox will rear its ugly head. You’ve probably already noticed that both RAM and ROM can access any given byte at a time, so ROM is technically random access memory, too (as opposed to sequential access memory, where you have to look at each byte in a series before you can examine each successive byte). Also, when the power is off, your Apple II GS remembers changeable things like the clock time and your Control Panel settings, and these have to be written somewhere to be stored. If you can’t write to ROM, it must be RAM that they are written to. But we just said RAM lost its contents when the power is turned off.
off. The designers cheated by hooking up a battery to a small (256) number of bytes of memory to keep a few things, like the time and the Control Panel settings, on hand while the power is turned off.

Since ROM routines are always in the machine at the same place all the time, you can CALL them from your Applesoft BASIC program, just like the ONERR fix. To see how these might be used, let's consider some more short examples.

**Waiting for a keypress.** Usually, in an Applesoft BASIC program, when you want the program to wait for a keypress you must use GET A$ or something similar to get the character from the keyboard. The only disadvantage is that a flashing cursor appears on the screen. Suppose you want to write a program that waits for a keypress, but doesn't put a flashing cursor on the screen. Using CALL 64780 from the previous chart, you could write a program like Program 1-5.

Program 1-5. Waiting for Keypress

```basic
10 HOME
15 PRINT "WAIT FOR KEYPRESS DEMO"
20 VTAB 12: PRINT "PRESS A KEY TO CONTINUE..."
25 CALL 64780
30 HOME
35 VTAB 12: PRINT "THANK YOU!"
40 END
```

**Moving around.** This next program, Program 1-6, demonstrates how to move the cursor around on the screen, depending on what key is pressed. (These calls only work from 40-column mode.)

Program 1-6. Moving the Cursor

```basic
10 HOME: PRINT CHR$(17): REM 40-COLUMN MODE
20 HTAB 20:VTAB 12:REM CENTER CURSOR
30 GET A$
40 IF A$="U" THEN CALL 64538
50 IF A$="D" THEN CALL 64614
60 IF A$="L" THEN CALL 64528
70 IF A$="R" THEN CALL 64500
80 IF A$="Q" THEN END
90 GOTO 30
```

**Calling them by name.** Another trick we can add to an Applesoft BASIC program to make the CALLs more understandable is to set Applesoft BASIC variables equal to the address you want to call. By making the names of
the variables a clue to the function of the routine itself, your program becomes more understandable.

For example, let’s take that last program, and change it just a bit (Program 1-7).

Program 1-7. Using Meaningful Variable Names

```
5 UP=64538: DOWN=64614:LEFT=64528:RIGHT=64500
10 HOME: PRINT CHR$(17): REM 40-COLUMN MODE
20 HTAB 20:VTAB 12:REM CENTER CURSOR
30 GET A$
40 IF A$="U" THEN CALL UP
50 IF A$="D" THEN CALL DOWN
60 IF A$="L" THEN CALL LEFT
70 IF A$="R" THEN CALL RIGHT
80 IF A$="Q" THEN END
90 GOTO 30
```

In naming the routines in Applesoft BASIC, remember that only the first two letters of the name are really used by Applesoft BASIC. This means you can’t use two different routines called MOVEUP and MOVEDOWN, for example. Also, you have to use legal variable names, and you must avoid the use of Applesoft BASIC keywords. You can’t create a variable called PRINT, for example.

**Hardware Locations and Softswitches**

There is one other area of memory you should be aware of, the range from 49152 to 53247. This area of memory contains addressable locations that are not necessarily either ROM or RAM. Instead, many of the memory locations in this area are direct electrical connections to part of your computer’s hardware. The computer is designed so a program can examine certain of these locations to determine the status of physical parts of the computer, such as the keyboard, and to also change other parts that are hardware controlled, such as the speaker, text and graphics display, and other functions. Figure 1-5 shows the location in memory of hardware and softswitches.

**Figure 1-5. Memory Locations of Softswitches**

<table>
<thead>
<tr>
<th>Hardware &amp; Softswitches</th>
<th>Applesoft BASIC ROM Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>49152</td>
</tr>
<tr>
<td>(to 53247)</td>
<td>53248</td>
</tr>
<tr>
<td></td>
<td>65535</td>
</tr>
</tbody>
</table>
For example, suppose you want to know if someone were pressing the Open Apple key on the keyboard. Since a GET A$ and an INPUT A$ (the usual methods of getting characters from the keyboard) won’t work, you need some other alternative. Fortunately, memory location 49249 is directly connected to the Open Apple key. In this case, we use the term *memory location* in a very general sense. Although we can gain information with a PEEK (49249), there really isn’t any RAM or ROM there—just a wire to the keyboard.

To see how to use this in a program, try this sample program:

```plaintext
10 PRINT "PRESS THE OPEN APPLE KEY..."
20 IF PEEK (49249) < 128 THEN 20
30 PRINT "THANK YOU!"
40 END
```

You’ll notice that what we’re looking for is for the apparent “contents” of location 49249 to reach a value greater than 127. If you were to print the contents on a continuous basis (say in a program loop), you’d see all kinds of different values. The important change that takes place when the Open Apple key is pressed is that although a wide range of different values will still be seen, all of them will have a value greater than 127. Later on, you’ll see what’s so special about the values 127 and 128.

There is also another group of these apparent memory locations that are called *softswitches*. A softswitch is an addressable switch that will change the state of something in the computer just by accessing the location.

For example, if you want to switch from the text display to a view of whatever is on the hi-res page, the steps in Program 1-8 will do the trick.

**Program 1-8. Switching to Hi-Res**

```plaintext
10 PRINT “PRESS A KEY FOR THE HI-RES DISPLAY...”
20 GET A$
30 POKE 49239,0: REM HI-RES SWITCH
40 POKE 49232,0: REM GRAPHICS DISPLAY
50 POKE 49235,0 : REM MIXED DISPLAY
60 VTAB 22: PRINT “PRESS A KEY FOR TEXT AGAIN...”
70 GET A$
80 POKE 49233,0: REM BACK TO TEXT
90 END
```

There are two separate softswitches that are used to control hi-res graphics. The first, 49239, is a control switch that tells the computer that you want hi-res (as opposed to lo-res) graphics. However, setting this switch doesn’t actually change the display. It’s the other location, 49232, that switches the display from text to graphics.

Line 80 uses location 49233 to switch back to text. Many programs that
Chapter 1

let you switch between text and graphics views use POKEs (or an equivalent) to switch the display.

It is important to note that it's the accessing of the location that does the switch, not any actual manipulation of the data there, because there is no RAM to be changed. You could just as easily use a POKE 49233,157 on line 80—it wouldn't make any difference. Zero is used just to keep things simple in the listing, but the computer doesn't care what you use.

What You’ve Learned So Far

The important thing so far is to understand that there are many levels of programs in your Apple computer. You’re familiar with Applesoft BASIC, but ultimately, the computer actually runs series of number values stored in memory, called a machine language program.

A machine language program is called by jumping to a given address, rather than a line number as in BASIC. Most machine language routines are like BASIC subroutines, and they eventually return control back to the point from where they are called. This means that they can be used with a CALL statement from within an Applesoft BASIC program, just like BASIC subroutines, to enhance the programs you are writing today.

Not every memory location has to contain an actual program, however. Some locations store flags to indicate the status of something. Others contain data, such as the words to print on the screen, and some may not be "real" memory at all, but instead are hardware locations that control certain computer functions or tell you something about what's going on in the system.

The Applesoft BASIC commands POKE and PEEK can be used to both alter the contents of memory and to examine any given memory location to see what's already there. POKE can also be used to access hardware softswitches to change things like the screen display.

You may want to go back to programs you've seen in magazines, or on your local user group program disks, and look for POKEs, PEEKs, and CALLs to see how these are used in many different Applesoft BASIC programs for a variety of results.

Secret #1

You can use POKEs to change the screen, background, and text colors on your Apple IIGS right from within an Applesoft BASIC program.

Here's a program that asks for new color screen values, and then changes the system accordingly.
0 REM SCREEN COLOR DEMO
5 DIM C$(15): FOR I = 0 TO 15: READ C$(I): NEXT I
10 SV = PEEK (49186): REM SCREEN VALUE
15 TX = INT (SV / 16): REM TEXT COLOR VALUE
20 BK = SV - (TX * 16): REM BACKGROUND COLOR VALUE
25 REM
30 BV = PEEK (49204): REM BORDER VALUE
35 HV = INT (BV / 16): REM OTHER HARDWARE VALUES
40 BC = BV - (HV * 16): REM BORDER COLOR VALUE
45 OS = SV: OB = BV: REM SAVE ORIG. VALUES
50 TEXT : HOME
55 PRINT "BACKGROUND COLOR IS: ";C$(BK)
60 PRINT "TEXT COLOR IS: ";C$(TX)
65 PRINT "BORDER COLOR IS: ";C$(BC)
100 REM GET NEW COLORS
105 PRINT
110 INPUT “ENTER VALUE FOR BACKGROUND: (0-15)”;BK$:BK$ = VAL (BK$): IF BK < 0 OR BK > 15 THEN 110
115 INPUT “ENTER VALUE FOR TEXT: (0-15)” ;TX$:TX$ = VAL (TX$): IF TX < 0 OR TX > 15 THEN 115
120 IF BK = TX THEN PRINT : PRINT CHR$ (7); "YOU WON'T BE ABLE TO SEE WHAT": PRINT "YOU ARE TYPING!": GET A$: GOTO 100
125 INPUT “ENTER VALUE FOR BORDER: (0-15)”;BC$:BC$ = VAL (BC$): IF BC < 0 OR BC > 15 THEN 120
200 REM RESET VALUES
210 SV = TX * 16 + BK: POKE 49186,SV: REM SET BACKGROUND & TEXT
220 BV = HV * 16 + BC: POKE 49204,BV: REM SET BORDER COLOR
230 PRINT : INPUT “TRY AGAIN? (Y/N)”;IS
235 IF IS = “Y” OR IS = “y” THEN 100
290 POKE 49186,OS: POKE 49204,OB: REM RESTORE SCREEN
299 END
999 DATA BLACK,DEEP RED,DARK BLUE,PURPLE,DARK GREEN,DARK GRAY,MEDIUM BLUE,LIGHT BLUE,BROWN,ORANGE,LIGHT GRAY,PINK,GREEN,YELLOW,AQUAMARINE,WHITE

Although it may seem a little long, the basic principles are quite simple. The softswitch at location 49186 contains both the background color and the text color. Dividing by 16 separates text color value. Subtracting this value from the total value then yields the background color. Since each color range can only be in the range of 0 to 15, it is possible to pack both values into a single byte.

Location 49204 holds the value for the border color, along with some other system values. The designers of the Apple II GS probably didn’t want to waste half a byte. The array C$( ) has been set up with the words for each color value in the interest of making the program user-friendly. After inputting the new values, the new values are recombinced by multiplying the text color
Chapter 1

by 16 and then adding the background color (the reverse of the decoding process). When this value is POKEd back to location 49186, the background and text colors immediately change. The screen border byte is recalculated in a similar manner and is likewise updated with a POKE.

The program saves the original contents of both locations so things can be restored when the program is finished. If you make a mistake entering the program and suddenly find yourself with black text on a black background, or something similarly unreadable, just press RESET to restore the colors you've already chosen in the Apple II GS Control Panel.
Chapter 2

Real Machine Language Programming
Machine language programming is the process of storing the exact number values in memory that the 65816 will be able to understand to carry out a set of instructions.

In the last chapter, you saw how to CALL machine language program from Applesoft BASIC in much the same way as you would do a GOSUB to a BASIC subroutine.

To see how a BASIC program can create and then run a short ML program, enter Program 2-1.

Program 2-1. ML Programming Using BASIC POKEs

10 HOME
20 POKE 768,32
30 POKE 769,12:POKE 770,253
40 POKE 771,96
50 PRINT “PRESS A KEY...”
60 CALL 768
70 PRINT “THANKS!”
80 END

When you RUN this program, the screen should clear, and the PRESS A KEY prompt and a flashing cursor should appear on the screen. When you do press a key, THANKS! should appear and the program will end.

Functionally, this program is equivalent to the "Waiting for Keypress" demo program in Chapter 1. In that program, a call to a built-in routine at 64780 was done to wait for a keypress. How can the program above accomplish the same function by calling location 768? And what are all those POKEs for?

If you examine the program closely, you can see that four memory locations, starting at location 768, are being POKEd to hold various number values. The number series 32, 12, 253, 96 is understandable by the 65816 microprocessor in your Apple to mean "wait for a keypress."
This is a true machine language program, which is placed in memory starting at location (address) 768. Line 60 of the Applesoft BASIC program then uses a CALL statement to execute the routine. Remember that a CALL is very much like a GOSUB, except that instead of going to a line number in your BASIC program, it jumps to a memory location, expecting to find a machine language program there.

You know that every subroutine in BASIC must end with a RETURN to properly return to the main program. The same principle applies to machine language subroutines. Can you guess which number code in our machine language program is equivalent to the RETURN in an Applesoft BASIC program? If you guessed the last value, 96, you’re right.

You could look in a technical reference manual (or at the back of this book) to determine the meaning behind the other values, but there is an easier way—use the built-in Monitor that is present in every Apple II GS computer. The word Monitor is used on many microcomputers to mean a sort of mini-language that is used at a lower level than BASIC to make life easier when dealing with machine language programs and data.

The Monitor, like Applesoft BASIC, is itself a group of machine language routines. These routines start at memory location 63488 and go up to 65535. In fact, earlier, when we said that the Applesoft BASIC routines were in the range of 53248 to 65535, it wasn’t quite the whole picture. Actually, the Applesoft BASIC routines start at 53248 but end at 63487. The next byte is then where the Monitor routines actually start. As it happens, Applesoft BASIC does call the Monitor routines to do many functions like cursor handling, keyboard input, and so on. That’s why the cursor routines in Chapter 1 were in the Monitor memory range from 63488 to 65535.

Figure 2-1 is a more complete memory map.

Figure 2-1. Memory Map Showing the Location of the Monitor

<table>
<thead>
<tr>
<th>Hardware &amp; Softswitches</th>
<th>Applesoft BASIC ROM Routines</th>
<th>Monitor ROM Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (to 53247)</td>
<td>53248 (to 63487)</td>
<td>63488</td>
</tr>
<tr>
<td></td>
<td></td>
<td>65535</td>
</tr>
</tbody>
</table>

The Monitor is useful for examining any portion of the computer’s memory and has a number of other handy features as well, including a small assembler and the ability to list ML programs in an understandable way.

One of the most common uses of the Monitor is to look at the contents of a range of memory locations. To examine the part of memory that holds the
machine language program POKEd into memory by the BASIC program, you need to first change to the Monitor command mode. To enter the Monitor voluntarily,* type in:

CALL -151

and press RETURN.

The Applesoft BASIC prompt ( ] ) should change to an asterisk ( * ), which indicates you are officially in the Monitor. That is to say, Applesoft BASIC commands are no longer recognized, and instead, Monitor commands are. You can try typing HOME at this point to confirm this (your computer should just beep at you), but don’t try randomly typing anything else just yet. Now, to look at those numbers that make up our program, type in

300.303 (and press RETURN)

The screen should display

*300.303
00/0300:20 0C FD 60- .}

At this point, you may justifiably say “Wait a minute—this doesn’t look like the same numbers my BASIC program used. And I thought it started at location 768, not 300.”

You’re almost right. The numbers don’t look the same, but—believe it or not—they do represent the same values.

Hexadecimal Numbers

In explaining the numbering systems used in computers, people who write books and columns on machine language programming often talk about how we count by sets of 10 because we have 10 fingers. Ten is called the base of our numbering system because it’s the foundation for how we count: We group quantities by tens.

Regardless of why you think we count by 10’s, it is true that in a mathematical sense, the base for a counting system can be any number you want. In the computer, for various electrical and logical reasons that we won’t discuss

* As opposed to the surprise visits, usually accompanied by a beep, program crash, and a message something like:

00/0320: 00 D8 00/0320: 00 D8 00/0320: 00 D8
A=0000 X=0000 Y=0000 S=01DD D=0000 P=30
B=00 K=00 M=0C Q=80 L=1 m=1 x=1 e=1

25
Chapter 2

here, your Apple computer likes to count by 2's at the machine level of operation.

This brings us back to the Monitor. Applesoft BASIC is written using 12K of machine language routines to support the roughly 100 BASIC commands. The Monitor also has a limited space in which to create its commands, and it has to take care of things like reading the keyboard and managing the screen. Therefore, when they went to create the routines to print the numbers stored in a given memory location (or to decode numbers typed in by the user), it was much easier to deal with routines in a number system based on 16 (a power of 2) than it was 10. This number system, based on groups of 16, is called the hexadecimal number system, called hex for short.

It really isn’t necessary for you to be an expert in counting in base 16 to be able to use the hex numbers on your screen, but a brief explanation will make things a little easier.

When counting in base 10, you count from O to 9, and then use a second position (for the 1) to create the next number (10). In a sense, we’re just putting a limit on how many symbols are legal in a given digit position. When counting from 10 to 99, we again limit ourselves to the digits 0 through 9, and thus have to move over another position for the value after 99.

When the counting system for hex numbers was created, the mathematicians asked why they couldn’t count past 9 by using other nonnumerical symbols. Seeing no reason to quit at nine, they began using letters to continue counting.

When you count in hex, you start just like you did before:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9

but, when you get to 9, you begin using letters of the alphabet:
... 8, 9, A, B, C, D, E, F

Allowing characters up to the letter F lets us use just one digit position (now called a hexit) to count to what would normally be 15. In any case, when you continue, you now do just what you did to mark a group of 10 in base 10—add a 1 to the left to mark a set of 16—and keep counting:
... 8, 9, A, B, C, D, E, F, 10, 11, 12 ...

Like before, when you use up the characters on the right, you just repeat the pattern:
... 19, 1A, 1B, 1C, 1D, 1E, 1F, 20, 21, 22 ...

Remember: Don’t worry about being a math whiz at this point. All you
need to do is to believe that the symbol 1B is a legitimate way of writing the number quantity denoted as 27 in base 10.

1B (hex) = (1 × 16) + 11 = 27 (decimal)

If you count high enough in hex numbers, the symbols run out again, and you’ll have to add another number position:

... 99, 9B, 9C, 9D, 9E, 9F, A0, A1...
... F9, FA, FB, FC, FD, FE, FF, 100, 101...

Some people get nervous at this point because they’re not used to saying 1A as a number, or they’re not sure that 100 means one hundred anymore. Actually, as long as the other person knows that you’re talking hex addresses or numbers, saying one hundred, or even B-hundred (for B00) is just fine.

To avoid confusion in written text, however, a dollar sign is usually put in front of hex numbers (example: $B00), so someone can tell at a glance which numbering system you’re using.

Now, back to that list of numbers you were looking at in the Monitor.

00/0300:20 0C FD 60- .}

The screen display shows that location $300 holds the value $20, 32 decimal, our first POKE in the Applesoft BASIC program. Each value that follows is a number in hexadecimal notation that indicates the value stored at the successive memory locations ($301, $302, and so forth).

Note that no dollar sign is used in the Monitor. The Monitor only expects hex values, so no dollar sign is needed.

The Monitor also provides some extra information, to the right of the hex numbers, starting with the dash (−) character. We’ll explain that a little later, so you can ignore whatever’s printed there for the time being.

To verify that these numbers are the same values as POKEd in by the BASIC program, you need to be able to convert a hexadecimal number back to a decimal. Actually, that’s not too difficult. You just need to remember a few special tricks. The first is that the last six letters when counting in hex (A, B, C, D, E, F) stand for 10, 11, 12, 13, 14, and 15 in decimal. The second is that when you have more than one digit in a number, you multiply by either 16, 256, or 4096 for numbers in the second, third, or fourth positions, respectively.

$A = 10
$B = 11
$C = 12
$D = 13
$E = 14
$F = 15
Chapter 2

Example of converting a multidigit number:

\[
\begin{align*}
4321 &= 4 \times 4096 = 16384 \\
+ 3 \times 256 &= 758 \\
+ 2 \times 16 &= 32 \\
+ 1 \times 1 &= 1 \\
\hline
&= 17185
\end{align*}
\]

It’s actually more confusing to explain than to do. Let’s just look at a few examples. To convert the starting address, $300, you just need to know that the third number position gets multiplied by 256 to convert it to base 10: $3 \times 256 = 768. (For the time being, you can ignore the 00 in front of the / character in the address).

For 20, remember that numbers in the second position get multiplied by 16 to convert them to base 10: $2 \times 16 = 32; that checks with the first number POKEd into memory on line 20 of the program. For the second value, $0C, the C is equivalent to 12 in base 10. For the next number, $FD, multiply 15 ($F = 15) by 16; then add 13 ($D = 13). That’s (15 \times 16) + 13 = 253, which agrees with the third POKE. At the very end, 60 represents 6 \times 16 = 96, the value which instructs the 65816 to return from the machine language program.

Don’t worry about needing to convert back and forth between decimal and hex in order to program in machine language. Usually, your assembler will do the necessary math for you. The Monitor also has a built-in converter that will assist you. It’s important to have a general idea of what’s going on when the computer does the conversion for you. To see how the Monitor does the conversion, type in:

```
20=
```

and press Return. The screen will display:

```
*20=
Decimal-> 32 \{ +32 \}
```

The Monitor has translated $20 to decimal 32 for you. That’s a good feature. Try typing the other hex numbers followed by an equal sign, and see if you get the decimal values calculated by hand, above.

The Monitor can convert decimal to hex, also. Just put the equal sign at the beginning of the number instead of the end. For example, type in

```
=32
```

and press Return. The screen will display

```
* = 32
Hex-> $00000020
```

28
Try the other decimal numbers used in the BASIC program, and verify that they agree with the hex values shown when you typed in 300-303.

**Why Bother with Hex Numbers?**

You may be wondering why you even need to bother with hexadecimal notation. Couldn't the computer do all the translations automatically? It could, but it turns out that hex numbers are actually easier to work with in many cases.

For example, you probably don't remember all the decimal addresses given as the boundaries between the hardware locations, Applesoft BASIC, and the Monitor ROM routines. That's understandable, especially considering how arbitrary the values seem. To refresh your memory, Figure 2-2 is the same memory map as Figure 2-1. Trying to remember all these addresses is difficult because there's no apparent pattern.

**Figure 2-2. Memory Map in Decimal**

<table>
<thead>
<tr>
<th>Hardware &amp; Softswitches</th>
<th>Applesoft BASIC ROM Routines</th>
<th>Monitor ROM Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 49152 (to 53247)</td>
<td>53248 (to 63487)</td>
<td>63488</td>
</tr>
<tr>
<td></td>
<td>65535</td>
<td></td>
</tr>
</tbody>
</table>

Next, look at the same chart using hex numbers, Figure 2-3.

**Figure 2-3. Memory Map in Hex**

<table>
<thead>
<tr>
<th>Hardware &amp; Softswitches</th>
<th>Applesoft BASIC ROM Routines</th>
<th>Monitor ROM Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 $C000 (to $CFFFF)</td>
<td>$D000 (to $F7FF)</td>
<td>$F800</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$FFFF</td>
</tr>
</tbody>
</table>

Addresses like $2000, $4000 and even $F800 are much easier to remember than 8192, 16384, and 63488.

This may seem like a lot to absorb right away, but be patient. These concepts will be covered again many times throughout the book. It will become clear to you later. So, be patient, allow yourself some breaks in between topics, and you'll find that learning hex is almost effortless.
It’s Culture That Counts
Many people have remarked that our choice of ten as a number base is related to the fact that we have ten fingers on our hands. One can only guess how a different set of circumstances would have profoundly changed our lives. Speculating, for instance, on which two commandments would have been omitted had we only eight fingers is enough to keep you awake at night.

A living example of this arbitrary nature of number bases was recently brought to light by the discovery of a long lost tribe living in the remote jungles of South America. The tribe had been isolated from the rest of the world for at least 7000 years. An important aspect of their life was a huge population of dogs living among the people. In fact, dogs so out-numbered the people (so to speak) that the people had evolved a counting system based on the number of legs on a dog, as opposed to our decimal system. They counted in the equivalent of base four.

In counting, they would be heard to say, “one, two, three...” For the next number, they just used what seemed a natural whole unit. They wrote it as 10 and called it doggy. Continuing to count they would say, “doggy-one (11), doggy-two (12), doggy-three (13). . . .” Quite practical really, 13 = 1 dog, three legs.

This system worked quite well, and they only needed four symbols to count with (0, 1, 2, 3). When they got to the number after doggy-three, they wrote it as 20 and called it twoggy. A similar procedure was used for 30.

\[
\begin{align*}
20 &= \text{twoggy} \\
21 &= \text{twoggy-one} \\
22 &= \text{twoggy-two} \\
23 &= \text{twoggy-three} \\
30 &= \text{troggy} \\
31 &= \text{troggy-one} \\
32 &= \text{troggy-two} \\
33 &= \text{troggy-three}
\end{align*}
\]

Now, upon reaching 33, adding one more meant writing the new number with three digits.

You’re probably wondering what they called it. The digits are naturally 100. Oh, the name. Why, of course, it’s one houndred.

POKEs, PEEKs, and CALLs in the Monitor
World sociology is just amazing, isn’t it? Speaking of languages, you know how BASIC lets you change and examine memory locations using POKE and PEEK, and you now know how to run a machine language program from BASIC.
These operations are also present in the Monitor.

You've seen how to examine a range of memory locations: Type in the addresses separated by a period and press RETURN. This is equivalent to BASIC's `PEEK` command. If you want to look at a single location, just type in the single hex address. To examine location $300$, for example, just type

300

The screen should print

*300
00/0300:20-

You can also examine a set of different addresses, which are not necessarily continuous in memory by typing the addresses to examine separated by spaces. For instance, type

300 302

and you should see

*300 302
00/0300:20-
00/0302:FD-

To do the equivalent of BASIC's `POKE`, type the address followed by a colon. For example, let's use a part of memory not currently used by our program, say location $30A$.

First, see what's there by typing

30A

and press Return. Make a note of what number value you find there.

To change the value of the contents of location $30A$ from its current value, to $\text{FF}$ (or pick any number you want), type in

30A: FF

Now type

30A

The screen should print

00/030A:FF-

The value has been changed. Typing an address followed by a colon and a number value is the equivalent of BASIC's `POKE` command.

Now, what about the equivalent of a `CALL` statement? In the Monitor,
all commands are single letters, rather than complete words, and they follow the address of interest. To run your program from the Monitor, type in 300G

A blinking box should appear on the screen, waiting for you to press a key. When you do, the asterisk Monitor prompt should return. The letter G that followed the address 300 stands for Go and tells the Monitor to run a machine language program at the specified address.

**The Monitor LIST Command**

You’re probably thinking by now that if you have to deal entirely with numbers to do machine language programming, it’s going to get old very fast—and you’re right. Fortunately, few people program using solely the number values, and the Monitor has some features to do things a little easier.

To list a machine language program, the monitor has a nifty command, just like BASIC does. Type in 300L

The screen should now list 20 lines on the screen that look like the example shown in Figure 2-4.

**Figure 2-4. A Monitor Listing**

```
L is the Monitor LIST command, and it tells the Monitor to display the contents of memory as an assembly language program.

Now, since we keep tossing around both assembly language and machine language, you might well ask what’s the difference?

Once, programmers programmed their computers by entering pure numbers to tell the microprocessor what to do for a given program. (That’s probably when programming got a bad name.)

Then one day, someone asked why letter codes couldn’t represent the
Real Machine Language Programming

commands, with the computer figuring out which numbers are represented by the character strings. Others thought it a good idea, so began searching for a way to do it.

Since they were very-memory conscious at the time (in 1978, 16K of RAM for an Apple cost $300), they used short abbreviations for words, called mnemonics (pronounced neh-monics). Mnemonics are memory aids; programmers use them to remember the code word for a given operation. The term code when talking about object code and source code probably originated then, too.

Thus, at $303, you see that the Monitor has translated the $60 to RTS. This stands for Return from Subroutine, and sounds a little like the RETURN that you’d put at the end of a BASIC subroutine.

It’s much easier to remember RTS when creating a program than it is to remember $60.

A program that takes a list of mnemonic codes and then translates them into the proper series of numbers in memory is called an assembler. The listing that is typed in by the programmer is called a source listing, and it corresponds to the text in the rightmost column of the Monitor listing.

The numbers that are put in memory by the assembler are called the object code, and they correspond to the numbers in the columns that were shown in Figure 2-1. The numbers at the far left are the addresses (or memory locations) of the numbers that make up the object code and are displayed for convenience only.

When you BLOAD or BSAVE a binary file that is a machine language program, you’re loading or saving just a series of numbers that makes up the object code to a machine language program.

In the technical sense then, assembly language is the programming of the computer by typing in the mnemonic codes, which are then assembled by an assembler. Machine language programming involves putting pure numbers directly into memory without the aid of an assembler. The end result is the same though, a machine language program.

As it happens, most people use the two terms interchangeably. Few people write pure machine language programs of any size anymore. Assembly language and machine language really mean two different things, but, anymore, in general conversation the distinction is not that important.

Imagine writing programs by POKEing each value into memory.

From now on, we’ll try to be somewhat consistent, and correct, in using the terms, but this means you may see both words in the same sentence, apparently referring to the same thing. There will be a difference though. What you’ll write will be an assembly language program, using the mnemonics like
Chapter 2

JSR and RTS. What you load and run in the computer is a machine language subroutine, made up of pure numbers. Therefore, you can expect to see a passage something like: "After you've finished your assembly language program, save it to the disk. Your Applesoft BASIC program can then BLOAD and CALL the machine language subroutine as it's needed."

Using Subroutines

Now, back to our program in progress. In the listing, the first group of numbers in memory (20 0C FD) are translated as JSR FDOC. JSR is a mnemonic that stands for Jump to SubRoutine. A JSR in assembly language is very much like a GOSUB in BASIC. However, instead of jumping to a line number, a JSR goes to an address in memory. As long as the routine there ends with an RTS (like a RETURN in BASIC), control will eventually return to the calling program.

In the case of the JSR, $FD0C has a decimal equivalent of 64780. Sound familiar? When we said that this sample program was functionally equivalent to the Waiting sample, the similarity was closer than you might have guessed. All we did in this program was put the CALL 64780 in our own machine language program at location 780 ($300). See Figure 2-5.

Another incidental fact to note is that the address $FD0C is stored in memory at locations $301 and $302—in reverse order. That is, $0C comes first at $301, and $FD is stored at $302. This is similar to how the line number of an Applesoft BASIC error was stored by the ONERR routine in the demo program in Chapter 1.

It turns out that this is the standard way microprocessors store, and expect to find, address values in memory. The first stored value is sometimes called the low-order byte, and the second is the high-order byte. As a further example, for the address $2000 (the beginning of the first hi-res page), the low-order byte is 0, and the high-order byte is $20. This would be stored in memory as $0,$20.

An important tip here is that the high- and low-order bytes for any hex number can be quickly determined by just visually splitting the number: $FD0C has a high-order byte of $FD, and a low-order byte of $0C. In decimal notation you have to divide by 256 to obtain the high-order byte, and use the remainder as the low-order byte. For example, 64780 ($FD0C) divided by 256 equals 253 ($FD = high-order byte) with a remainder of 12 ($0C = low-order byte).

Finally, the program ends with a RTS, which returns control to either the Applesoft BASIC program that called it, or, if you did a 300G from the Monitor, back to the Monitor prompt. Generally speaking, all ML routines called from Applesoft BASIC should always end with RTS, so that control properly
returns to the calling program. RTS in assembly language is very much like a RETURN in Applesoft BASIC—every subroutine should end with one.

The remainder of the listing will vary from time to time and computer to computer, depending on a number of factors.

By the way, any time you want to return to Applesoft BASIC from the Monitor, just press Control-C and press Return. This will get you back to Applesoft BASIC with any program you had there still intact.

**The Simplest ML Program**

The program just presented is almost the simplest machine language program possible. The only thing simpler could be RTS by itself. However, it has helped illustrate some of the key elements to machine language programming.
Chapter 2

In this chapter you’ve learned about hex numbering, assemblers, the Monitor, Monitor commands, two assembly language commands (JSR and RTS), and more.

Another key concept is that of flow of control. The computer is always executing some program somewhere. When you write a program, you’re merely redirecting that flow of control temporarily to your own program. One program can pass control to another program, or to another part of itself.

In our final example, control passed from the Applesoft BASIC program to our own machine language program, and from there to another routine at another location, and then back again through each level to the BASIC program. Mastering the ability to direct this process, and to control what happens along the way, is the essence of machine language programming. As you progress in this book, you’ll learn how to make the computer do just about anything you want by using the proper set of instructions.
Chapter 3

The Apple IIGS Mini-Assembler
Chapter 3

The Apple II GS Mini-Assembler

As you move through this book, you’ll notice that each chapter builds on the material from previous chapters. In the last chapter, you saw how to create a machine language program by POKEing the appropriate values into memory, and then either CALLing it from BASIC, or using the Monitor GO command.

In this chapter, you’ll be introduced to your first assembler, the one that’s built right into your Apple II GS. Along the way, we’ll also take a better look at the 65816 microprocessor and get a feel for how it manipulates data in the computer.

Although there are a number of Applesoft BASIC programs available that create short subroutines with the POKE and CALL method, most programs are written using an assembler. You’ll recall that an assembler is a programmer’s tool that lets you type in easy-to-remember abbreviations for microprocessor instructions. Then the assembler program translates these abbreviations into the proper number values.

Full-featured assemblers, like word processors, are usually commercial software products that you buy as an addition to the computer itself. However, for short programs, and occasional corrections to an existing program, the Apple II GS has a built-in mini-assembler you can use without purchasing any separate software at all. Let’s see how it could be used to create the same program that our BASIC program did.

Starting and Using the Mini-Assembler
From the Applesoft BASIC prompt, type CALL - 151 to get to the Monitor. When the Monitor prompt appears, type an exclamation mark (!) and press Return. The prompt should change to an exclamation mark. This indicates that you are in the mini-assembler.

The first thing to remember is that in assembly language, the addresses of each instruction take the place of line numbers in BASIC. To start a new program, you need to first tell the computer where you want the program to begin, much the same as you would start with some line number when beginning a new BASIC program.
We want our program to start at $300. To begin, type

```
300: JSR FDOC
```

but don’t press Return yet. Now type the first instruction for the program in assembly language

```
300: JSR FDOC
```

and press Return. Be sure that the 300 is at the very beginning of the line. You don’t have to put a space between the 300: and the JSR, but you’ll need a space after the JSR. The easiest way to keep things straight is to type them in so they look the best. (You don’t need a dollar sign in front of any of the numbers since the mini-assembler assumes everything will be in hex.)

When you press Return, the text you’ve entered will be rewritten, and the screen should display

```
00/0300: 20 0C FD JSR FDOC
```

A new ! prompt will await your next command. Notice how the mini-assembler has automatically translated the JSR FDO0C into the proper machine language values and has placed them in memory for you.

If you make a mistake, and, for example, type the letter O instead of a zero, the mini-assembler will try to help out with a beep and a ^ marker under where the error occurred. Unfortunately, it doesn’t always put it in the right spot, and you’ll usually get and something like

```
^!
!300:JSR FDO0C
```

If you do get the error display, retype the line carefully.

Now you’ve successfully typed your first line of a true assembly language program. You might think that you’ll have to now know the address for the next instruction ($303), but that’s not so. The mini-assembler is keeping track of the current memory location for you. To continue the program, just type a single space and RTS, and then press Return. The space is very important here—a space must precede each instruction that you want added to the current program location. The screen should display

```
00/0300: 20 0C FD JSR FDO0C
00/0303: 60 RTS
```

That’s the whole program. To turn off the mini-assembler, press Return at the beginning of the line, and the Monitor prompt ( *) should return. Now,
to verify that the program is there, type in

300L

You should get a listing exactly like that in Chapter 2 (at least for the first two lines that make up our program). To test the program, type

300G

from the Monitor as you did before.

Now, let’s write a BASIC program to call the routine. Press Control-C and Return to go back to Applesoft BASIC. Then type NEW and enter this program:

10 HOME
20 PRINT "PRESS A KEY..."
30 CALL 768
40 PRINT "THANKS!"
50 END

After typing this in, enter RUN and press Return. The program should work exactly like the BASIC program in Chapter 2, except that you can see it didn’t put the machine language program in memory itself—that was created by you using the mini-assembler.

**Saving a machine language program.** The next question is, how can you save your work? Saving the BASIC program is easy—just save it like any other BASIC program. But what about the machine language part?

To save a binary file under ProDOS, you must know the starting address and the length (number of bytes) to save. The starting address is $300. The length is four bytes. As a tip, you can always look at the number value immediately after the RTS at the end of your program and subtract the starting address from that to determine the length ($304−$300 = 4). Thus, to save this program, you would type in

BSAVE BINARY,1,A$300,L$4

This is usually done from the immediate mode of Applesoft BASIC, but you can also do it right from the Monitor. You’ll notice we use BSAVE (not SAVE) to save a machine language program (or any block of memory for that matter).

When you want to run the program again, you’ll have to get that binary file back in memory. The whole sequence for this would be:

BLOAD BINARY,1
LOAD MYPROGRAM
RUN
Chapter 3

BLOAD is used to load the binary file back into memory. You don't need to give the starting address, since ProDOS will remember where it came from. The second command loads your BASIC program, after which RUN gets everything going. I'm using the names BINARY.I and MYPROGRAM as examples for the binary and BASIC programs, respectively, but you can use any names you'd prefer.

As a side note, it really doesn't matter what order you do the BLOAD and LOAD in, as long as both files are in memory by the time you type RUN. As a matter of fact, the best approach is to have the BASIC program load the binary file itself.

10 HOME
15 PRINT CHR$(4);"BLOAD BINARY.I"
20 PRINT "PRESS A KEY . . . "
30 CALL 768
40 PRINT "THANKS!"
50 END

In this revised program, line 15 loads the binary file into memory automatically. This is the preferred arrangement, because then you (or the ultimate user) can just type in RUN MYPROGRAM to run the program right from the disk, without having to know which files to load.

Storing Data in Memory with the 65816

So far, we've been letting someone else do all the work, namely that routine at $FD0C. All our program did was tell the microprocessor where to go after it got to us via the Applesoft BASIC program.

The first step to writing a true program is to learn how to control the memory contents of the computer. In fact, this is really all any program, no matter how complicated it is, ever does. When you type on a word processor, the program in the background just watches the keyboard as you type each key, and then it stores an appropriate value in memory. With 64K of memory just for Applesoft BASIC, we can store a lot of letters, even though we're using up one byte of memory for each character.

With that inspiration, let's see how to write a program that prints a letter on the screen. Before we start though, you'll have to learn a little bit more about the 65816.

You already know that the 65816 microprocessor works by just scanning through memory one byte at a time, and carrying out some instruction depending on what it finds at each new location. To accomplish this, and many other important tasks, the microprocessor has the equivalent of built-in memory locations, called registers that are used to keep track of things.
The Apple IIgs Mini-Assembler

The most important of these is the Program Counter. The Program Counter (sometimes abbreviated PC) is a two-byte register, whose contents point to wherever in memory the microprocessor is currently operating.

A byte associated with the Program Counter called the Program Bank Register, or PBR. Although Applesoft BASIC can only use the first 64K of memory in the computer, remember that there can be up to 16 million bytes of memory. Counting in hex, the first 64K of addresses go from $0000 to $FFFF. It takes two bytes to store the pair of FFs for the last address. To count further, starting at $010000, you need a third byte.

The 65816 can run programs and access data anywhere in the computer, so a third byte is needed to make a complete address. To make things easier to read, you can write the address $010000 like this: $01/0000. The first byte, stored in the Program Bank Register, tells us which group of 64K bytes, called a bank, we’re talking about. The next four characters, representing two bytes, give the address in that bank. An address that uses all three bytes is called a long address, as opposed to a short address that only uses two bytes to give an address in a given bank, usually bank 0.

When the Monitor lists the contents of address $300 as

00/0300: 20

it’s giving the entire address, bank $00, address $300.

To make things a little more visual, we’ll make a model of the 65816 that we’ll add to as we introduce new facts about how it works. Our model is shown in Figure 3-1.

Figure 3-1. The 65816 Microprocessor Model 1

<table>
<thead>
<tr>
<th>Prog. Bank Reg. (PBR)</th>
<th>Program</th>
<th>Counter (PC)</th>
</tr>
</thead>
</table>

Figure 3-1 shows the Program Bank Register and the Program Counter with their three bytes internal to the 65816. This means that the 65816 doesn’t need to use any memory in the computer itself to store any of its “personal” information. The contents of both registers continually change as the 65816 accesses different parts of memory in different banks. The registers are like RAM in that they lose their contents if the power is turned off.
Chapter 3

The Busy Accumulator

To make things easier, the designers also put in a number of other internal registers that the 65816 can use. The most often-used is called the Accumulator (usually abbreviated as the A register). It’s called the Accumulator because the result of each successive math operation is left in this register, and is used as the basis for each successive operation, thus letting the result accumulate.

Figure 3-2. The 65816 Microprocessor Model 2

Like the Program Counter, the Accumulator is made up of two bytes (see Figure 3-2). Sometimes programmers say the Accumulator is two bytes wide. When calling 65816 routines from Applesoft BASIC, only the first byte of the Accumulator is usually used. The second byte of the Accumulator is labeled B, to distinguish it from the first byte.

The Accumulator can be loaded with either a specific number value, or the microprocessor can be instructed to use the contents of a given memory location as the value. The assembly language code for loading the Accumulator is LDA, which stands for Load Accumulator. Although the 65816 can be instructed to load both bytes (A and B) with one instruction, the default mode when calling a routine from Applesoft BASIC is to use only the A part of the Accumulator. The number of bytes actually loaded into the Accumulator can be controlled by the programmer to be either one or two bytes, as desired.

Routines called from Applesoft BASIC, or BRUN from ProDOS 8 or ProDOS 8.SYSTEM files all default to one-byte operations and use only the A part of the Accumulator. Chapter 7 discusses how to control this in your own programs. For the time being, assume that all registers operations involve only one byte unless otherwise noted.

For example, if you wanted to load the Accumulator with the value $05, you could enter

LDA #05

as the instruction in the mini-assembler. Notice the number sign ( # ) in front of the 5. This tells the mini-assembler to use the specific value 5.
An interesting alternative is to tell the computer to look in a memory location and place whatever value is stored there in the Accumulator. That instruction looks like this:

LDA 05

This second form is used the most often; it lets you move a piece of data from one location in the computer to another.

**Using the Accumulator**

The text screen display in Applesoft BASIC (or any text-based program for that matter) is controlled by manipulating the contents of a specific part of memory, namely the range from $400 to $7FF. By putting a given value in the appropriate spot in memory, any character can be made to appear at any location on the screen.

You’ve seen how to load the Accumulator with a given value or the contents of a memory location. To store that value somewhere, you use the instruction **STA (STore Accumulator)**. In a program, it looks like this:

STA $06

This instruction stores whatever was in the Accumulator in memory location $06. This can be combined with the LDA instruction to store a given value anywhere in memory. As an example, let’s use the mini-assembler to write a program that prints the letter A in the center of the screen. If you’re not already there, go to the Monitor, and then to the mini-assembler, and enter this program:

300: LDA #C1
    STA 5BC
    RTS

When you’re finished entering the program (remember to press Return alone to exit the mini-assembler), the screen should display

*!
00/0300: A9 C1 LDA #C1
00/0302: 8D BC 05 STA 05BC
00/0305: 60 RTS

Although you can type 300G to run this program, let’s keep BASIC around a little longer, even if just to clear the screen. Press Return to exit the
mini-assembler. Type Control-C, Return to go to Applesoft BASIC, and then enter this program:

10 HOME
20 CALL 768
30 END

When you run this program, the screen should clear, and the letter A will appear in the middle of the screen.

The program works by first loading the Accumulator with the value $C1, the code value for the letter A. Memory location $5BC is the byte assigned to the middle of the screen. By putting the value $C1 there, the A is instantly displayed on the screen.

For truly inquiring minds, suffice it to say that the final magic is accomplished by electrical hardware that is constantly scanning memory from $400 to $7FF and using the contents of each memory location to control the video display. If you’re really interested in this, try to find a copy of one of Jim Sather’s books on the Apple, particularly *Understanding the Apple Ile*, listed in the bibliography of this book. Although his book was based on the Ile, much of it still applies to how the Apple IIGS behaves with an Applesoft BASIC program, and is interesting reading.

The only remaining problem is that our program still needs the Applesoft BASIC program to clear the screen. If only there were a way to clear the screen from machine language.

There is. In fact, we can call the same routine, located at $FC58 that Applesoft BASIC uses to clear the screen. Try retyping the program like this:

300: JSR FC58
      LDA #C1
      STA 5BC
      RTS

When you’re done, the screen should look like this:

*!
00/0300: 20 58 FC JSR FC58
00/0303: A9 C1 LDA #C1
00/0305: 8D BC 05 STA 05BC
00/0308: 60 RTS
*!

If you type 300G now, the screen will clear without the Applesoft BASIC program. This complete program can be saved to the disk by typing

BSAVE HOME.A,A$300,L$09

46
You've just written your own self-contained machine language program. This program can even be BRUN from the immediate mode of Applesoft BASIC.

The X and Y Registers
As long as we're talking about the various registers in the 65816, we might as well mention two more: the X and Y registers. (See Figure 3-3.)

Figure 3-3. The 65816 Microprocessor Model 3

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>X Register</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Y Register</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Prog. Bank Reg. (PBR)</td>
<td>Program</td>
<td>Counter (PC)</td>
</tr>
</tbody>
</table>

Having only one internal register in the microprocessor would make things rather limited, so they added two more, the X and Y registers, that you can use to hold data with. There are also some special tricks you can do with these registers that you can't do with the Accumulator, but more on that later.

Like the Accumulator, the X and Y registers are two bytes wide. Just as you can use LDA and STA to load and store data using the Accumulator, there are equivalents, LDX, LDY and STX and STY for the X and Y registers. Like the Accumulator, you can also use the number sign (#) in front of a number value to load either register with a specific value. The number of bytes loaded and stored defaults to 1 when calling a routine from Applesoft BASIC, but this also can be changed if desired. There is no separate name for the second bytes in the X and Y registers.

As an exercise, try rewriting our HOME.A program using the X and Y registers instead of the Accumulator. You can BSAVE them to your disk with the names HOME.X and HOME.Y. When correctly entered, the programs
Moving Data in Memory

The previous examples have shown how to load a register with a specific number value, and to then store it in memory somewhere.

In the word processor we were talking about, if all you could do was store a character in memory, editing would be pretty difficult. To write any program that does very much at all requires moving data from one place in memory to another.

The general technique behind any movement of data is to first load the Accumulator, or one of the other registers, with the contents of some memory location, and to then store it somewhere else. Here's a variation on the program just presented that first stores the character to be printed on the screen in one location, and then uses a different register to pick that value up and move it to the special screen display location:

```
00/0300: A9 C1 LOA #C1
00/0302: 85 06 STA 06
00/0304: A6 06 LOX 06
00/0306: 8E BC 05 STX 05BC
00/0308: 60 RTS
```

In this program, we first load the Accumulator with the code value for the letter A, and then temporarily store it in memory location 6. Choosing $06, by the way, is fairly arbitrary. It was chosen only because it happens to be a memory location that is never used by Applesoft BASIC, so it's a safe place, rather like the range from $300 to $3CF, where you can put machine language data without hurting anything. In fact, the bytes $06 through $09 are all available for programs you write in assembly language.

The next part of the program is the part that actually moves a byte of data from one place to another. First, it loads the X register with the value
The Apple IIgs Mini-Assembler

#$C1 that we just stored, and then moves it to location $5BC, where it becomes visible on the screen. This LDX/STX combination (or you can use LDA/STA or LDY/STY) is typical of how to move data from one place to another.

There is also a special command for storing a zero in memory: STZ, Store Zero. Without this command, you would have to have these steps in a program to put a zero in location $06:

LDA #$00
STA $06

With the STZ command, you only need

STZ $06

This takes fewer bytes, and doesn’t require you to change the contents of the Accumulator. Because zeroing memory is done quite often, this is a very useful instruction.

Moving Data Between Registers

You’ve seen how to move data between one memory location and another, but suppose you want to move a byte from one register to another? To do that you just use the transfer instructions of the microprocessor. There are six transfer instructions of immediate interest:

TAX  TXA  TAY  TYA  TXY  TYX

TAX stands for Transfer Accumulator to X, and does just that, it copies the value in the Accumulator into the X register. TXA is for Transfer X to Accumulator, and provides the transfer in the reverse direction. There is also a set for transferring between the Accumulator and the Y register (TAY, TYA), and also for transferring between the X and Y registers directly (TXY, TYX).

Here’s a list of these six transfer instructions:

TAX  Transfer Accumulator to X
TXA  Transfer X to Accumulator
TAY  Transfer Accumulator to Y
TYA  Transfer Y to Accumulator
TXY  Transfer X to Y
TYX  Transfer Y to X

As an example, here’s a variation on that last program that loads the value for the letter A into the Accumulator first, and then transfers it to the X
You can also use the B part of the Accumulator to store a byte temporarily, by using another transfer instruction, XBA. This swaps the A and B parts of the Accumulator. You could use it in a program like this:

```
00/0300: A9 C2 LOA #C2
00/0302: EB XBA
00/0303: A9 C1 LOA #C1
00/0305: 8D BC 05 STA 05BC
00/0308: EB XBA
00/0309: 8D BD 05 STA 05BD
00/030C: 60 RTS
```

In this program we first load the Accumulator with the code value for the letter B, and then use the XBA swap command to move it to the B part of the Accumulator. The value for A is then loaded and stored on the screen as before. Finally, we use XBA again to retrieve the value for the letter B, and store it on the screen right next to the letter A.

### Long Addressing

The use of JSR, LDA, and STA so far have been limited to simple two byte addresses. When a routine is called from Applesoft, is BRUN, or is part of a ProDOS 8 System file (filetype SYS), it’s assumed that the program, and any associated data is in the first bank of 64K of memory. Although occasional use may be made of the second bank by routines such as the 80-column drivers or ProDOS itself, access in these routines is done by switching banks using a softswitch, as was described in Chapter 1.

The 65816, however, does allow direct access to any part of the 16-megabyte address space ($00/0000 to $FF/FFFF). This is done by using the **long** addressing mode, in which **three** bytes are used to determine the address. The long addressing mode is signified in the mini-assembler by just typing a three byte address after the LDA or STA, including any leading zeroes as needed. For example, if your program wanted to load a byte from bank 5, location $300, and then store it in bank 6, location $1A00, the following lines could be typed in using the mini-assembler:

```
!300: LDA 050300
! STA 061A00
```
This would show on the screen as

```
00/0300: AF 00 03 05 LDA 050300
00/0304: 8F 00 1A 06 STA 06IAOO
```

Try typing the lines again using only the two-byte addresses $0300 and $1A00, and compare this with the mini-assembler output from the previous example.

```
! 300: LDA 0300
    STA 1A00
```

This will show on the screen as

```
00/0300: AD 00 03
00/0303: SD 00 IA
```

You can see that in the long address form, three bytes instead of two follow the opcodes for the LDA and STA instructions. Notice also that the opcode values are different ($AF = LDA long; $AD = LDA short).

You can also do the equivalent to a JSR to anywhere in memory, but this requires a new instruction, **JSL (Jump Subroutine Long)**. A JSR will only jump to a two-byte address in the bank that a program is currently executing in (bank 0 for Applesoft, for instance). A JSL, however, has a three-byte operand, and can be used to jump to a subroutine anywhere. For example, you can use the mini-assembler to type in this instruction to jump to a subroutine at $05/0300:

```
! 300: JSL 050300
```

The screen will display

```
00/0300: 22 00 03 05 JSL 050300
```

A routine must know whether it is being called with a JSR or a JSL, because all routines called by a JSL must end with the instruction **RTL (ReTurn from subroutine Long)**, not an RTS. If a routine does not end with the proper return instruction, the computer will not return to the proper place in memory. Thus, it's up to you, the programmer, to keep track of which routines you're calling, and how you are calling them. For the time being, we'll only be using the JSR instruction for Monitor subroutines, which all end in an RTS.
Using the Monitor to Debug a Program

Before we leave the mini-assembler, there is one other technique that is very important to learn early on, and that is how to find out what's wrong with a program that isn't working the way you want it to.

This process is called debugging a program. The term's origin goes back to the early days of computers, and consisted of very large machines that used electromechanical relays for memory. The story goes that one day a computer wasn't working properly. When some technicians went inside (ah, those were the days, when you really could get inside your computer), they found a poor moth caught in one of the relays. When they came out, and someone asked them what the problem was, one said "Oh, just a bug in the computer." Well, expressions have a way of sticking around, and pretty soon everybody was blaming everything on a bug somewhere.

The result is that programmer's use the term bug to mean any error in the programming logic (or just plain typing error) that causes the program to not work the way they intended. Debugging a program is the process of tracking down the cause of the problem, and, hopefully, correcting it. One of the easiest ways to fix a bug in a program is to describe it in the documentation, and give a good reason why it's there. Best of all is to describe the bug in a really creative way, and tell everyone it's a feature!

For your programs, though, you'll probably just want them to work the way you intended, and so a few tips on how to track down these critters is in order.

For starters, use the mini-assembler to enter this program:

```
300: JSR FC58
    LDA Cl
    STA 5BC
    RTS
```

This should list as:

```
00/0300: 20 58 FC  JSR FC58
00/0303: A5 Cl  LDA Cl
00/0305: 8D BC 05  STA 05BC
00/0308: 60  RTS
```

When you run this program, either with a CALL 768 from Applesoft BASIC, or a 300G from the Monitor, the letter o should appear in the middle of the screen, instead of the expected A. What could the problem be? Generally speaking, the program looks pretty good.

This is where debugging comes in. Usually the first step is to logically think through the part of the program that seems to be causing the problem.
The screen is clearing OK, so the JSR FC58 is probably not the problem. Since the letter o is appearing in the middle of the screen, maybe the Accumulator isn't holding what we think it is. The second line does say LDA C1, but let's make sure.

The way to check this is with a breakpoint. A breakpoint is a place in the program where you force the computer to stop executing the program. It's very much like a STOP instruction in BASIC, except that in the Apple, the break-handling function prints out some extra helpful information that you usually don't get with Applesoft BASIC's STOP command.

The number code for a break is 0, pretty easy to remember. (The mnemonic is BRK, but this is probably the only machine language command that gets typed in more often directly by hand than with an assembler.)

To put a break in our program, you just need to type 0 at the point at which you want to see what the system status is. In our case, let's replace the RTS with a BRK. To do this type

308: 00

Now, type 300G to run the program. In the wink of an eye (and a quick wink at that), the screen should clear, the letter o will be placed on the screen, you'll hear a beep, and the top of the screen will show

00/0308: 00 00 BRK 00
A=00EF X=0000 Y=0000 S=01DD D=0000 P=B0
B=00 K=00 M=0C Q=80 L=1 m=1 x=1 e=1

The beep happens whenever a break occurs. The computer then prints a display of the contents of all the registers, and the address at which the break occurred.

Our break instruction was at $308, and this checks out with the display. You can also see the contents of the A, X, and Y registers, including both bytes in each (plus a whole lot more we haven't quite gotten to).

You'll notice that the Accumulator holds the value $EF, not $C1 as we had intended. Just for fun, type in

C1

and press Return.
You should see

00/00C1: EF -o

Aha—because the number sign was missing in front of the C1 in our program, the microprocessor loaded the Accumulator with the contents of location $C1, rather than the specific value $C1. This is confirmed by examining
the contents of location $C1, which just happens to be $EF, consistent with what the break display showed.

You'll also notice that the monitor shows the letter o to the right of the display of the contents of $C1. That's the significance of these characters to the right of every memory display: If the number value shown were printed on the screen, the Monitor shows what text character would appear. This character is the character associated with the number in memory according to something called the ASCII value. ASCII character codes are explained in greater detail in Chapter 6.

Even if you suspected earlier that the missing number sign was the culprit, this example still demonstrates one of the fundamental principles of debugging—placing a break point at a critical point in the program and observing whether the registers contain what you think they should.

Debugging a program can be one of the most challenging (and frustrating) parts of programming, but sooner or later everyone has to do it. Taking the time to carefully think through your programs as you're writing them, and thinking carefully about them again when problems occur, will make the process much easier. The secret is to try to narrow down the precise instant at which the program has deviated from the way you think it should be running, and to then look at what's not right at that particular spot. From there you can usually figure out what's gone wrong.

There are commercial software programs (called debuggers) that let you step through a program one line at a time, with all the registers displayed for each step. These can be helpful sometimes, but for most of what's in this book, and in fact many of the problems you'll encounter even in large programs, the judicious use of just one or two BRK instructions can quickly solve most programming mysteries.
**Nineteen Instructions**

With the completion of this chapter you’ve learned 19 assembly language commands:

<table>
<thead>
<tr>
<th>BRK</th>
<th>JSL</th>
<th>JSR</th>
<th>LDA</th>
<th>LDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDY</td>
<td>RTL</td>
<td>RTS</td>
<td>STA</td>
<td>STX</td>
</tr>
<tr>
<td>STY</td>
<td>STZ</td>
<td>TAX</td>
<td>TAY</td>
<td>TXY</td>
</tr>
<tr>
<td>TXA</td>
<td>TYA</td>
<td>TYX</td>
<td>XBA</td>
<td></td>
</tr>
</tbody>
</table>

and a little bit about trouble-shooting a program. You’ve also written, run, and saved a self-contained assembly language program. In the next two chapters we’ll discuss how to assemble programs using two popular assemblers.

---

**Secret #2**

Turn on your Apple II GS but don’t put a disk in the drive. You should get a bouncing Apple with the message, *No Startup Device*!

Hold down the Control, Option, and Apple keys and press the N key. You should get a list of all the people that worked on the Apple II GS.
Chapter 4

Assembling a Program with Merlin 8/16
The next two chapters will provide specific information on writing assembly language programs with the two most common assemblers for the Apple IIgs, Merlin 8/16 and APW (Apple Programmer's Workshop).

There are at least two other assemblers that will create 65816 programs, namely the SC Assembler, from SC Software, and the assembler that comes with the book by William Sanders called Elementary Assembly Language: Apple IIgs. Each of these has advantages, especially Sander's price, but, as of the date this book was written, they didn't have the ability to create the special kind of files that we'll ultimately need to run some of the built-in IIgs tools like the pull-down menus and windows.

If you only have the APW, you should still read this chapter because all of the listings throughout the remainder of the book use the Merlin 16 assembler. This chapter may help you understand the differences between the two systems and aid in the few changes that need to be made to a Merlin 16 source listing to assemble it using the APW. If you only have Merlin 8/16, you may wish to skip Chapter 5.

The specific technical facts presented in this chapter, such as assembler fields, labels, and so forth are repeated in Chapter 5, so you don't need to worry about missing anything really important if you only read one of the chapters.

Both assemblers come with rather substantial manuals that explain how to use them. This book will not attempt in any way to duplicate that information, for obvious space reasons. We will, though, discuss the most basic file editing commands and how to assemble, save, and run programs using each. But it is assumed that any questions regarding specific functions in either assembler will be answered by their respective manuals.
Merlin 8/16

Merlin 8/16 is actually a combination product. When you buy it, the package has three assemblers in it. They are:

<table>
<thead>
<tr>
<th>Assembler</th>
<th>Operating System</th>
<th>System Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merlin 8</td>
<td>DOS 3.3</td>
<td>Apple IIe,IIc or IIgs</td>
</tr>
<tr>
<td>Merlin 8</td>
<td>ProDOS</td>
<td>Apple IIe,IIc or IIgs</td>
</tr>
<tr>
<td>Merlin 16</td>
<td>ProDOS</td>
<td>IIgs (or IIe with 65816 or 65802 installed)</td>
</tr>
</tbody>
</table>

Merlin 8 is intended primarily for use on Apple IIe and IIc computers, and creates code for the 6502/65C02 microprocessor, whose instructions are a subset of the 65816 instructions.

Merlin 16 requires the presence of a 65816 to work properly, and is intended to assemble programs for the Apple IIgs. When this book was written, however, it was running on any Apple IIe and IIc computers that have had a 65816 (or 65802, a IIe/IIc compatible) microprocessor added to them, such as can be done with products from Applied Engineering and Checkmate Technologies.

In the following instructions, we'll assume that you're familiar enough with ProDOS to be able to run Merlin 16 from BASIC, the Launcher, or some other program selector, and are able to successfully quit Merlin 16 and return to Applesoft BASIC. Later on, as the programs become more sophisticated, you'll be able to run them directly from the Launcher or a program selector. But, for now, a CALL from Applesoft BASIC is probably the best way. It's also easier to the Monitor for debugging if you run your programs from Applesoft BASIC.

You should also have an initialized disk, formatted under ProDOS, on which you can save the sample programs as we go along. For our examples, we'll assume that disk has the volume name /PROGRAMS, but you can use any name you wish. If you have a disk you're already using to save the programs presented earlier, you can continue to use that.

Writing a Program Using the Merlin 16

The first step is to get the Merlin 16 up and running. Use the Apple Program Launcher that comes on your Apple IIgs System Disk to run MERLIN.16, or use whatever other technique you usually use to run a program.

When run, the following title screen will appear.
Merlin 16 has 3 levels of operation: the main menu, the full screen editor, and the assembler/linker.

The screen above is the main menu. Looking over the list of commands, you can see options for loading and saving source files, object files, executing disk commands and more.

There is also a display of the current ProDOS prefix. Depending on what disk you have Merlin 16 on, this prefix will vary.

You can use the main menu D command (for Disk) to change the prefix. Try it now by inserting your own program disk (we’ll assume it’s called /PROGRAM). Press D. The command prompt near the bottom of the screen should change to

Disk command:

Type in:

PREFIX /PROGRAMS

and press Return. The disk should on for a moment, and then the Prefix indicator should change to /PROGRAMS.

You can view the disk catalog by pressing the C key (for Catalog). Merlin will prompt

%Pathname:
Press Return alone to use the Prefix shown on the screen, which is presumably /PROGRAMS (or whatever the name of your data disk is) at this point.

When you’re done looking at the catalog, press Return to view the main menu again.

To enter your first program, press F to go to the full screen editor mode of Merlin 16.

The screen will clear, and the cursor will change to an inverse I, and you’ll see a vertical bar, and the number 1 at the top of the screen.

The vertical bar is a width indicator, set for most printers, to show how far to the right you can type before the text will be too wide for the printer. This position can be adjusted, and is explained in the Merlin 8/16 manual.

The number at the upper right corner of the screen is a line number indicator, and gives you an idea of where you’re at in your listing. Try it out now by pressing the down arrow, and then the up arrow to move the cursor up and down on the screen. The line number indicator will change as you move. Use the up arrow to bring the cursor back to line 1.

A good assembler is like a word processor for programming. It should have features inserting and deleting text, moving blocks of text around, formatted printing, and so forth—features that make dealing with source files easier. In the previous chapters, we called the list of LDAs, the source listing, even though they weren’t saved separately in their own file. With a real assembler, they are. The lines you type in will eventually be saved as a text file. In fact, you could even use another word processor to edit the files if you liked, although you’ll probably find the Merlin editor more convenient.

The Merlin 16 assembler is also different from the mini-assembler in that it lets you add text as comments to your programs, very much like a REM statement in Applesoft.

Entering your program. To start off our program, let’s begin by putting a title at the beginning. With the cursor on line 1, hold down the Open Apple key, and then press 8. A row of asterisks should appear on line 1. In most assemblers, any line that begins with an asterisk is considered a comment line, and anything after the first asterisk is ignored by the assembler when creating the actual program. In this case, we’ll use asterisks to create a title box at the beginning of the program listing.

If you haven’t done so already, press Return to move the cursor to line 2 of the listing. Now hold down the Open Apple key and press 9, but don’t press Return yet. This time the line will be bracketed by asterisks, and the middle will be left blank. This is where we’ll type the title. While we’re on the subject, from now on, we’ll just indicate Open Apple commands in the form Open Apple-8, and assume you’ll remember to hold down the Open Apple key while
you press 8. If you press the 8 key first, you'll just get the character 8 on the screen.

Use the right-arrow key to move the cursor over a few spaces, then type

1st Merlin Program

You'll notice that as you type, the asterisk at the end of the line automatically moves ahead. This action is called the insert mode and means that the editor automatically inserts the characters in the line as you type them. To avoid this, first use the Delete key to erase what you've typed and move the cursor back to the third character position. Then press Control-I.

This toggles the insert mode, and you'll see the cursor change to a solid block. Toggling means that the mode will flip on and off each time you press Control-I. When the insert mode is off, the cursor just types over what is already on the screen, instead of creating new space as you type. This is called the overstrike mode and, for now, this is what you want.

You can always tell what mode the editor is in by the appearance of the cursor: I indicates insert mode; a solid block means the overstrike mode. Now again type

1st Merlin Program

If you make a mistake typing, use the left-arrow key to back up. Delete will always remove spaces, regardless of the Insert/Overstrike status.

When everything looks OK, press Return to move the cursor to the next line. Now type Open Apple–9 again to create another line in the box, and fill it in with By and your name. It's a good idea to make up a title box for every program you write, so that when you look at the program later, you'll remember what it does. You'll usually want to include even more information, such as the date, any commands the program may recognize, and any other useful information.

This process is called documenting a program, and it is a very important step. While you're writing a program, you may remember what each part does, but even by the next morning you'll be much happier if you included lots of text information explaining what each part of the program does.

When you're done filling in the third line, press Return. You should now be on line 4. Press Open Apple–8 to create another solid line for the bottom of the box. If you've done everything correctly, your program should look something like this:

*****************************************************************
* 1st Merlin 16 Program    *
* By <your name>          *
*****************************************************************
Chapter 4

While you’re editing, there are a number of editor commands that can make your life easier. The four directional arrow keys will move the cursor around. You can also press Control-B to move to the beginning of a given line, and Control-N to move to the end.

You can also use Control-i to toggle the insert mode, although using this with the arrow key and the delete key—and keeping everything lined up—takes a little practice. Even though this listing is rather short, you can also use Open Apple-B and Open Apple-N to move to the very beginning and the end of the entire source listing itself.

When you’re ready, press Return or use the down arrow to move to line 6. This will leave a blank line below the title box. The assembler doesn’t care about all this, but it makes the program look better.

Before you enter the first line, let’s review how you entered a line in the mini-assembler. There, you typed in

300: JSR FC58

There are three pieces of information here. At the left is the address to put the instruction. In the second position is the opcode for the particular instruction you want, in this case JSR. In the third position is the address the JSR needs to go to.

In a real assembler, these positions are also used, and are called fields. You usually separate them by spaces, just as you did in the mini-assembler.

The first field is called the label field. In a true assembler, you don’t have to worry about actual addresses at all. The Merlin 16 assembler starts assuming a default address of $8000 (32768 decimal) for every program. This value can be changed if you like, but more on that later.

Adding labels. Since the assembler is going to keep track of actual addresses for us, the first field can be given a name to represent the address of that instruction, and the assembler will make sure that any references to this name are eventually translated into the proper address when the program is assembled.

There is no analogy in BASIC, since in Applesoft BASIC you can’t say “line 100 = PRINT MENU”, and then later on use the command GOSUB MENU. In an assembler, on the other hand, the line numbers are only of incidental interest, and all JSRs and references to addresses can use a label. This makes it much easier for the programmer, and in itself is a good reason to use a real assembler instead of the mini-assembler, which has no provision for labels.

Let’s call the first line of our program BEGIN. Type BEGIN, and then press the space bar once (don’t press Return yet). Even though you only typed
one space, the cursor will automatically jump to the second field position on
the line. The second field is called the opcode field and is where you put a given
65816 instruction, such as JSR. Merlin automatically uses the space character to
separate fields and to move the cursor to a given column so that the final list­
ing will look orderly.

In the second field, type JSR and press the space bar again. The cursor
then moves to the third field, which is called the operand field. An operand is
the information needed by an opcode. For example, a JSR by itself doesn’t tell
the assembler where the JSR was destined. Type $FC58 here for the operand,
and press the space bar again. In the Merlin 16 assembler, you have to use the
dollar sign ( $ ) in front of all hex numbers. If you don’t, Merlin will treat it as
a decimal number.

The fourth, and last, field is called the comment field. This is where you
can add text to explain what a specific line does. It’s customary (some assem­
blers, though not Merlin 16, insist on it) to begin a comment with a semicolon.
Type ; CLEAR SCREEN and press Return.

If all went well, your program should look like this:

```
******************************************
* 1st Merlin 16 Program                  *
* By <your name>                        *
******************************************
```

```
BEGIN JSR $FC58 ; CLEAR SCREEN
```

For the second line, we won’t use a label. Actually, a label is always op­
tional unless it is directly referenced somewhere else in the program, for ex­
ample by a JSR somewhere. In that light, we didn’t really need the label at the
beginning of the first line of the program, but we’ll leave it there since it
doesn’t hurt anything.

Since we don’t need a label for the second line, press the space bar once
to move the cursor to the opcode field. Now type LDA, a space, and #$C1.
Then press the space bar once more, and type

```
; LETTER "A"
```

and press Return. For the next line, space to the second field and type in
```
STA $5BC ; SCREEN LOCATION
```

and press Return. Finally, finish the program with the label DONE, and RTS.
You don’t need an operand for RTS, and you can omit the comment, too.
Chapter 4

Your program should now look like Program 4-1.

Program 4-1. First Program

*******************************
* 1st Merlin 16 Program *
* By <your name>          *
*******************************

BEGIN JSR $FC58 ; CLEAR SCREEN
LDA #$Cl ; LETTER “A”
STA $5BC ; SCREEN LOCATION
DONE RTS

Assembling with Merlin 16

To assemble the program, press Open Apple–O (for Open dialog box). This will open a small dialog box on the Merlin screen in which you can enter direct Merlin system commands.

The actual assembly is started by typing

ASM

When you press Return, text will suddenly appear on the screen, and things should look like Program 4-2.

Program 4-2. Assembly of First Program

1  ********************************
2  * 1st Merlin 16 Program   *
3  * By <your name>          *
4  ********************************
5
008000: 20 58 FC 6 BEGIN JSR $FC58 ; CLEAR SCREEN
008003: A9 C1 7 LDA #$C1 ; LETTER “A”
008005: 8D BC 05 8 STA $5BC ; SCREEN LOCATION
008008: 60 9 DONE RTS

--End Merlin-16 assembly, 9 bytes, Errors: 0

Symbol table - alphabetical order:

? BEGIN =$8000 ? DONE =$8008

Symbol table - numerical order:

? BEGIN =$8000 ? DONE =$8008
Assembling A Program with Merlin 8/16

There are a number of things to notice. The first is that Merlin has created a listing at the far left of the addresses where each instruction will ultimately be put in memory. I mentioned that Merlin assumed a starting location of $8000, and you can see each line indicates the memory address followed by the actual machine language for each instruction of your program. If you want the listing to go by a little more slowly, you can press the space bar immediately after typing ASM, and Merlin will step through the lines one at a time, keypress by keypress. Pressing Return resumes the normal speed listing.

At the end of the listing is something called the symbol table. This prints a list of all the labels used in the program, such as BEGIN and DONE. The questions marks are printed by Merlin; they indicate that these labels are not called by name anywhere in the program. You can ignore them if you wish.

To save the program to your disk, you need to go back to the main menu. Type Q (for Quit) now in the immediate mode of the editor. The main menu should reappear.

Merlin-16 3.00+
Copyright (c) 1987 By Glen Bredon
14-NOV-87 8:55

C :Catalog
L :Load source
S :Save source
A :Append file
D :Disk command
E :Editor, command mode
O :Save object code
@ :Set date
Q :Quit

Source: A$0901,L$0000
Object: A$8000,L$0009,BIN

Prefix: /MERLIN/
%

You can see that a new indicator has been added to the screen in the lower right corner, below the word Source. The heading Object: shows that there is now an object file in memory, and that it has a length of nine bytes. If this indicator does not show up, it's because either an error occurred during the
assembly or the file was assembled directly onto the disk, a problem that will be discussed later.

There are now two files that need to be saved, the source file and the object file. To save the source file, press the S key (for save Source file). The prompt will change to

%Ssave:

Type in the name MERLIN.1 and press Return. The disk will come on and the source file will be saved to the disk. Then the prompt returns, you can view the disk catalog by pressing the C key (for Catalog).

In the disk catalog you should see the file MERLIN.1.S. The .S suffix is automatically added by Merlin when a source file is saved, or loaded, so don’t worry about ever having to type this yourself.

To save the object file, press the O key. The prompt

%Object:MERLIN.1

will appear. Merlin is offering the same name as a default. Press the Y key (for Yes) to accept this default. Merlin will use this name without the .S suffix, so you don’t have to worry about overwriting your source file.

If you catalog the disk again, you’ll see your source file saved as a text (TXT) file and the object file, MERLIN.1, saved as a binary (BIN) file.

To run the assembled file, you’ll have to quit Merlin and go to Applesoft BASIC, or a to a program selector other than Apple’s Program Launcher, that can run binary files. Applesoft BASIC is the recommended method for now.

Press the Q key to quit Merlin. It will ask you to confirm with the prompt

%Quit

Press Y to finish the quit and go to Applesoft BASIC. When you’re in Applesoft you’ll probably have to reset the prefix to your /PROGRAMS disk before you can run MERLIN.1.

Assuming the prefix is correct, and that the /PROGRAMS disk is in a drive, you can now type either

BRUN MERLIN.1

or

-MERLIN.1

to run the program. You can also type

BLOAD MERLIN.1
CALL 32768
This last bit may surprise you. Remember, though, that Merlin 16 used a default address of $8000 (32768) to assemble the program. That is also where the binary file will automatically load MERLIN.1 when you do a BLOAD.

There are two ways of moving the code to $300 (768). The first is to specify the address as part of the BLOAD command

BLOAD MERLIN.1,A$300
CALL 768

The other way to move the code is to tell the assembler where you want the code to ultimately run. To do that, we'll need to go back to Merlin 16. Type BYE in the immediate mode to quit BASIC back to a program launcher, or set the ProDOS prefix back to the Merlin program disk and type in

-MERLIN.16

**Assembler Directives: ORG**

When Merlin is back up and running, load your source file MERLIN.1 from the main menu. Remember that you don't need to include the .S suffix with the name.

When the source file is loaded, enter the editor/assembler with the E command. Then press E to edit the file. The full-screen editor should start up, with the cursor on the first line, in the upper left corner of the screen.

Move the cursor down, using the down arrow, to line 5 (the blank line) and press Return. A new blank line should be inserted. When the editor is in the insert mode, the cursor will be shown as an inverse I, and pressing Return will insert a new blank line.

Press the space bar once to go to the opcode field, and type ORG. Then press the space bar once, and type $300 and press Return. The listing is shown in Program 4-3.

Program 4-3. First Program

```
******************************************
* 1st Merlin 16 Program      *
* By <your name>            *
******************************************

ORG $300
BEGIN JSR $FC58 ; CLEAR SCREEN
LDA #$C1 ; LETTER "A"
STA $5BC ; SCREEN LOCATION
DONE RTS
```

69
ORG is not a 65816 opcode. Instead, it's an instruction specifically for the assembler. ORG stands for Origin, and tells the assembler where the machine language program will ultimately run. Instructions like ORG are called assembler directives, because they direct the assembler to take a certain action.

After you've added the new line with ORG, press Open Apple-Escape, and then type ASM to assemble the program. If there are no errors, press Q to go back to the main menu.

When you press S to save the modified source file, Merlin will prompt you with the name MERLIN.1. Merlin remembers the name of the file last loaded. To keep this name, just press Y (for Yes). After the source file is saved, press O to save the object file, and again accept the default name.

When you've saved the new file, quit Merlin, go back to BASIC, and type:

BLOAD MERLIN.1
CALL 768

This time, the file is located at $300. That's because you told Merlin where you wanted the file to load with the ORG directive.

More About Labels
Now go back to Merlin again, and load MERLIN.1. The listing should look like Program 4-2.

Looking at the listing, you can see addresses like $FC58 and $5BC, and the code value for the letter A, $C1. As a program gets larger, and the listing longer, it can start to get rather confusing trying to remember many different number values. To make programming more convenient, most assemblers allow you to give routines a name and remember the address for you.

Start by putting the cursor on line 7, press Return once, and type the following:

HOME EQU $FC58
SCREEN EQU $5BC

Now move the cursor to this line:

BEGIN JSR $FC58 ; CLEAR SCREEN

Move the cursor to the end of the address $FC58, and press the Delete key until it's completely erased. Type HOME, but don't press Return. Now use the down-arrow key to move down two lines, and do the same thing to replace the address $5BC with SCREEN.

Your program should now look like Program 4-4.
Assembling A Program with *Merlin 8/16*

Program 4-4. First Program with EQU

***********************************************************************
* 1st Merlin 16 Program      *
* By <your name>            *
***********************************************************************

ORG $300
HOME EQU $FC58
SCREEN EQU $5BC
BEGIN JSR HOME ; CLEAR SCREEN
LDA #$C1 ; LETTER "A"
STA SCREEN ; SCREEN LOCATION
DONE RTS

Use Open Apple-O and the command ASM again to assemble this listing. The assembly output is shown in Program 4-5.

Program 4-5. Assembly of First Program With EQU

1  ***********************************************************************
2 * 1st Merlin 16 Program      *
3 * By <your name>            *
4 ***********************************************************************
5
6 ORG $300
7 =FC58 8 HOME EQU $FC58  
=05BC 9 SCREEN EQU $5BC

000300: 20 58 FC 11 BEGIN JSR HOME
000303: A9 C1 12 LDA #$C1
000305: 8D BC 05 13 STA SCREEN
000308: 60 14 DONE RTS

--End Merlin-16 assembly, 9 bytes, Errors: 0

Symbol table - alphabetical order:
  ? BEGIN =$0300 ? DONE =$0308 HOME =$FC58 SCREEN =$05BC

Symbol table - numerical order:
  ? BEGIN =$0300 ? DONE =$0308 SCREEN =$05BC HOME =$FC58

You’ll notice that this time the assembler translates the labels HOME and SCREEN into their assigned addresses. Labels are always associated with some value in the source listing. If you look at the symbol table at the end of the listing, you’ll also notice that there are no question marks next to the labels
HOME and SCREEN, like there are next to BEGIN and DONE. The assembler is telling us that the labels HOME and SCREEN are actually referenced by the program, whereas BEGIN and DONE are not. This is intended as a debugging aid, to point out any labels you defined, but might have forgotten to use.

Whenever the assembler encounters a label, either at the beginning of a line, or following the EQU directive, it assigns a value, usually an address, to it. For EQU statements, it uses whatever value is on the line. If the label is at the beginning of the line, it uses its own internal address counter for where it's at in the program.

For instance, since BEGIN is the first statement in the program, the assembler gives BEGIN a value of $300. As it continues to assemble each line, it keeps track of where it is. By the time it gets to DONE, the address counter is up to $308, and that's what it assigns to DONE. Thus, if you want some other part of the program to do a JSR, for example, to BEGIN or DONE, you just use the line

JSR DONE

or

JSR BEGIN

In this program, such a JSR wouldn't make much sense, just as a GOSUB to the first or last line in a BASIC program wouldn't make sense; but, as your programs get larger, you'll want to go to subroutines within the program. In assembly language, you use the labels, not line numbers, to tell the JSR where you want it to go.

Labels can be used for more than addresses. We could also assign a label to the value for the letter A as shown in Program 4-6.

Program 4-6. Using EQU to Define a Character

*-------------------------------*
* 1st Merlin 16 Program        *
* By <your name>               *
*-------------------------------*

ORG $300

HOME EQU $FC58
SCREEN EQU $5BC
LETTERA EQU $C1

BEGIN JSR HOME ; CLEAR SCREEN
LDA #LETTERA ; LETTER "A"
STA SCREEN ; SCREEN LOCATION

DONE RTS
Using EQU for Constants

However, there is an easier way. Enter the full-screen editor, and use the De­
lete key to remove just the characters $C1 from line 12. Replace it with the
characters “A” (quotes included) like this:

    LDA #"A"  ; LETTER "A"

Now quit the editor and try another assembly. This time you’ll see the
assembler has correctly used the value $C1 for the letter A. That’s because the
letter codes have been standardized, and the assembler already knows all the
codes for all the letters.

You should also go back to BASIC, after saving the new source and ob­
ject files, and run the new assembly, just to prove to yourself that the new list­
ing works (and to make sure you haven’t made any errors along the way).

There’s nothing wrong with assigning a constant (as opposed to an ad­
dress) with an EQU statement, it just takes up a little more room in your listing
and the computer’s memory. This technique is used when you want to use a
number value throughout a listing, but want to keep your options open on be­
ing able to change your mind later.

For instance, suppose you’re testing a program that goes through a loop
200 times, and that the number 200 is used many times in the program. During
testing, it might be nice to just use the number 5 for a shorter loop, instead.
Using the label

COUNT EQU 5

makes it easy to change later, without having to search through the listing for
all the number 5’s on every line.

There’s one other common error you can avoid. Sometimes programmers
will define a constant this way:

COUNT EQU #$5

and then, later in the listing, this way:

    LDA COUNT

They think that the pound sign is included in the label COUNT. It isn’t. The
assembler evaluates each operand as a numeric expression, and then stores the
result as a the number value, not the text of the line that generated the result.
The instruction assembled as though the line read

    LDA $5
When the program runs, the accumulator is loaded with the *contents of location $5*, not the actual value 5. Because there is no error generated during the assembly, and because the value loaded into the accumulator can be almost anything (and usually is), this kind of mistake can be very difficult to track down.

Therefore, be sure you remember to always put the pound sign in front of any labels that you want used as an absolute value.

```
COUNT EQU $5
LDA #COUNT
```

As your programs get larger, so will the listing of all the labels at the end of the assembly. If you would like *Merlin* to omit the label list, you can add the instruction LST OFF to the end of your listing. LST is a *Merlin* directive that tells the assembler not to list the lines following that instruction to the screen or printer. With LST OFF at the end, your listing would look like this:

```
STA SCREEN ; SCREEN LOCATION
DONE RTS
LST OFF ; SUPPRESS SYMBOL TABLE
```

**Assembling Directly to Disk**

There is also one other assembly option in the *Merlin 16* assembler that should be mentioned. In the previous examples, we assembled each file and then went to the main menu to save both the source and object files. In many assemblers, there is also the option to write the object file to disk as it’s being created. This leaves more room in memory for the source file, although it does slow down the assembly because of the disk access.

To try this out, insert this in the listing after the ORG directive, but before the actual program:

```
DSK MERLIN.I
```

*DSK* (for disk) is a special *Merlin 16* directive to tell the assembler to save the object file as it’s being created. No quotation marks are needed around the name. Because we are creating the object file, you also don’t want to put an .S suffix; that is reserved for the source file. The amended listing is shown is Program 4-7.

Assemble this file. During the assembly, the disk should come on as the file is automatically saved to disk. When you quit the editor/assembler to go to the main menu, you’ll also notice there is no Object: indicator. This is because the object file is not created in memory, but is written directly to disk.

If you make changes to a source file, you’ll still have to save it with the S command. *Only the object file is automatically written to disk; the source file is not.*
DSK can also be used with another Merlin 16 command, TYP (for file type) to create any legal ProDOS file type as an alternative to the usual binary (BIN) file type. This won’t be needed right away, but it will come in handy later on.

Program 4-7. Using the Merlin DSK Directive

```
* 1st Merlin 16 Program
* By <your name>

ORG $300
DSK MERLIN.1 ; CREATE OBJECT FILE ON DISK
HOME EQU $FC58
SCREEN EQU $5BC
LETTER EQU $C1
BEGIN JSR HOME ; CLEAR SCREEN
LDA #LETTER ; LETTER “A”
STA SCREEN ; SCREEN LOCATION
DONE RTS
```

Starting a New Program

Whenever you want to erase the current source listing (be sure to save it to disk first), just press Open Apple-0 to open the dialog box, and type NEW. If you want to load a new file directly from the main menu, you don’t have to type NEW first.

Long Addresses in Merlin 16

In the Merlin 16 assembler, there is a special opcode syntax for the long addressing modes of the LDA and STA instructions. To indicate you want to use a long address, use the greater than symbol (>) in front of the label to be assembled in the long address form. The long form of a JSR (JSL) requires no additional syntax. The reason Merlin requires a different opcode for a long LDA or STA is to resolve the potentially ambiguous case of LDA LABEL where LABEL equals a value less than $01/0000. For example, the statement LDA $300 could be assembled as either a short form LDA $0300 ($AD 00 03), or as the long form LDA $000300 ($AF 00 03 00).

Program 4-8 is a program segment that illustrates some of the different instructions possible.
Program 4-8. Long Addressing Example

This listing shows how various label addresses are resolved by different opcodes, and it also shows the syntax of the JSL instruction. In the first case (lines 10, 11), LABEL1 and LABEL2 are truncated to two bytes, even though both evaluate to three bytes, and the short address mode of LDA and STA are used. In the second case (lines 13, 14), LABEL1 and LABEL2 are assembled using the long address form. Lines 16, 17 show the alternate possible outputs of assembling the address $300, depending on which form of LDA is used. Line 19 shows the assembly of the JSL instruction.

There is also another syntax available for telling Merlin to use the long form: add \textit{L} to the opcode, as in LDAL LABEL or STLAL LABEL. Lines 21 and 23 show this.

\textbf{Merlin in Review}

If you’ve worked through the examples in this chapter, you should be able to enter, edit, assemble, and save both a source listing and the object file using the Merlin 16 assembler. To learn more about editing, such as how to cut, copy and paste sections, or how to use search and replace, you should read the manual that came with Merlin.
You should have an understanding of how to use labels in a source program to represent a given number value or address, how to use ORG and EQU to control the values that labels are assigned, and where the program will load into memory from disk.

*Merlin* directives introduced in this chapter:

EQU ORG LDAL STAL DSK TYP LST
Chapter 5

Assembling a Program with APW
The APW (Apple Programmer's Workshop) assembler is the second assembler we'll discuss in this book. The information presented about labels and assembler directives is roughly equivalent to that presented in the last chapter on the Merlin 16.

If do not have the APW, you may wish to skip this chapter, so as to not add to the burden of remembering what's already been discussed that is specific to the Merlin 16. In general, the Merlin 16 is a much simpler program to use, and this chapter may seem more difficult. This is because the APW was not designed with the novice programmer in mind. Is is, though, possible to master.

Before beginning this section, you should spend at least some time reviewing the documentation that came with the APW regarding installation and basic file handling procedures.

A word of warning: The APW was not designed for the beginner. The discussion in this chapter has been made as clear as possible, but there is no getting around the fact that even a minimal assembly with the APW seems like a lot of work at first. If you do have difficulties, don't be discouraged about programming altogether. The APW is covered in this book because it is likely that many people will have that assembler when they read this. However, it is not necessarily the best choice for someone who is not a professional programmer.

**Apple Programmer's Workshop**

The Apple Programmer's Workshop (APW) system is made up of many subunits, which create a program development environment. The APW can only be run on an Apple IIgs using the ProDOS 16 operating system.

When the APW is run, what is actually in memory is a shell program.
that has very few commands of its own. It knows how to load and run the
other modules as you need them. The general theory of operation is as follows:

Editor Edit and save FILE (SRC) source file
Assembler Assembles and saves FILE.ROOT object module
Linker Processes FILE.ROOT into loader file. run program
  Execute loader file
  Change file type to BIN, SYS, S16 or other type recog-
  nizable by Program Launcher or SYSTEM boot process.

First, the editor is used to create the original source listing from which
the machine language program will ultimately be generated. When the source
file is ready, it is saved to disk. The APW shell is then instructed to assemble
the program. This generates a file called the object module that is an interme-
diate file to the final usable program. The assembler automatically adds the suffix
.ROOT to a name you specify as what you want for the final usable file.

Next, the shell is instructed to process the intermediate file, through a
process called linking. It processes it into a file, called a loader file, that is exe-
cutable from within the APW shell. This program has the file type EXE (Exe-
cutable), and can only be run from within the APW system.

To create the final file that can be run from the Program Launcher or
BASIC, the shell is given a command to tell it exactly what kind of final file
type the program should have. This is so the programmer can create any of the
legal ProDOS file types, such as ProDOS 16 system files (S16), ProDOS 8 sys-
tem files (SYS), desk accessories (CDA), or others. Of course, the assembler
can’t tell if the program you have written is appropriate to the file type you
specify. It’s up to you to stay out of trouble.

The assembly and link processes can be done in one step, and there is
also a provision in the APW assembler for creating a command file that will do
the entire process automatically by just executing the command file.

In the following instructions, we’ll assume that you’re familiar enough
with ProDOS to be able to run APW from the Launcher, or some other program
selector, and are able to successfully quit APW and return to Applesoft BASIC.
Later on, as the programs become more sophisticated, you’ll be able to run
them directly from the Launcher or a program selector. For now, a CALL from
Applesoft BASIC is probably the best way. It is also easier to get into the Moni-
tor for debugging if you run your programs from Applesoft.

You should also have an initialized disk, formatted under ProDOS, on
which you can save the sample programs as we go along. For our examples,
we’ll assume that disk has the volume name /PROGRAMS, but you can use
any name you wish. If you have a disk you’re already using to save the pro-
grams presented earlier, you can continue to use that.
Assembling a Program with APW

Writing a Program Using the APW

The first step is to get the APW up and running. Use the Apple Program Launcher that comes on your Apple IIGS System Disk to run the APW, or use whatever other technique you have decided on to run the APW.

When run, the following title screen will appear:

```
Apple IIGS Programmer's Workshop V2.0
Copyright Byte Works, Inc. 1980-1986
Copyright Apple Computer, Inc. 1986
All Rights Reserved
```

The screen you see is the main level of the APW shell. The pound sign (#) is the APW shell prompt. There is no list of commands on the screen, but you can enter HELP to print a list of available help files. Type HELP now and press Return to see how this works. If you get a ProDOS: File not found error, it means there are no help files on your particular APW disk. See the APW manual for details on setting up help files.

If you do have help files on your disk, you can also ask for help on a specific item by typing the word HELP followed by a space and one of the subject names as printed in the HELP list. For example, if you wanted information about printing files, you could type in

```
HELP PRINTER
```

If you’ve prepared your own data disk, now would be a good time to set the prefix to tell the APW about it. To set the prefix, just type

```
PREFIX /PROGRAMS
```

and press Return. The disk should come on for a moment, and the prefix should now be set to /PROGRAMS. To look at your disk, and make sure that the prefix is properly set, type

```
CATALOG
```

and press Return. You should get the directory listing for your disk. Don’t worry that this might cause the APW to forget where its help files are located—it still remembers where its own files are.

To enter your first program, you’ll need to go to the editor module of the APW. The editor requires any file it opens have a name, so you’ll have to give it a name to begin. Type

```
EDIT APW.1.S
```

The .S suffix is not required for APW files, and the APW doesn’t do any
automatic management of the filenames, but adding the .S makes it easier to keep your files straight.

After a moment, the screen will change, and you’ll be in the editor. At the bottom of the screen is a status line that tells you the line and column position of the cursor, how much memory is left, and the name of the current source file, which should read APW.1.S.

Try it out now by using the directional arrow keys to move the cursor around on the screen. The line and column number indicators will change as you move. Use the up-arrow key to bring the cursor back to line 1, column 1.

A good assembler is like a word processor for programming. It should have features like inserting and deleting text, moving blocks of text around, formatted printing, and so forth.

In the previous chapters, we called the list of instructions such as LDA, the source listing, even though they weren’t saved separately in their own file. With a real assembler, they are. The lines you type in will eventually be saved as a text file. In fact, you can even use another word processor to edit the files if you like, although you’ll probably find the APW editor more convenient.

In contrast to the mini-assembler, this assembler also lets you add text as comments to your programs, very much like a REM statement in Applesoft.

To start off our program, let’s begin by putting a title at the beginning. With the cursor on line 1, type an asterisk ( * ) and hold down until it starts repeating. Fill the line to column 30 and then stop. If you go too far, just press the Delete key to back up and erase extra characters.

In most assemblers, any line that begins with an asterisk is considered a comment line, and anything after the first asterisk is ignored by the assembler when creating the actual program. In this case, we’ll use asterisks to create a title box at the beginning of the program listing.

If you haven’t done so already, press Return to move the cursor to line 2 of the listing. Start this line with an asterisk also, space over a few spaces, then type

1st APW Program

Finally, use the space bar or right arrow until the cursor is again in column 30, type an asterisk, and then press Return.

Type another line just like the second, only this time fill it in with By and your name. It’s a good idea to make up a title box for every program you write, so that when you look at the program later, you’ll remember what it does. You’ll usually want to include even more information, such as the date, any commands the program may recognize, and any other information that may be useful.
Assembling a Program with APW

This process is called documenting a program, and is a very important part. While you’re writing a program, you may remember what each part does, but even by the next morning you’ll be much happier if you include lots of text information explaining what each part of the program does.

When you’re done filling in the third line, press Return. You should now be on line 4. Now type another line of 30 asterisks to finish the bottom of the box. If you’ve done everything correctly, your program should look something like this:

```
*****************************************************************************
* 1st APW Program         *
* By <your name>           *
*****************************************************************************
```

While you’re editing, there are a number of editor commands that will make life easier. The four directional arrow keys will move the cursor around. You can also press Open Apple← (or Open Apple-, ) to move to the beginning of a given line, and Open Apple→ (or Open Apple-. ) to move to the end.

Open Apple–E controls the insert mode. When the editor is in insert mode, it automatically inserts the characters in the line as you type them.

Open Apple–E toggles the insert mode, and you’ll see the words at the bottom of the screen change from EDIT to EDIT INSERT. Toggling means that the mode will flip on and off each time you press Open Apple–E. When the insert mode is off, the cursor just types over what is already on the screen, instead of creating new space as you type. This is called the overstrike mode. If you make a mistake typing, use the left-arrow key or the Delete key to back up. Delete will always remove spaces, regardless of the Insert/Overstrike status.

Don’t worry if it seems a little awkward at first. With a little practice, it will become much easier. Although the listing is rather short, you can also use Open Apple–1 and Open Apple–9 to move to the very beginning and the end of the entire source listing itself. You might want to try out these commands now to get a feel for things before we get to entering the actual program.

When you’re ready, press Return, or use the down arrow to move to line 6. This will leave a blank line below the title box. The assembler doesn’t care about all this, but it makes the program look better.

Before you enter the first line, let’s review how you entered a line in the mini-assembler. You typed in

300: JSR FC58
Chapter 5

There are three pieces of information here. At the left is the address to put the instruction. In the second position is the opcode for the particular instruction you want, in this case JSR. In the third position is the address the JSR needs to go to.

In a real assembler, these positions are also used, and are called fields. You usually separate them by spaces, just as you did in the mini-assembler.

The first field is called the label field. In a true assembler you almost don't have to worry about actual addresses at all.

Since the assembler is going to keep track of actual addresses for us, the first field can be given a name to represent the address of that instruction, and the assembler will make sure that any references to this name are eventually translated into the proper address when the program is assembled.

There is no analogy in BASIC, since in Applesoft BASIC you can't say line 100 = PRINT MENU, and then later on use the command GOSUB MENU. In an assembler, on the other hand, the line numbers are only of incidental interest, and all JSRs and references to addresses can use a label. This makes it much easier for the programmer, and in itself is a good reason to use a real assembler instead of the mini-assembler, which has no provision for labels.

Let's call the first line of our program BEGIN. Type BEGIN now, and then press the TAB key once (don't press Return yet). The cursor will automatically jump to the second field position on the line. The second field is called the opcode field, and is where you put a given 65816 instruction, such as JSR. APW uses any number of space characters to separate fields. The TAB key moves the cursor to a given column so that the final listing will look orderly.

In the second field, type JSR and press the TAB key again. The cursor then moves to the third field, which is called the operand field. An operand is the information needed by an opcode. For example, a JSR by itself wouldn't tell the assembler where the JSR was destined. Type $FC58 here for the operand, and press the TAB key again. In the APW assembler, you have to use the dollar sign ($) in front of all hex numbers. If you don't, APW will treat it at a decimal number.

The fourth, and last, field is called the comment field. This is where you can add text to explain what a specific line does. It is customary (some other assemblers insist on it) to begin a comment with a semicolon. Type "; CLEAR SCREEN" and press Return.
Assembling a Program with \textit{APW}

If all went well, your program should look like this:

```
******************************************
* 1st APW Program                     *
* By <your name>                      *
******************************************
BEGIN JSR $FC58 ; CLEAR SCREEN
```

For the second line, we won't use a label. Actually, a label is always optional unless it is directly referenced somewhere else in the program, for example by a JSR somewhere. In that light, we didn't really need the label at the beginning of the first line of the program, but we'll leave it there since it doesn't hurt anything.

Since we don't need a label for the second line, press the TAB key once to move the cursor to the opcode field. Now type LDA, TAB, and #$C1. Then press the TAB key once more, type

; LETTER "A"

and press Return. For the next line, TAB to the second field and type in:

```
STA $5BC ; SCREEN LOCATION
```

and press Return. Finally, finish the program with the label DONE, and an RTS. You don't need an operand for RTS, and you can omit the comment too.

Program 5-1 shows how your program should now look.

Program 5-1. First Program

```
******************************************
* 1st APW Program                     *
* By <your name>                      *
******************************************
BEGIN JSR $FC58 ; CLEAR SCREEN
LDA #$C1 ; LETTER "A"
STA $5BC ; SCREEN LOCATION
DONE RTS
```

\textbf{Assembler Directives}

Before assembling, there's one more requirement. The \textit{APW} assembler assumes, and rather insistently too, that everything you assemble is just a smaller part of some larger program. These pieces are called \textit{code segments}, and every code segment must have a \textbf{START} and an \textbf{END} assembler command in it.

It also assumes that our program will be operating with the Accumulator in the long, or two-byte wide, mode of the 65816 microprocessor. Since that
Chapter 5

has not been covered yet, we'll need to add a line to tell it not to use the long Accumulator mode quite yet.

To add these new lines to the listing, first go to the end of the listing, TAB to the second field, and type END. Then use the up arrow to move to line 6, and insert these lines:

LONGA OFF
MAIN START

The new listing is shown in Program 5-2.

Program 5-2. First APW Program

******************************************
* 1st APW Program
* By <your name>
******************************************

    LONGA OFF
    MAIN START

BEGIN JSR $FC58 ; CLEAR SCREEN
    LDA #$C1 ; LETTER "A"
    STA $5BC ; SCREEN LOCATION
DONE RTS

END

START, END, and LONGA are not true 65816 opcodes. Instead, they're instructions specifically for the assembler and are called assembler directives, because they direct the assembler to take a certain action. Most assemblers have quite a number of directives to make assembling programs easier.

START and END are special APW directives that mark out the beginning and end of a program segment. LONGA OFF is a directive that tells the assembler not to use the long mode for the LDA and STA instructions.

Assembling a Program

To assemble this program, you need to leave the full-screen editor. This is done by typing Control-Q. The screen will change to the editor file menu. Press S to save the source file to disk. Then press E to exit back to the APW shell.

The actual assembly is started by typing

ASSEMBLE +L +S APW.1.S

When you press Return, the disk will come on for a while, the text shown in Program 5-3 will appear on the screen.
Assembling a Program with APW

Program 5-3. Assembly of First APW Program

<table>
<thead>
<tr>
<th>Line</th>
<th>Address</th>
<th>Code</th>
<th>Label</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0005</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0006</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0007</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0008</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0009</td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>0000</td>
<td>2058</td>
<td>BEGIN</td>
<td>JSR $FC58 ; CLEAR SCREEN</td>
</tr>
<tr>
<td>0011</td>
<td>0003</td>
<td>A9 C1</td>
<td>LDA</td>
<td>#$Cl ; LETTER “A”</td>
</tr>
<tr>
<td>0012</td>
<td>0005</td>
<td>8D BC05</td>
<td>STA</td>
<td>$5BC ; SCREEN LOCATION</td>
</tr>
<tr>
<td>0013</td>
<td>0008</td>
<td>60</td>
<td>DONE</td>
<td>RTS</td>
</tr>
<tr>
<td>0014</td>
<td>0009</td>
<td></td>
<td></td>
<td>END</td>
</tr>
<tr>
<td>0015</td>
<td>0009</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Symbols

000000 BEGIN 000008 DONE

15 source lines
0 macros expanded
0 lines generated

There are a number of things to notice here. First, APW has created a listing at the far left of each line number in your program, and the relative addresses where each instruction will ultimately be put in memory. The APW is different than the Merlin in that it doesn’t use any absolute memory addresses until the very end of the program construction process, and sometimes it doesn’t use them then.

You can see each line indicates a memory address, starting at 0000 followed by the actual machine language codes for each instruction of your program.

At the end of the listing is the symbol table. This prints a list of all the labels used in the program, such as BEGIN and DONE.

In the ASSEMBLE command, the +L and +S tell the assembler to print out the assembled listing to the screen and to add the symbol table at the end. If you don’t wish to see these, just omit these parts of the command line.

To save the program to your disk, you need to tell the assembler what filename to use. To do this, type

ASSEMBLE APW.1.S KEEP=APW.1

KEEP is an APW directive for saving the output file.

After the assembly is complete, type CATALOG. On the disk you should
see your source file, APW.1.S, and also the file APW.1.ROOT. This is an inter-
mediate file created by the APW assembler, and is called an object file by Apple
Computer in the APW environment. Contrary to the common usage of the term
object file, this file is not directly executable program code. Instead it consists of
the program plus information to be used when that segment is ultimately com-
bined with other segments in the big program.

To convert the object file into something that can be used somewhere re-
quires a few more steps.

**The Linker**

Another part of the APW shell is something called the Linker. This is the sys-
tem that pulls all those pieces together when constructing a large program. Fortu-
nately, there is a command that will both assemble the file and link our file
into an almost-final file. To assemble and link your program, type

```
ASML APW.1.S KEEP= APW.1
```

ASML is the APW shell command to do both the assembly and the link
of the source file.

After this assembly, CATALOG the disk again. Now, in addition to your
source file and the intermediate APW.1.ROOT file, you should see the desired
APW.1 file. You'll notice that its file type is EXE. This means that it is a file
type executable within the APW shell. Unfortunately, our programs aren't quite
yet sophisticated enough to survive outside of the normal Applesoft BASIC/
ProDOS 8 environment, so we'll need one more step to get what we need.

The file type you need is a binary type (BIN). To change the output file
EXE into a binary file, you need to use the APW shell command MAKEBIN. To
use this now, type

```
MAKEBIN APW.1 ORG = $300
```

**ORG** is another assembler directive used to tell the assembler where you
want the program to run. In just a moment, we'll see how to include this in the
assembly listing itself.

If you catalog the disk again, you'll see the file type of APW.1 has now
changed to BIN.

To run the assembled file, you'll have to quit APW and go to Applesoft
BASIC or to a program selector other than Apple's Program Launcher—one
that can run Binary files. Applesoft BASIC is the recommended method for now.

Type QUIT to quit the APW.

When you're in Applesoft BASIC, you'll probably have to reset the pre-
fix to your /PROGRAMS disk before you can run your program APW.1.
Assuming the prefix is correct, and that the /PROGRAMS disk is in a drive, you can now type either

`BRUN APW.l`

or

`-APW.l`

to run the program. You can also type

`BLOAD APW.l`
`CALL 768`

You’ll recall that in using the `MAKBIN` command, you specified the starting address $300 (768 decimal) for the file. Using the `ORG=` statement in the `MAKBIN` command is one way to tell the assembler where you want the code to ultimately run. The other is to put the `ORG` directive in the program source listing itself. To do that, we’ll need to go back to the `APW`. Type `BYE` in the immediate mode to quit `BASIC` back to a program launcher, and from there go back to the `APW`.

By the way, this long cycle time between the `APW` and Applesoft `BASIC` is the other disadvantage of using the `APW` at this stage in your learning. Eventually, you’ll be able to write self-contained programs that can be run directly from the Program Launcher, or even from within the `APW` shell, but for now, changing back and forth is necessary.

**More Assembler Directives: ORG, KEEP, and EQU**

When the `APW` is back up and running, load your source file `APW.1` by typing

`EDIT APW.1.S`

Before entering any new text, type `Open Apple-E` to put the editor in the insert mode. When it is, the words `EDIT INSERT` will be displayed in the status line at the bottom of the screen. Move the cursor down, using the down arrow, to line 5 (the blank line) and press Return. A new blank line should be inserted. When the editor is in the insert mode, pressing Return will insert a new blank line.

Press the `TAB` key once to go to the opcode field, and type `ORG`. Then press the `TAB` key, and type `$300` and press Return twice.

Next press the `TAB` key, type `KEEP`, tab to the next field, and type `APW.1` and then press Return. The listing should now look like Program 5-4.
Chapter 5

Program 5-4. First APW Program with KEEP Directive

***************
* 1st APW Program *
* By <your name>  *
***************

ORG $300
KEEP APW.1
LONGA OFF

MAIN START
BEGIN JSR $FC58 ; CLEAR SCREEN
  LDA #$C1 ; LETTER “A”
  STA $5BC ; SCREEN LOCATION
DONE RTS

END

Like ORG, KEEP is another assembler directive. It tells the assembler what name to use in creating the output files.

After you’ve added the new lines with ORG and KEEP, press Control-Q to go to the editor file menu, and save the new listing with the S command. Then assemble and link the source file with the command

ASML APW.1.S

You’ll notice that this time you don’t need to use the KEEP command. When the assembly is done, convert the EXE file APW.1 to a binary file by typing

MAKEBIN APW.1

Since the ORG directive was included in the source file, you don’t need to use it here.

Try running this file from Applesoft BASIC to prove that it is equivalent to the first. You may also want to use the Monitor (CALL -151 from Applesoft BASIC) to look at the object file in memory.

More About Labels
Now go back to the APW again and load APW.1. The listing should look like Program 5-4.

Looking at the listing, you can see addresses like $FC58 and $5BC, and the code value for the letter A, which is $C1. As a program gets larger, and the listing longer, it can get confusing trying to remember a lot of different number
values. To make programming more convenient, most assemblers allow you to give routines a name and they remember the address for you.

Start by putting the cursor on line 13, press Return once, and type the following:

```
HOME EQU $FC58
SCREEN EQU $5BC
```

Now move the cursor to the end of the address $FC58, and press the Delete key until it's completely erased. Now type HOME, but don't press Return; instead, use the down arrow to move down two lines, and do the same thing to replace the address $5BC with SCREEN. The Program is shown in Program 5-5.

Program 5-5. First APW Program with EQU

******************************************
* 1st APW Program
* By <your name>
******************************************

```
KEEP APW.1
ORG $300
LONGA OFF
MAIN START
HOME EQU $FC58
SCREEN EQU $5BC
BEGIN JSR HOME ; CLEAR SCREEN
LDA #"A" ; LETTER "A"
STA SCREEN ; SCREEN LOCATION
DONE RTS

END
```

Quit the editor with Control-Q, save the source file, and then assemble by typing

```
ASML +L +S APW.1.S
```

Since we've added the +L command, the listing will appear on the screen, and should look like Program 5-6.

The main items of interest here are the listing itself and the Local Symbols listing, although you may find the other information interesting.

You'll notice that this time the assembler translates the labels HOME and SCREEN into their assigned addresses. Labels are always associated with some value in the source listing.
Chapter 5

Program 5-6. Assembly of First APW Program with EQU

0001 0000  ******************************************
0002 0000  * 1st APW Program  *
0003 0000  * By <your name>  *
0004 0000  ******************************************
0005 0000
0006 0000  ORG  $300
0007 0000
0008 0000  KEEP  APW.1
0009 0000
0010 0000  LONGA  OFF
0011 0000
0012 0000  MAIN    START
0013 0000
0014 0000  HOME    EQU  $FC58
0015 0000  SCREEN  EQU  $5BC
0016 0000  LETTERA EQU  $C1
0017 0000
0018 0000  20 58 FC  BEGIN  JSR  HOME  ; CLEAR SCREEN
0019 0003  A9 41  LDA  A"A"  ; LETTER "A"
0020 0005  8D BC 05  STA  SCREEN  ; SCREEN LOCATION
0021 0008  60  DONE  RTS
0022 0009
0023 0009  END

Local Symbols

000000 BEGIN 000008 DONE 00FC58 HOME 0000C1 LETTERA
0000BC SCREEN

23 source lines
0 macros expanded
0 lines generated
Link Editor V1.0 B3.2

Segment:

00000300 00000009 Code: MAIN

Global symbol table:

00000300 G 01 00 MAIN

Segment Information:

<table>
<thead>
<tr>
<th>Number</th>
<th>Type</th>
<th>Length</th>
<th>Org</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$00</td>
<td>$00000009</td>
<td>$00000300</td>
</tr>
</tbody>
</table>

There is 1 segment, for a length of $00000009 bytes.
Whenever the assembler encounters a label, either at the beginning of a line or following the EQU directive, it assigns a value (usually an address) to it. For EQU statements, it uses whatever value is on the line. If the label is at the beginning of the line, it uses its own internal address counter for where it is in the program.

For example, since BEGIN is the first statement in the program, the assembler gives BEGIN a relative value of $0. As it continues to assemble each line, it keeps track of where it is. By the time it gets to DONE, its address counter is at $8, and that’s what it assigns to DONE. Thus, if some other part of the program wanted to do a JSR, for example, BEGIN or DONE, you could just use the line

```
JSR DONE
```

or

```
JSR BEGIN
```

In this program, such a JSR wouldn’t make much sense—just as a GOSUB to the first or last line in a BASIC program wouldn’t make sense. But, as your programs get larger, you’ll want to go to subroutines within the program. In assembly language, you use the labels, not line numbers, to tell the JSR where you want it to go. The label MAIN is not included in the local symbol table, because it is not considered to be within your program by APW. It’s listed in the Global Symbol table, and it has been given an address value of $300 there.

**Labels as Constants**

Labels can be used for more than addresses. We could also assign a label to the value for the letter A (Program 5-7).

There’s nothing wrong with assigning a constant (as opposed to an address) with an EQU statement, it just takes up a little more room in your listing and the computer’s memory. This technique is used when you want to use a number value throughout a listing, but want to keep your options open so you can change your mind later. For example, suppose you’re testing a program that goes through a loop 200 times, and that the number 200 is used in many different places in the listing. During testing, it might be nice to just use the number 5, for a shorter loop, instead. Using the label

```
COUNT EQU 5
```

makes it easy to change later, without having to search through the listing for all the number 5’s on every line.
There’s also another common error to avoid. Sometimes people define a constant like this:

```
COUNT EQU #$5
```

and then later try to do this:

```
LDA COUNT
```

They think that the pound sign is included in the label COUNT. It isn’t. The assembler evaluates each operand as a numeric expression, and then it stores the result as a the number value, not the text of the line that generated the result. The instruction is assembled as though the line read

```
LDA $5
```

When the program runs, the accumulator is loaded with the contents of location $5, not the actual value 5. Because there is no error generated during the assembly, and because the value loaded into the accumulator can be almost anything (and usually is), this kind of mistake can be very difficult to track down.

Therefore, be sure to remember to always put the pound sign in front of any labels that you want used as an absolute value.

```
COUNT EQU $5
LDA #COUNT
```

Program 5-7. Using EQU to Define a Character

```
***************************************************
* 1st APW Program                                   *
* By <your name>                                    *
***************************************************
ORG $300
KEEP APW.1
LONGA OFF
MAIN START
HOME EQU $FC58
SCREEN EQU $5BC
LETTERA EQU $C1
BEGIN JSR HOME ; CLEAR SCREEN
    LDA #LETTERA ; LETTER “A”
    STA SCREEN ; SCREEN LOCATION
DONE RTS
END
```
Long Addresses in \textit{APW}

In the \textit{APW} assembler, the assembler automatically decides which form, long or short, of LDA and STA to use, depending on operand. If the address evaluates to $FFFF$ or less, the short form is used. If the address evaluates to three or more bytes ($01/0000$ or greater), the long address form is used. If you know your address will evaluate to only two bytes, and you still want to force the long addressing mode, you can precede the label with a greater-than sign ( $>$ ).

Program 5-8 is an assembly program segment that illustrates some of the different instructions possible.

Program 5-8. \textit{APW} Long Addressing Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Label</th>
<th>Address</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001 0000</td>
<td></td>
<td>MAIN</td>
<td>START</td>
<td></td>
</tr>
<tr>
<td>0002 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0004 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0005 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0006 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0007 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0008 0000</td>
<td></td>
<td>LABEL1</td>
<td>EQU $050300 ; $05/0300</td>
<td></td>
</tr>
<tr>
<td>0009 0000</td>
<td></td>
<td>LABEL2</td>
<td>EQU $061A00 ; $06/1A00</td>
<td></td>
</tr>
<tr>
<td>0010 0000</td>
<td></td>
<td>LABEL3</td>
<td>EQU $300    ; $00/0300</td>
<td></td>
</tr>
<tr>
<td>0011 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0012 0000</td>
<td>AF 00 03 05</td>
<td>LDA</td>
<td>LABEL1</td>
<td></td>
</tr>
<tr>
<td>0013 0004</td>
<td>8F 00 1A 06</td>
<td>STA</td>
<td>LABEL2</td>
<td></td>
</tr>
<tr>
<td>0014 0008</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0015 0008</td>
<td>AF 00 03 00</td>
<td>LDA</td>
<td>$&gt;$LABEL3</td>
<td></td>
</tr>
<tr>
<td>0016 000C</td>
<td>8F 00 03 00</td>
<td>STA</td>
<td>$&gt;$LABEL3</td>
<td></td>
</tr>
<tr>
<td>0017 0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0018 0010</td>
<td>AD 00 03</td>
<td>LDA</td>
<td>LABEL3</td>
<td></td>
</tr>
<tr>
<td>0019 0013</td>
<td>8D 00 03</td>
<td>STA</td>
<td>LABEL3</td>
<td></td>
</tr>
<tr>
<td>0020 0016</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0021 0016</td>
<td>22 00 1A 06</td>
<td>JSL</td>
<td>LABEL2</td>
<td></td>
</tr>
<tr>
<td>0022 001A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0023 001A</td>
<td></td>
<td></td>
<td></td>
<td>END</td>
</tr>
</tbody>
</table>

In the first case (lines 12, 13), LABEL1 and LABEL2 both evaluate to three bytes, so the long address mode of LDA and STA are used. In the second case (lines 15, 16), LABEL3 only evaluates to two bytes. $0300$. The greater-than symbol is used to force the assembler to use the long address form. Without the greater-than sign, the assembler would use the short address form, as shown on lines 18, 19. Line 21 shows the assembly of the JSL instruction.
Chapter 5

**APW in Review**

If you've worked through the examples in this chapter, you should be able to enter, edit, assemble, and save both a source listing and the object file using the APW assembler. To learn more about editing, such as how to cut, copy and paste sections, or how to use search and replace, you should read the manual that came with APW.

You should also now understand how to use labels in a source program to represent a given number value or address, how to use ORG and EQU to control the values that labels are assigned, and where the program will load into memory from disk.
Chapter 6

Loops and Counters
In BASIC, the FOR-NEXT loop is an important part of many programs; this is also true in assembly language programming. The only difference is how the loop-counter combination is actually carried out.

In BASIC, the testing of counters is done either by IF-THEN statements or automatically in the NEXT statement of a FOR-NEXT loop. In assembly language, the testing is done by examining flags in the Processor Status Register (Figure 6-1). These flags indicate the results of the last mathematical operation of the 65816; general zero/nonzero conditions of numbers loaded into the X, Y and Accumulator registers; and other handy things within your program.

Figure 6-1. 65816 Microprocessor Model

<table>
<thead>
<tr>
<th>Accumulator</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor Status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>P</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prog. Bank Reg. (PBR)</th>
<th>Program</th>
<th>Counter (PC)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Processor Status Register, abbreviated P in the 65816 model, is the fourth register of the 65816, one not previously mentioned. Before going on
with loops and counters, it’s necessary to briefly discuss the Status Register and binary numbers.

Different from the other three registers (the Accumulator and the X and Y registers) the Status Register only holds a single byte. You’ll recall that each byte in the Apple can have a value from 0 to 255 ($00 to $FF).

As it happens, there are many ways of looking at and interpreting numbers. One common way of looking at numbers is to consider only size. When we notice that 255 is larger than 128, we gain a very simple bit of information—we learn whether a number is either less than, equal to, or greater than another number.

A second way of looking at numbers is to use the digits that make up the number as a carrier of information. For example, street addresses generally increment by 100 for each city block, regardless how many houses are actually on the street. The address 9105 is usually one block further than 9005. The leading two digits tell you how many blocks away from the first street the address is.

Binary Numbers

In the binary number system, you’re limited to counting by groups of 2. However, this system allows you to see more information in a number. This in turn can make it that much more useful for an assembly language program.

We have already seen how a single byte can be represented either as 0 to 255 or $00 to $FF. In the binary system, the same byte can have the range of 00000000 to 11111111. For instance, 133 (base ten) was represented as $85 (hexadecimal). In binary it has the appearance of 10000101. Each of the eight positions in a binary number are called bits (a word created from the term Binary digits). There are eight bits in a byte. In this case, each 1 or 0 can represent the presence or absence of a given condition. Thus, eight distinct pieces of information are conveyed, as well as all the various combinations possible.

Before you run shrieking from the room, remember that this is all done to make things easier (really), not harder. Besides, learning base sixteen (hex) wasn’t that bad a few chapters back, was it? So, let’s take a moment to see what these bits and bytes are all about.

The Apple is an electronic device and, actually, in many ways, a simple one at that. In most parts of its circuitry, the flow of electricity is either off or on. That’s it. No in-between. Having two possible conditions is perfect for base two.

The idea of a number base has to do with how many symbols, or units, you use for counting. We normally use 10. There are a total of 10 possible symbols to write in a single position before we have to start doubling up and using two positions to represent a number. You’ll recall that in hex, by using 0
through 9 and A through F we had 16 possibilities; thus we were in base 16.
With the on/off nature of the Apple, we're limited to 2 possibilities: 0 or 1.

How high can we count with just two symbols in one position? Not very. We start at 0, then go to 1, and that's it—we're out of symbols. Then we have to add another position. The next number, therefore, is 10. As before, remember that, in this case, 10 represents what we usually call two. 100 would represent the quantity 4 in base ten.

By using eight positions, we can go up to 11111111, which just happens to be 255. This is the same maximum value as our bytes. And, if the truth be known, it's not just coincidence—the value 255 is a result of the computer being based on the binary system of numbering. We use the numbers 0 through 255 because we are using eight bits to make up each byte. Whether each bit is 0 or 1 depends on whether the part of the electrical circuit that is responsible for that bit is off or on.

Counting in base 2:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>$00</td>
<td>000</td>
</tr>
<tr>
<td>00000001</td>
<td>$01</td>
<td>001</td>
</tr>
<tr>
<td>00000010</td>
<td>$02</td>
<td>002</td>
</tr>
<tr>
<td>00000011</td>
<td>$03</td>
<td>003</td>
</tr>
<tr>
<td>00000100</td>
<td>$04</td>
<td>004</td>
</tr>
<tr>
<td>00000101</td>
<td>$05</td>
<td>005</td>
</tr>
<tr>
<td>00000110</td>
<td>$06</td>
<td>006</td>
</tr>
<tr>
<td>00000111</td>
<td>$07</td>
<td>007</td>
</tr>
<tr>
<td>00001000</td>
<td>$08</td>
<td>008</td>
</tr>
<tr>
<td>00001001</td>
<td>$09</td>
<td>009</td>
</tr>
<tr>
<td>00001010</td>
<td>$0A</td>
<td>010</td>
</tr>
<tr>
<td>00001011</td>
<td>$0B</td>
<td>011</td>
</tr>
<tr>
<td>00001100</td>
<td>$0C</td>
<td>012</td>
</tr>
<tr>
<td>00001101</td>
<td>$0D</td>
<td>013</td>
</tr>
<tr>
<td>00001110</td>
<td>$0E</td>
<td>014</td>
</tr>
<tr>
<td>00001111</td>
<td>$0F</td>
<td>015</td>
</tr>
<tr>
<td>00010000</td>
<td>$10</td>
<td>016</td>
</tr>
<tr>
<td>00010001</td>
<td>$11</td>
<td>017</td>
</tr>
<tr>
<td>00010010</td>
<td>$12</td>
<td>018</td>
</tr>
<tr>
<td>00010011</td>
<td>$13</td>
<td>019</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111000</td>
<td>$F8</td>
<td>248</td>
</tr>
<tr>
<td>11111001</td>
<td>$F9</td>
<td>249</td>
</tr>
</tbody>
</table>
Chapter 6

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111010</td>
<td>$FA</td>
<td>250</td>
</tr>
<tr>
<td>11111011</td>
<td>$FB</td>
<td>251</td>
</tr>
<tr>
<td>11111100</td>
<td>$FC</td>
<td>252</td>
</tr>
<tr>
<td>11111101</td>
<td>$FD</td>
<td>253</td>
</tr>
<tr>
<td>11111110</td>
<td>$FE</td>
<td>254</td>
</tr>
<tr>
<td>11111111</td>
<td>$FF</td>
<td>255</td>
</tr>
</tbody>
</table>

Although binary numbers may seem overwhelming in the sheer visual size of each number, it’s fairly unlikely you’ll ever have to convert a binary number to decimal, or even to hexadecimal. It’s discussed here mainly so that you’ll have seen the underlying principles of how number values in the computer are created, and why the terms binary and bits show up once in a while.

You might notice, however, that there is a distinct correlation between the binary number patterns and the hex number patterns, whereas there is no apparent visual relationship between binary and decimal. For example, in the hex column, the number changes to $10 at the same point that the binary number changes from 00001111 to 00010000. In fact, the very observant will notice that the two digits in the hex number are made up from the upper and lower four bits (called a nibble) of the binary number. If you can count from $0 to $F (0 to 15 decimal) in the binary system, you can convert any binary number you see to its hexadecimal equivalent.

For example:

10011101 (binary) =
1001  = 9  = $90
1101  = 13 = $0D

10011101 = $9D

The Status Register

As was mentioned above, the bits that make up a number can be used as flags or indicators for eight independent conditions. By individually setting or clearing the eight bits in a byte, the computer can store the status of eight different things in just one byte. That’s the idea behind the Status Register.

Here is a representation of a single byte, made up of eight bits. In particular, the byte shown in Figure 6-2 is the Status Register of the 65816. The important difference between this register and the others is that it’s not used to store number values. Instead, it indicates various conditions.
The bits of the Status Register are numbered from right to left, 0 to 7. Each bit in this register indicates the results of different operations, and is called a flag. It’s by using this register that we can create counters and loops in our programs. Bit 1 is the zero flag. In terms of commands discussed so far, the zero flag is affected by an LDA, LDX, or LDY.

If the value loaded into the Accumulator, X, or Y register were $00$, the flag would be set to 1. If it were a nonzero number, the flag would be 0. Seemingly backward perhaps, but remember each flag is set to show the presence or absence of a given condition; in this case the presence of 0. The setting or clearing of each Status Register flag is done automatically by the 65816 after each program step, indicating the results of any particular operation.

Incrementing and Decrementing

To create a counter and then a loop, we’ll use the Status Register to tell when a given register or memory location reaches zero. We will also need a way of changing the value of a counter in a regular fashion. In the 65816, this is done by incrementing or decrementing (adding or subtracting 1 each time) a memory location, or the X or Y register. This is done with one of the following commands:

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>X Register</th>
<th>Y Register</th>
<th>Memory Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment:</td>
<td>INC</td>
<td>INX</td>
<td>INY</td>
</tr>
<tr>
<td>(Add 1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decrement:</td>
<td>DEC</td>
<td>DEX</td>
<td>DEY</td>
</tr>
<tr>
<td>(Subtract 1)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This table shows the commands used to increment or decrement a particular register or memory location.
Chapter 6

The usual syntax for using these commands is an assembly listing as illustrated by the following sample lines:

```
INX ; Add 1 to the X register
INY ; Add 1 to the Y register
INC $0600 ; Add 1 to a memory location
DEX ; Subtr. 1 from the X register
DEY ; Subtr. 1 from the Y register
DEC $AA53 ; Subtr. 1 from a memory location
```

For the X and Y register operations, the command stands alone, with no need of an operand. In the case of INC and DEC, the memory locations to be operated on are given.

In the Merlin assembler, INC and DEC are used for both the Accumulator and a memory location. For example,

```
INC ; Add 1 to accumulator
INC MEM ; Add 1 to location mem
```

In the first line, INC alone tells the assembler to increment just the Accumulator. By adding the label MEM, it knows it needs to increment a memory location.

Some assemblers use the opcodes INA and DEA for incrementing and decrementing the Accumulator, others, like the APW, require you to put the letter A in the operand field, like this:

```
INC A ; ADD 1 TO ACCUMULATOR
DEC A ; SUBTRACT 1 FROM ACCUMULATOR
```

The increment/decrement commands affect the zero flag, depending on whether the result of the operation is 0 or not.

One thing to mention here is the wraparound nature of all the operations. To understand this, examine the following chart.

<table>
<thead>
<tr>
<th>Original Value</th>
<th>Result of Increment</th>
<th>Result of Decrement</th>
<th>Z-Flag Set</th>
<th>Z-Flag Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$05</td>
<td>$06</td>
<td>$04</td>
<td>no; no</td>
<td>0; 0</td>
</tr>
<tr>
<td>$0F</td>
<td>$10</td>
<td>$0E</td>
<td>no; no</td>
<td>0; 0</td>
</tr>
<tr>
<td>$01</td>
<td>$02</td>
<td>$00</td>
<td>no; yes</td>
<td>0; 1</td>
</tr>
<tr>
<td>$FF</td>
<td>$00</td>
<td>$FE</td>
<td>yes; no</td>
<td>1; 0</td>
</tr>
<tr>
<td>$00</td>
<td>$01</td>
<td>$FF</td>
<td>no; no</td>
<td>0; 0</td>
</tr>
</tbody>
</table>

The effects of incrementing and decrementing different one-byte values are shown, along with the effects on the zero flag after the operation. The first case is simple: \( 5 + 1 = 6; 5 - 1 = 4 \). In both cases, the result is a nonzero number, so the zero flag is not set. For $0F, the same holds true. Remember
that, in hex, the number after $0F is $10. In the case of $01, incrementing produces $02. When we decrement $01, the result is $00; the zero flag is set.

Here's where it gets interesting. When the starting value is $FF, adding 1 would normally give $100. However, since a single byte only has a range of $00 to $FF, the new 1 is ignored, and the value becomes $00. This sets the zero flag. In the case of decrementing $FF, $FF - 1 = $FE, so the zero flag is not set.

If we start with $00, although incrementing produces the expected $01, decrementing wraps around in the reverse of the previous case, giving $FF. Both results are nonzero, so Z (short for the Z-flag), is clear—that is, it's not set—for both operations.

**Looping with BNE**

The only thing missing now to enable you to create a loop is a way of testing the Z flag and then being able to get back to the top of the loop for another pass. In BASIC, a simple loop might look like this:

```
10 HOME
20 X = 255
30 PRINT X
40 X = X - 1
50 IF X <> 0 THEN GOTO 30
60 END
```

In this program, the counter X is set to 255. The value is printed and then decremented, and the process repeated until the counter reaches zero. We can make the loop execute any number of times by properly setting the initial value of X.

In assembly language, the test and the GOTO are done with a *branch instruction*. In this case, the one we'll use is BNE, which stands for **Branch Not Equal** (to zero). This is a *conditional instruction*, and will be executed only when a register is loaded with a nonzero number. This can happen either directly with something like LDA #$01, or as the result of an arithmetic operation, such as INX. Program 6-1 is the assembly language equivalent of the BASIC listing.

```
107
```

This listing shows both the source lines, and the object code output on the left, as it would be printed during an assembly.

The program can be tested by going to Applesoft BASIC, BLOADing it (it should load automatically at $300), and then doing a CALL 768.

**Note:** If you're using the APW assembler, be sure to add the START, END, and KEEP directives. Review Chapter 5 if you are unsure as to how to create a binary file using APW.
Chapter 6

The program starts with the usual screen clear, and then loads the X register with a starting value of $FF. The loop starts by storing the contents of the X register at $5BC; this will make the loop's action visible as a character on the screen for each pass through the loop. Lines 10, 15, and 16 introduce a new monitor routine, \texttt{WAIT} = $FCA8, which is a delay function based on the contents of the Accumulator. This is required because, without it, the loop would execute so quickly you couldn't see it in action. Experiment with different values in the Accumulator on line 16 to see what effect they have; and try eliminating the JSR \texttt{WAIT} altogether for maximum speed.

On line 17, DEX subtracts one from the current value of the X register. The BNE will then continue the loop back up to LOOP until the X register reaches $00, at which point the test will fail, and program execution will fall through to the RTS at the end of the program.

Program 6-1. Loop Demo Routine 1

\begin{verbatim}
ORG $300

; CLEAR VIDEO SCREEN
000300: 58 FC 12 BEGIN JSR HOME ; X = 255
000303: FF 13
000305: BC 05 14 LOOP STX SCREEN ; MAXIMUM WAIT TIME
000308: FF 15 LDA #$FF ; ACC = 255
00030A: FC 03 16 JSR WAIT ; BRANCH IF X <> 0
00030D: 50 17 DEX ; THAT'S ALL FOLKS!

000310: 60 19 DONE RTS
\end{verbatim}

--End Merlin-16 assembly. 17 bytes, Errors: 0

\textbf{Looping with BEQ}

The complement of the BNE instruction is \texttt{BEQ}, \texttt{Branch EQual} (to zero). It operates in just the opposite fashion as BNE, that is, branching only when the register or memory location reaches a value of zero.

For example, consider this BASIC listing:

\begin{verbatim}
10 HOME
20 X = 255
\end{verbatim}
Loops and Counters

30 PRINT X
40 X=X-1
50 IF X=0 THEN 70
60 GOTO 30
70 END

In this case, the loop continues as long as X is not equal to zero. If it is, the branch instruction is carried out and the program ends. The equivalent in assembly language is shown in Program 6-2.

Program 6-2. Loop Demo Routine 2

```
1 ******************************************
2 * LOOP PROGRAM #2 *
3 * MERLIN 8/16 ASSEMBLER *
4 ******************************************
5
6 ORG $300
7
8 HOME EQU $FC58
9 SCREEN EQU $05BC
10 WAIT EQU $FCA8

000300: 20 58 FC JSR HOME ; CLEAR SCREEN
000303: A2 FF 13 LDX #$FF ; START COUNTER AT 255
000305: SE BC 05 14 WOP STX SCREEN ; PUT CHAR ON SCREEN
000308: A9 80 15 LDA #$80 ; TIME DELAY VALUE
00030A: 20 AS FC JSR WAIT
00030D: CA 17 DEX ; X = X - 1
00030E: F0 03 0313 18 BEQ DONE ; DONE IF X = 0
000310: 4C 05 03 19 JMP LOOP ; NEXT CYCLE
000313: 60 20 DONE RTS ; ALL DONE!

--End Merlin-16 assembly, 20 bytes, Errors: 0
```

Notice that this program also uses a new command, JMP (Jump). JMP is like a GOTO in BASIC. It doesn’t expect an eventual RTS. The JMP on line 19 will cause program execution to jump to the routine starting at LOOP each time. Only when the X-register reaches zero does the BEQ take effect and cause the program to skip to the RTS at end. Here is the way this would appear when put into memory, and then listed with the L command from the Monitor:

```
*300L
1=m 1=x 1=LCbank (0/1)
0/0300: 20 58 FC JSR FC58
109
```
Chapter 6

00/0303: A2 FF LDX #FF
00/0305: 8E BC 05 STX 05BC
00/0308: A9 80 LDA #80
00/030A: 20 A8 FC JSR FCA8
00/030D: CA DEX
00/030E: F0 03 BEQ 0313 { +03}
00/0310: 4C 05 03 JMP 0305
00/0313: 60 RTS

The assembler automatically translates the positions of LOOP and END into the appropriate addresses to be used by the BEQ and JMP when it assembles the code.

Remember that to the left are the addresses and the values for each opcode and its accompanying operand. The more understandable translation to the right is Apple's interpretation of this data.

Notice that the JMP's and JSR's are immediately followed by the address (reversed) that they're to jump to, such as in the first JSR at $300.

However, branch instructions are handled a differently. As opposed to a JSR or JMP that go to an absolute memory location, a branch goes to a relative memory address. That is to say, it goes to an address relative to where the branch instruction itself is located. At $30E, the $F0 is the opcode for BEQ. The $03 that follows is an offset that tells the 65816 to branch down through the code three bytes from the address of the next instruction that follows the branch itself (at $310). Adding $03 to $310 gives us $313, the address of the desired RTS.

Branching in the reverse direction (up through the listing) is also possible and is shown by operands greater than $80. There is not much need of going into great detail about the actual calculations used, since your assembler will determine the proper values for you when assembling code, and Apple's disassembler will give the destination address when reading other code.

As the X register is incremented in this program, we'll stuff the value into the screen location so we can see something on the screen as the counter advances.

There are also two other branch instructions, BRA (Branch Always), and BRL (Branch Long), that you can use instead of a jump instruction. BRA can only be used when you don't have to branch more than 127 bytes forward or backward. BRL will branch to any address in the 64K address space. The main advantage of BRA is for creating programs that are position independent.

When a program is position independent, it doesn't matter where it is loaded into memory. It will run equally well anywhere. For example, if you loaded Program 6-2 at location $8000, and tried to run it, what would happen...
Loops and Counters

when it got to the JMP LOOP on line 19? Remember, the assembler assembled this line as JMP $305. If the program were loaded at $8000, there wouldn’t be anything related to this program at $305 when the JMP was executed. This would probably result in a program crash, with one of those unpleasant BRK messages.

An alternative is to use the BRA instruction as shown in Program 6-3.

Program 6-3. Loop Demo Program 2A

```
* ******************************************
* LOOP PROGRAM #2A *
* MERLIN 8/16 ASSEMBLER *
* ******************************************

ORG $300

=FC58 HOME EQU $FC58
=05BC SCREEN EQU $05BC
=FCA8 WAIT EQU $FCA8

000300: 20 58 FC  BEGIN JSR HOME ; CLEAR SCREEN
000303: A2 FF  LDX #$FF ; START COUNTER AT 255
000305: SE BC 05 LOOP STX SCREEN ; PUT CHAR ON SCREEN
000308: A9 80  LDA #$80 ; TIME DELAY VALUE
00030A: 20 AS FC JSR WAIT
00030D: CA  DEX ; X = X - 1
00030E: F0 02 =0312 BEQ DONE ; DONE IF X = 0
000310: 80 F3 =0305 BRA LOOP ; NEXT CYCLE
000312: 60 DONE RTS ; ALL DONE!
```

--End Merlin-16 assembly, 19 bytes, Errors: 0

This is also a good time to stress the importance of working through each of these examples on your own, step by step, to make sure that you understand exactly what happens at each instruction, and how it relates to the rest of the program.

Incrementing Two or More Bytes

The discussion of the increment and decrement commands has so far been limited to the assumption that a single byte is involved. Suppose, however, that you wanted to store the value for an address, which requires at least two bytes, not counting the bank byte. By using the BEQ instruction you can create a procedure, or code segment, that will increment or decrement a pair of bytes.

Code segment refers to a procedure, or to part of a procedure, that can
be embedded in your own programs to accomplish a given task. Although Program 6-4 has a title and an equate statement at the beginning, this program would never be used by itself, but rather would be used as a part of a larger program.

Program 6-4. Two-Byte Increment Example

```
; $06,07
=0006
6 PTR EQU $06 ; $06,07
7
008000: E6 06 8 INCR INC PTR ; ADD 1 TO LOW BYTE
008002: D0 02 =8006 9 BNE NEXT ; SKIP IF NOT = $00
008004: E6 07 10 INC PTR+1 ; ADD 1 TO HIGH BYTE
11
008006: EA 12 NEXT NOP ; YOUR PROGRAM
CONTINUES HERE...
13
```

-- End Merlin-16 assembly, 7 bytes, Errors: 0

In Program 6-4, a two-byte pair labeled PTR (PoinTeR) is defined. We'll assume that the two bytes contain the value for an address, with the low byte for the address in location $06, PTR. The high byte is stored in location $07. We could have used labels like PTRL and PTRH, for Pointer Low and Pointer High for locations $06 and $07. But a better, more commonly used system is to use the labels PTR and PTR+1. Most assemblers can handle mathematical expressions in the operand field, so PTR+1 will be properly calculated as $06+1 = $07.

The program works by first incrementing the low byte, PTR. If the result of the increment is not zero, no wraparound has occurred, and the flow of control goes to NEXT, where the body of your own program is found. If PTR is equal to $00, the BNE fails and PTR+1, the high byte is incremented.

The instruction at NEXT introduces a new, simple 65816 instruction, NOP. NOP stands for No Operation, and does just that—nothing. NOP is used in this example program so the assembly will be complete even though we don't know what might be placed at the NEXT statement.

NOP actually generates a byte of code, $EA. This can be useful when you want to deactivate a part of a program you're debugging, without having to do a new assembly. By putting NOPs in memory where you want to erase part of your program, the program will still execute and will ignore the NOPs when it gets to that part of the program. If you over-write part of your program with zeros instead, a BRK would occur when the first 0 was encountered.
If, at some point, you wish to just create a label in a program you’re writing, this can also be done, although the actual technique varies depending on what assembler you are using. In the Merlin assembler, you can just type a label and leave the opcode and operand fields blank. A comment is allowed. For example:

```
LABEL ; BEGINNING OF A SECTION
    LDA PTR
    ETC...
```

In the APW assembler, a specific APW directive, **ANOP** (Assembler **NOP**, not to be confused with the other NOP) must be used:

```
LABEL ANOP ; BEGINNING OF A SECTION
    LDA PTR
    ETC...
```

You might ask why you would want to do this. One of the most common reasons is as simple as convenience in editing. Suppose you are working on a part of your program, and it looks like this:

```
LABEL STA SCRN ; STORE CHARACTER ON SCREEN.
```

Suddenly realize you forgot to start the segment with `LDA #$C1`, or some other important instruction. Normally, you would have to erase the text `LABEL` from the beginning of the current line, and then insert a new line above it with the new `LABEL` in it, like this:

```
LABEL LDA #$C1
    STA SCRN
```

By making the beginning of the routine a label with no opcode, you can edit the lines below it at any time, adding or deleting instructions or data as you wish without having to retype the first line of the routine:

```
LABEL ; BEGINNING OF SECTION
    LDA #$C1
    STA SCRN
```

In the APW, the lines look like this:

```
LABEL ANOP ; BEGINNING OF A SECTION
    LDA #$C1
    STA SCRN
```

A label without an operand is also used at the beginning of a data storage section for similar reasons, namely easy adding and deleting of new entries. Data storage in an assembly source listing is discussed in Chapter 9.

Going back to the idea of incrementing a two-byte pointer, it has been
mentioned before that the 65816 has a two-byte mode where all register (A, X, and Y) operations can be made to take place two bytes at a time. In such a mode, any of the increment or decrement commands will correctly operate on both bytes in one instruction. However, this does not eliminate the need for the multiple byte techniques discussed here, because in such programs you’ll most likely be using four bytes (a long address) to store both the bank byte and address bytes of the address. Thus you’ll still need a similar multibyte increment or decrement on occasion. When operating in the two-byte mode, the following program segment would increment four bytes properly:

```
PTR EQU $06  ; $06,07,08,09
INCR INC PTR ; INCREMENT $06 AND $07
BNE NEXT ; IF NOT = $0000
INC PTR+2 ; $08,09
NEXT NOP ; YOUR PROGRAM HERE...
```

The 65816 knows whether to increment one or two bytes with the INC command, depending on the condition of a special flag bit, as is described in Chapter 7.

The pair of bytes that make up a two-byte address or half of a four-byte address are called data words. A word, in computer terminology, is the width in bytes or bits of data chunks as handled by the microprocessor. The 65816 has a bit of a split personality in this regard: It can use either 8-bit or 16-bit words. When dealing with a long address, the byte pairs are sometimes referred to as the low-order word and the high-order word, analogous to the low- and high-order bytes of a two-byte address. For the address $0006/1A00, $0006 is the high-order word, $1A00 is the low-order word.

There are other computers, like the Apple Macintosh, that can deal with 32 bits (4 bytes) at a time. Generally speaking, the greater the data width, the more powerful the machine—although this is a simplification that ignores clock speed (how fast each instruction is executed), the total number of instructions available to the microprocessor, overall system design, and other factors that contribute to the final net performance of a computer.

**Decrementing Two or More Bytes**

Decrementing a two-byte pointer is slightly different, only because we are looking for a transition from $00 to $FF (or $0000 to $FFFF). This is done using a compare instruction as part of the segment (see Program 6-5).
Loops and Counters

Program 6-5. Two-Byte Decrement Example

1 ****************************
2 * TWO-BYTE DECREMENT EXAMPLE *
3 * MERLIN ASSEMBLER        *
4 ****************************
5 =0006  6 PTR EQU $06 ; $06,07
6
7 008000: C6 06  8 INCR DEC PTR ; SUBTRACT FROM LOW BYTE
8 008002: A5 06  9 LDA PTR ; LOAD FOR CMP
9 008004: C9 FF 10 CMP #$FF ; WRAPAROUND?
10 008006: D0 02 =80A11 BNE NEXT ; NOPE.
11 008008: C6 07 12 DEC PTR+1 ; SUBTR. 1 FROM HIGH BYTE
12 13
14 0080A: EA 14 NEXT NOP ; YOUR PROGRAM CONTINUES HERE...

--End Merlin-16 assembly, 11 bytes, Errors: 0

We can't use this code segment:

DEC PTR ; SUBTRACT 1 FROM PTR
BNE NEXT
DEC PTR+1 ; SUBTRACT 1 FROM PTR+1

because PTR will reach 0 one cycle before we want to decrement PTR+1. Remember, the count-down will look like something like this:

Total Address: PTR PTR+1
$502 $02 $05
$501 $01 $05
$500 $00 $05
$4FF $FF $04

When PTR reaches $00, PTR+1 doesn’t change until the next cycle of our code segment. Therefore, the CMP #$FF is required to see when the DEC PTR has wrapped around to $FF, signifying it’s time to decrement PTR+1.

In the two-byte mode, the program segment would be similar:

PTR EQU $06 ; $06,07,08,09
DECR DEC PTR ; DECREMENT $06 AND $07
LDA PTR
CMP $FFFF
BNE NEXT ; IF NOT = $FFFF
DEC PTR+2 ; DECREMENT $08,09
NEXT NOP ; YOUR PROGRAM HERE...
Chapter 6

Long Address Jump Instruction: JML

Just as the JSR instruction had a long addressing counterpart in JSL, so the jump instruction has its counterpart, JML (Jump Long). Like the short form jump instruction, JML is a one-way trip (use JSL for a returning subroutine). The only difference between JMP and JML is that JML jumps to an absolute three-byte address in a given bank. JMP is limited to the current bank that the program is executing in.

In a program, JML would assemble like this (the address $06/1A00 is arbitrary):

```
= 0 61A00 1 LABEL EQU $061A00 ; $06/1A00
  2
00800: 5C 00 1A 06 3 JML LABEL

--End Merlin-16 assembly, 4 bytes, Errors: 0
```

Using the Monitor COUT Routine

The loop shown earlier lets you print text and numbers in a very limited way—wouldn’t it be nice to have a general text printing routine available? Such a routine already exists within the Monitor routines; it’s labeled COUT (pronounced C-OUT, for Character Output), and is located at $FDED. COUT is used by loading the accumulator with the value for the character you wish to print, and then calling the routine.

The neat part about using COUT is that you don’t have to write your own routines to handle screen control (what the line length is, when to scroll, and so forth). Better still, COUT can also be used to send characters to a disk file or printer, but more on that later.

We’re not done yet, though. We would like to have the counter value in the accumulator so we can print it via COUT, but unfortunately many built-in routines like COUT change the contents of the accumulator and other registers when they’re used. Thus, although the counter might be at 5 when we called COUT, the accumulator might hold 37 when control returned to our program.

To solve this, we’ll have to change the program. This time we’ll use a memory location as the counter, and then load the Accumulator on each pass through to print out a visible sign of the counter’s activity. Good locations to use for experimenting are locations $06 through $09. These are not used by Applesoft BASIC, DOS, or the Monitor. It’s important to avoid conflicts with Apple’s normal activities while running your own programs.

We’ll also go back to using BNE, since this requires fewer lines of source code, and is a better programming approach. In addition, a few blank lines
Loops and Counters

have been added to the listing to break up the logical groups of instructions for
easier reading (Program 6-6).

Program 6-6. Loop Demo Program 2B

```
1 *****************************************************
2  * LOOP PROGRAM #2B  *
3  * MERLIN 8/16 ASSEMBLER  *
4 *****************************************************
5
6   ORG $300
7
8 =0006  CTR   EQU $06
9 =FC58  HOME  EQU $FC58
10 =FDED  COUT  EQU $FDED
11
12 BEGIN JSR HOME ; CLEAR SCREEN
13 LDA #$FF ; START COUNTER TO 255
14 STA CTR ; STORE IN 'CTR'
15
16 LOOP LDA CTR ; GET CURRENT VALUE
17 JSR COUT ; PRINT CHARACTER
18 DEC CTR ; CTR = CTR - 1
19 BNE LOOP ; DONE IF CTR = 0
20
21 DONE RTS ; ALL DONE!
```

A CALL to this routine via our usual 300G from the Monitor, or a CALL
768 from BASIC, should clear the screen and then print all the available charac-
ters on your Apple—in all three display modes (normal, flashing, and in-
verse). We no longer need the WAIT routine because the characters will be
printed left to right, as opposed to all in the same spot on the screen. The beep
you hear is from when the Control-G (Bell) is printed to the screen via COUT.
The invisible control characters account for the blank region between the two
main segments of output characters.

The alphabet is backward because we started at the highest value and
worked our way down. The BNE test works because the loop will continue un-
til CTR reaches 0.

You'll remember, though, that when a byte is incremented by 1 from $FF,
the result also wraps around back to $00. This will produce an action also test-
able by a BNE. Using this wraparound effect of the increment command, we
can rewrite the program to be a little more conventional, counting in normal al-
phabetical order (Program 6-7).
Chapter 6

Program 6-7. Loop Demo Program 3

1 ******************************************
2 * LOOP PROGRAM #3 *
3 * MERLIN 8/16 ASSEMBLER *
4 ******************************************
5
6 ORG $300
7
8 CTR EQU $06
9 HOME EQU $FC58
10 COUT EQU $FDED

000300: 20 58 FC 12 START JSR HOME ; CLEAR SCREEN
000303: A9 00 13 LDA #$00 ; START COUNTER AT '0'
000305: 85 06 14 STA CTR ; STORE VALUE
000307: A5 06 15 LOOP LDA CTR ; GET CURRENT VALUE
000309: 20 ED FD 16 LDA CTR ; PRINT CHARACTER
00030C: E6 06 17 JSR COUT ; STORE VALUE
00030E: D0 F7 =0307 18 INC CTR ; CTR = CTR + 1
000310: 60 19 BNE LOOP ; GO AGAIN IF NOT

20

21 DONE RTS ; ALL DONE!

--End Merlin-16 assembly, 17 bytes, Errors: 0

A CALL to this routine should now print out the characters in a more familiar manner.

The ASCII System

The examples used so far have each put characters on the screen by using a number code to represent a given character. This coding system is based on the ASCII character set. ASCII (American Standard Code for Information Interchange) is a coding scheme for transmitting text. It’s also used in the Apple for encoding text in memory, screen display, disk files, printer output, and many other areas. The chart in Appendix E gives all the characters and their ASCII values.

You’ve probably already used the ASCII system if you’ve ever used the ASC( or CHR$( commands in Applesoft BASIC. A program line that says

10 D$ = CHR$(4)

assigns the string variable D$ to the ASCII character number 4, a Control-D. Likewise, the line

50 PRINT CHR$(9);"80N"

118
tells the program to print the character associated with the ASCII code 9, in this case a Control-I. Most Applesoft BASIC programs use the ASC( and CHR$( commands for creating control characters for printers, DOS, and so forth; but you can print any character, depending on the code you want. For instance, this line would print the letter A:

100 PRINT CHR$(65)

It turns out that it is possible to encode all the alphabetic characters (upper- and lowercase), numerics, special symbols, and control codes using only 128 number values. You can count to 128 using only the first seven bits of an eight-bit byte. This means that ASCII is considered a 7-bit code since all the information required to determine which character has been sent (or is stored in memory) is contained in bits 0 through 6 of the byte. This makes the condition of bit 7 (the eighth bit) to be somewhat irrelevant. Thus $8A (10001100) is reasonably equivalent to $0A (00001100) as far as its ASCII interpretation is concerned.

However, the matter of the high bit (bit 7) being set or clear can create considerable confusion when it's not the same as what the computer or output device (such as a printer) expects. This is because many hardware devices use the high bit to signal an alternate character set (such as inverse and flashing on the Apple screen) and block graphics characters on some printers.

Generally, the Applesoft BASIC assembly language routines and screen memory operate internally with the high bit set (set = on = 1) on all characters. That is to say, characters retrieved from the keyboard via $C000, and characters stored in the screen area of memory ($400 to $7F8), usually have the high bit set (values greater than $80). This is also the way Applesoft BASIC stores data within program lines. (To keep you on your toes though, strings within a program, such as A$ = "CAT", and ProDOS text files on disk, have the high bit clear.) When using COUT (the Monitor output routine), the high bit should be set (load the Accumulator with values greater than $80) before calling COUT or storing to the screen.

The ASCII system doesn't start with A as the first character (code number = 1). Instead, they assigned the first 32 values to what are called control characters. These were originally designed to be special codes to tell the mechanical teletype machines when to feed a new sheet of paper, when the transmission was complete, and other commands.

These codes are still used today, and your Apple IIGS keyboard has a key marked Control on it. When you hold down this key, it modifies the actual key pressed, much the same way a shift key changes a lowercase a into an uppercase A. For control characters, it creates the character Control-A. In printed
Chapter 6

listings such as those found in magazines and books, control characters are sometimes indicated with a caret (\(^\) symbol, like this:

\(^A = \) Control-A
\(^D = \) Control-D

It turns out then, that Control-A, is ASCII character 1, Control-B = 2, and so on. In addition, the ASCII character set can be arranged in groups to make things easier to keep track of:

<table>
<thead>
<tr>
<th>ASCII character codes (decimal):</th>
<th>ASCII character codes (hex):</th>
<th>Characters in Group:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 31</td>
<td>$00$ to $1F$</td>
<td>Control characters</td>
</tr>
<tr>
<td>32 to 63</td>
<td>$20$ to $3F$</td>
<td>Special characters</td>
</tr>
<tr>
<td>64 to 95</td>
<td>$40$ to $5F$</td>
<td>(%,$,0-9,&gt;,&lt;,\ldots)</td>
</tr>
<tr>
<td>96 to 127</td>
<td>$60$ to $7F$</td>
<td>Uppercase letters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(A,B,C,\ldots)</td>
</tr>
</tbody>
</table>

You can see that the groups seem more logical in their hex number groups rather than in their decimal groups. This is another example where being comfortable with hex number makes your life a little easier. You still don’t have to be able to convert a hex number to an ASCII character, but seeing the pattern helps keep everything straight in your head.

By the way, you might also notice that each group is made up of $20$ characters (32 decimal). This means that by adding or subtracting $20$ (32 decimal) from the ASCII value for an uppercase letter, you can print the equivalent control or lowercase letter.

The Applesoft BASIC lines

20 PRINT CHR$(65 + 32)
30 PRINT CHR$(65 - 32)

print a lowercase \(a\), and Control-A, respectively.

**Commands Learned So Far**

Here are the new commands you’ve learned, plus the ones covered in previous chapters.

In this chapter:

BEQ   BNE   BRA   BRL
JML   JMP   NOP
And in previous chapters:

<table>
<thead>
<tr>
<th>BRK</th>
<th>JSL</th>
<th>JSR</th>
<th>LDA</th>
<th>LDX</th>
<th>LDY</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL</td>
<td>RTS</td>
<td>STA</td>
<td>STX</td>
<td>STY</td>
<td>STZ</td>
</tr>
<tr>
<td>TAX</td>
<td>TAY</td>
<td>TXA</td>
<td>TXY</td>
<td>TYA</td>
<td>TYX</td>
</tr>
</tbody>
</table>

Secret #3

If you ran either of the LOOP programs, you may have noticed that your cursor changed to a new character after the programs were run.

The Apple II GS has a special "hidden" command to make the cursor anything you want.

For example, from Applesoft BASIC, type Control-Shift-6, then type an underline character (near the Delete key), followed by Return.

The cursor should change to a blinking underline character.

What’s actually going on is that whenever the computer sees a Control-A (ASCII code 30, decimal), it takes whatever the next character is and uses that for the cursor. You can do this from within a program with a line like this:

```
10 PRINT CHR$(30) ; + : REM MAKE CURSOR A “+”
```

Try this with different keyboard characters and see what cursors you can find for your own programs. To restore the cursor to normal, you can press Control-RESET, or set the cursor equal to the Delete key.
Chapter 7

Comparisons in Assembly Language
In the last chapter, you learned how to use instructions like BEQ and BNE to create simple loops. We used the X and Y registers as counters, and incremented or decremented by 1 for each cycle of the loop.

Now let’s expand our repertoire of instructions by adding some new ones, and in the process add some flexibility to what we can do with loops and tests in general.

In previous programs, we relied on the counters reaching zero and testing the Z-flag to take appropriate action. Suppose however, that you wish to test for a value other than zero. This is done using two new kinds of instructions, compare and branch-on-carry.

The compare instruction, with the mnemonic CMP, tells the computer to compare the contents of the Accumulator against some other value. The other value can be specified in a variety of ways. A simple test against a specific value would look like this:

```
CMP #$A0
```

This would be read "Compare Accumulator with an immediate $A0." This would tell the 65816 to compare the Accumulator with the specific value $A0. On the other hand, you may want to compare the Accumulator with the contents of a given memory location. This would be indicated by

```
CMP $A0
```

In this case, the 65816 would go to location $A0, see what was there, and compare that to the Accumulator. It’s important to understand that the contents of $A0 may be anything from $00 to $FF, and it is against the contents that the Accumulator will be compared. In each case, the comparison is done by subtracting the Accumulator from the specified value (although the result is invisible to the programmer).
Chapter 7

The other new instructions are Branch-on-carry. Branch-on-carry instructions, coincidentally enough, use the Carry flag. This is used to determine the result of the comparison. Right next to the Z-flag (zero flag) in the Status Register is the bit called the Carry (Figure 7-1).

Figure 7-1. The Carry Bit of the Status Register

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Z</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

This is used during addition and subtraction by the 65816. Because the compare operation involves subtraction, the Carry flag can also be used to test the result of a comparison. This is done with two branch instructions, BCC and BCS. BCC stands for Branch on Carry Clear. If the Accumulator is less than the value compared against, BCC will branch appropriately. BCS stands for Branch on Carry Set, and is taken whenever the Accumulator is equal to or greater than the value used.

This means you can not only test for specific values, but also test for ranges.

Program 7-1 is a variation on the LOOP program presented in Chapter 6, that only prints the letters A through Z.

This program is not designed for efficiency, but rather to demonstrate the principles of using BCC and BCS. The counter, CTR, is started at 0, as before. However, the LOOP part of the program has now been given some intelligence. It first compares the value of CTR with the ASCII value for the letter A, $C1. If CTR is less than $C1, program execution skips the use of COUT, and branches to L2, which increments CTR. Eventually, CTR reaches $C1, and printing begins. When CTR reaches the ASCII value of Z plus 1, $DB, BCS takes effect and the program ends. Remember that because BCS tests for greater than or equal—test for one more than the last value you want to print.

The X and Y registers can be compared in a similar manner by the codes CPX and CPY.

BEQ and BNE are also still usable after a compare operation. Here’s a summary:

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td>Compares Accumulator to something</td>
</tr>
<tr>
<td>CPX</td>
<td>Compares X register</td>
</tr>
</tbody>
</table>
Comparisons in Assembly Language

CPY     Compares Y register
BCC     Branch if register < value
BEQ     Branch if register = value
BNE     Branch if register <> value
BCS     Branch if register >= value
BRA     Branch always

Program 7-1. Loop Demo 4 Using CMP

```
I
••••••••••••••••••************************
2 * LOOP PROGRAM #4 *
3 * MERLIN 8/16 ASSEMBLER *
4 ******************************************
5
6 ORG $300
7
8 CTR EQU $06
9 HOME EQU $FC58
10 COUT EQU $FDED
11
12 START JSR HOME ; CLEAR SCREEN
13 LDA #$00 ; START COUNTER AT '0'
14 STA CTR ; STORE VALUE
15
16 LOOP LDA CTR ; GET CURRENT VALUE
17 CMP #$C1 ; "A" CHAR, HI BIT SET
18 BCC L2
19
20 CMP #$DB ; ASCII "Z" + 1
21 BCS DONE ; ALL DONE!
22
23 JSR COUT ; PRINT CHARACTER
24
25 INC CTR ; CTR = CTR + 1
26 BRA LOOP ; ALWAYS BRANCH
27
28 DONE RTS ; ALL DONE!

--End Merlin-16 assembly, 25 bytes, Errors: 0
```

Reading Data from the Keyboard
A good part of many formal machine language courses deal with just system I/O, that is, getting data In and Out via different devices. Writing such things as printer drivers, disk access routines, and hardware interface software are the
areas that hardcore programmers spend their youth mastering. Using the built-in routines on the Apple simplifies this greatly because you don't have to do most of the I/O details. You've already seen this is true by having used COUT ($FDED) for screen output, without having to know anything about how the actual operation is carried out. The keyboard is even easier.

In the memory map we've been using for the Applesoft BASIC, the address range from $C000 to $FFFF is devoted to hardware in that these memory ranges cannot be altered by running programs (we're ignoring the additional GS RAM for the time being). The range from $D000 to $FFFF is used by the ROM routines that we've been calling. The range from $C000 to $CFFF is assigned to I/O devices. Typically the second digit from the left gives the slot number of the device. For instance, if you have a printer in slot #1, a look at $C100 will reveal the machine language code in ROM on the card that makes it work. At $C600 you'll probably find the code that makes the disk drive in slot 6 boot.

$C000 to $C0FF is reserved not for slot 0, but for doing special things with the hardware portions of the Apple itself.

An attempt to disassemble from $C000 will not produce a recognizable listing, but it will probably cause your Apple to act a bit odd. This range is made up of a number of memory locations actually wired to physical parts of your Apple. If from the Monitor you type

* C030 <RETURN>

you'll see some random value displayed and the speaker should click. (If it doesn't click the first time, try again.) Each time you access $C030, the speaker will click as it moves in response to your action.

The keyboard is also tied into a specific location. By looking at the contents of $C000, you can tell if a key has been pressed. In BASIC, it's done with a PEEK ($-16384). In machine language, you would usually load a register with the contents of $C000, such as:

LDA $C000

Because it is difficult to read the keyboard at exactly the instant someone has pressed the key, the keyboard is designed to hold the value of the last key pressed until either another key is pressed or you clear something called the strobe, by accessing an alternate memory location, $C010. The strobe is wired to clear a character off the keyboard once a program has read the keypress.

It's always a good idea to clear the keyboard when you're done with it, otherwise you'll have the value for the key pressed for your input still hanging around for whatever reads the keyboard next, such as the next keyboard read in your own program, or an INPUT statement in BASIC. The strobe is cleared by any read or write operation. It's the mere access to it in any manner that ac-
complishes the clear. However, because of the way the microprocessor and hardware work, a read actually accesses the location twice.

Thus, although both a LDA $C010 or STA $C010 would clear the strobe, the LDA has the double disadvantage of ruining the contents of the accumulator, and clearing two characters off the keyboard if the keyboard buffer is turned on. This is very important to keep in mind. Although the LDA command will work (and some programs use this), it makes your program incompatible with keyboard buffering. If the user were to type HELLO THERE, your program would only see HLOTEE.

The last point to be aware of is that the keyboard is set up to tell you when a key is pressed by the value that is read at $C000. Now you might think that the logical way would be to keep 0 in $C000. Perhaps, but that’s not the way it’s done. Instead, we must add $80 the ASCII value of the key pressed. If a value less than $80 is at $C000, it means a key has not been pressed.

To illustrate all this, let’s look at some sample programs to read data from the keyboard. Look at Program 7-2.

Program 7-2. Keyboard Demo 1

```
1 ************************************************************************
2 * KEYBOARD PROGRAM #1A *
3 * MERLIN 8/16 ASSEMBLER *
4 ************************************************************************
5
6 ORG $300
7
8 =C000 8 KYBD EQU $C000
9 =FDed 9 COUT EQU $FDed
10 =FC58 10 HOME EQU $FC58
11
12 000300: 20 58 FC 13 START JSR HOME ; CLEAR SCREEN
14
15 000303: AD 00 C0 15 LOOP LDA KYBD ; READ KEYBOARD
16 000306: C9 80 16 CMP #$80 ; KEY PRESSED
17 000309: 90 F9 0303 17 BCC LOOP ; TRY AGAIN IF NOT...
18
19 00030A: C9 9B 19 CHECK CMP #$9B ; ‘ESCAPE’ KEY
20 00030C: F0 05 0313 20 BEQ DONE
21
22 00030E: 20 ED FD 22 PRINT JSR COUT ; PRINT ASCII CHARACTER
23 000311: 80 F0 0303 23 BRA LOOP ; DO IT AGAIN
24
25 000313: 60 25 DONE RTS
26
--End Merlin-16 assembly, 20 bytes, Errors: 0
```
Chapter 7

Line 15 loads the Accumulator from location $C000. This is then compared to $80, the minimum value for a key down. If the value read was less than $80, we go back for another look. Once a value greater than $80 is found, we can then check to see exactly what key was pressed.

Like any well-written program, this one avoids being in an infinite loop by allowing you to press the Escape key to stop it. The test specifically for the Escape key is on line 19.

All keys other than Escape are passed on to be printed on line 22, after which the program branches back to do it all over again.

You’ve probably noticed that the program runs on printing the same character until you press another key. That’s because the strobe is never cleared.

A better program is Program 7-3.

Program 7-3. Keyboard Demo 1B: A Better Way

```
1 ****************************************************
2 * KEYBOARD PROGRAM #1B *
3 * MERLIN 8/16 ASSEMBLER *
4 ****************************************************

5 ORG $300

6 =C000 8 KYBD EQU $C000
7 =C010 9 STROBE EQU $C010
8 =FDE0 10 COUT EQU $FDE0
9 =FC58 11 HOME EQU $FC58

10 START JSR HOME ; CLEAR SCREEN
11
12 000300: 20 58 FC 14 LOOP LDA KYBD ; READ KEYBOARD
13 000303: AD 00 C0 16 CMP #$80 ; KEY PRESSED
14 000306: C9 80 17 BCC LOOP ; TRY AGAIN IF NOT...
15 000308: 90 F9 =0303 18 LOOP
16
17 00030A: 8D 10 C0 20 CLEAR STA STROBE ; CLEAR CHARACTER
18
19 00030D: C9 9B 22 CHECK CMP #$9B ; ‘ESCAPE’ KEY
20 00030F: F0 05 =0316 23 BEQ DONE
21
22 000311: 20 ED FD 25 PRINT JSR COUT ; PRINT ASCII CHARACTER
23 000314: 80 ED =0303 26 BRA LOOP ; DO IT AGAIN
24 000316: 60 28 DONE RTS
25
26 ---End Merlin-16 assembly, 23 bytes, Errors: 0
```
Comparisons in Assembly Language

This should work better. Here the keyboard is cleared whenever a character is read and printed. Why not clear it right after the read on line 15? If we did it there, the only way we would see the character is if the user pressed the key in the time between when the strobe was automatically cleared and when the keyboard was checked again. This would create such a small time window that keypresses would very likely be missed. It is a much better technique to only clear the strobe after an actual keypress has been detected.

In trying out this program, you should type in enough text to wrap around onto the next line, and you should also try the arrow keys and Return. You may think all this performs as expected (with the exception of the missing cursor), but this should not be taken for granted. Without the screen management of COUT, you’d have to do quite a bit more programming to keep things straight. Once more, this is the advantage of using the routines already present in the computer, rather than have to worry about the details yourself.

The Emulation Bit

As mentioned earlier, there is a way to control the size of the Accumulator, X, and Y registers. This is done using the Carry, along with two other bits, in the Status Register.

On the first Apple II computers, before the Apple IIgs was a gleam in anybody’s eye, the Accumulator, X, and Y registers were limited to only a single byte. That’s why Applesoft BASIC is designed to only use one byte of these registers, and why when you call a machine language routine from Applesoft BASIC, the default register size is only a single byte.

In addition, the Status Register used bit 4 for a Break instruction flag (telling the system that a BRK instruction had been encountered), and bit 5 was not used at all.

When the 65816 was designed, they wanted to create a system that was compatible with the older machines, yet that was an improvement on the original design of the 6502 microprocessor used in the first Apples.

Figure 7-2. The Status Register

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>V</td>
<td>M</td>
<td>B</td>
<td>D</td>
<td>I</td>
<td>Z</td>
<td>E</td>
</tr>
<tr>
<td>Sign</td>
<td>Overflow</td>
<td>Accumulator Size Select</td>
<td>Break or X/Y Register Size Select</td>
<td>Decimal</td>
<td>Interrupt</td>
<td>Zero</td>
<td>Carry or Emulation</td>
</tr>
</tbody>
</table>

131
Chapter 7

The answer was to give the Carry bit a dual function (see Figure 7-2). By adding a new command, XCE (eXchange Carry with Emulation bit), the designers created a method for using two other bits in the Status Register for a new function. This is done by telling the 65816 to change the meaning of the Break flag to indicate the size of the X and Y registers (one or two bytes), and to use the previously unused bit 5 to indicate the size of the Accumulator. In the new state, bits 4 and 5 are labeled the m (Memory) and x (indeX) bits.

In the same way that XBA exchanged the current contents of the A part of the Accumulator with the B part, the XCE instruction swaps the current value of the Carry bit (0 or 1) with an invisible emulation bit. The emulation bit is an extra bit, in addition to the eight bits that make up the Status Register, but it can only be changed using the Carry flag and the XCE instruction.

When the Emulation bit is 1, the A, X, and Y registers are limited to a single byte in size, and their size cannot be immediately changed. The m and x bits have no real value at this point, because in the emulation mode they technically don't exist.

When the Emulation bit is 0 (native mode), the size of the Accumulator can be changed by changing the m register, and the size of the X and Y registers can be controlled using the x register. Note that a lowercase x is used to differentiate the x bit from the actual X register. The x bit controls the size of both the X and Y registers together; they cannot be independently set to different sizes.

Register Sizes:

Accumulator

- \( m = 0 \) 2 bytes (16 bits)
- \( m = 1 \) 1 byte (8 bits)

X and Y

- \( x = 0 \) 2 bytes (16 bits)
- \( x = 1 \) 1 byte (8 bits)

There is no underlying meaning to the 0 or the 1, as such. Rather they're just arbitrary flag values for the two modes. In general, 0 in the m or x flags indicates the full 16-bit mode.

You've seen the status of the Emulation bit shown, when the BRK instruction was used in debugging a program in Chapter 3:

```
00/0308: 00 00 BRK 00
A=00EF X=0000 Y=0000 S=01DD D=0000 P=$B0
B=00 K=00 M=0C Q=80 L=1 m=1 x=1 e=1
```

Along with the registers, the Processor Status Register, \( P = \$B0 \), is shown. You can also see the display of the Emulation bit; e is set to 1, meaning
that a BRK instruction will set bit 4 if encountered and that the size of the A, X, and Y registers are limited to one byte and cannot be changed. Because the e bit is a separate and invisible addition to the Status Register, its condition cannot be determined by looking at the value of P in the register display itself.

The register display also shows the condition of the m and x bits. This display is set up whenever a BRK instruction is encountered, and is used to help you see what the entire system status is at a given point in a running program.

**Setting and clearing the Carry.** To change the Emulation bit in a program, you must first condition (set to a 0 or 1) the Carry bit. There are two commands specifically for conditioning the Carry bit, **SEC (SEt Carry)** and **CLC (CLear Carry).** SEC sets the Carry bit to 1; CLC sets it to 0.

If you want set clear the Emulation bit to 1 (sometimes called the Native mode) so you can change register sizes, you must first put a zero in the Carry (CLC), and then exchange this with the Emulation bit (XCE):

```
CLC ; c = 0
XCE ; e = 0 = native mode
```

To set the Emulation bit back to 1 (called the Emulation mode), you must set the Carry to 1 (SEC), and then do the exhange:

```
SEC ; c = 1
XCE ; e = 1 = emulation mode
```

To change the m and x bits in the Status Register, there are two commands, **SEP (SEt Processor status bits)** and **REP (REset Processor status bits).** Notice that the designers were a bit inconsistent here using set/clear for one pair, and set/reset for the other.

In any event, REP and SEP are used to make any bit in the Status Register either 0 or 1. They look like this in a program:

```
SEP #10 ; %00010000 binary
REP #30 ; %00110000 binary
```

The comments show the operand number in binary form. In the SEP instruction, you set the bits in the operand to match whatever bits in the Status Register you want set to 1. In the REP instruction, the desired bits are also indicated by 1’s in the operand; however, the corresponding bits in the Status Register will be cleared to 0 by the instruction.

In the first line, SEP #10 sets bit 4, the m bit, to 1, thus setting the Accumulator to a 1-byte size (8 bits). In the second line, REP #30 clears both the m and x bits (bits 4 and 5), making both the Accumulator and index registers 2 bytes wide (16 bits).
Chapter 7

The most common use of the REP and SEP instructions is to clear or set the \( m \) and \( x \) bits to control the \( A, X, \) and \( Y \) register sizes. They're rarely used to change any of the other Status Register bits, although nothing prohibits it.

To demonstrate the effects of these instructions, enter Program 7-4.

Program 7-4. \( e-, m-, \) and \( x \)-Bit Demo

```
1 ******************************************
2 * e-, m- and x-bit demo             *
3 * MERLIN 16 ASSEMBLER              *
4 ******************************************
5
6 ORG $300
7
8 =FC58 HOME EQU $FC58
9
10 =05BC SCRN1 EQU $5BC ; SCREEN LOCATION #1
11 =063C SCRN2 EQU $63C ; SCREEN LOCATION #2
12
13 000300: 20 58 FC BEGIN JSR HOME ; CLEAR SCREEN
14
15 000303: A9 Cl LOA #"A" ; 1 BYTE LETTER "A"
16
17 000306: SD BC 05 STA SCRN1 ; PUT ONE BYTE ON SCREEN
18
19 000309: 18 18 CLC ; C = 0
20 00030A: FB 27 XCE ; E = 0 = NATIVE (16-BIT MODE)
21
22 00030C: A9 C1 00 LDA #"A" ; 2-BYTE LETTER "A"
23 00030F: 8D 3C 06 STA SCRN2 ; PUT TWO BYTES ON SCREEN
24
25 000312: 38 26 SEC ; C = 1
26 000315: FB 27 XCE ; E = 1 = EMULATION (8-BIT MODE)
27
28 000318: 60 30 DONE RTS
29
30 --End Merlin-16 assembly, 21 bytes, Errors: 0
31
After assembling, BLOAD and run the object file with a CALL 768.

The program starts off very much like the sample program in Chapter 3 with the mini-assembler. The ASCII value for the letter \( A \) is loaded into the Accumulator and then is stored at memory location $5BC, which makes it visible as a screen character.

Now, the exciting part. Line 18 sets the carry to zero, in preparation for
Comparisons in Assembly Language

the XCE that immediately follows. The XCE sets the Emulation bit to zero, which puts the 65816 into the native mode that enables the option of changing register sizes. Line 21 uses the REP #$20 instruction to clear bit 5, the \textit{m} bit, to zero. This puts the Accumulator in the 2-byte (16-bit) mode.

When the 65816 encounters the LDA instruction on line 23, it now \textit{loads two bytes} into the Accumulator. This has two immediate, and important effects. \textit{First, even though you think you're loading the value $\text{Cl}$ into the Accumulator, you need to be aware that the full 16-bit value, $\text{00Cl}$, is used. Second, because this value does take two bytes, the instruction is assembled as a total of three bytes, as opposed to only two when the 8-bit mode is used.}

The one- or two-byte status of a register also applies to any store operations associated with it. In this program, the STA SCRN2 instruction on line 24 puts \textit{two bytes} on the screen, for the complete value $\text{00Cl}$. Notice that the $\text{C1}$ (the low-order byte) is stored first at $\text{63C}$, followed byte $\text{00}$ (the high-order byte) at $\text{63D}$. This is visual proof of the reversed-order use of byte pairs by the 65816.

\textit{The final stage of our program is to return the computer to the 8-bit emulation mode. This is a requirement for any routine that you call from Applesoft BASIC that changes the register sizes or emulation bit. If you don't restore things properly, Applesoft BASIC itself will crash when you exit your routine.}

Line 26 sets the Carry bit ($C = 1$), so that the following XCE instruction will set the emulation bit to 1. We don't have to worry about resetting the register sizes, because setting the emulation bit to 1 automatically removes the \textit{m-} and the \textit{x-bit} functions, and makes all registers 1 byte wide.

To see how this program lists from the Monitor, go to the Monitor now, and list the program starting at $\text{300}$.

1=m 1=x 1=LBank (0/1)
00/0300: 20 58 FC JSR FC58
00/0303: A9 C1 LDA \#C1
00/0305: 8D BC 05 STA 05BC
00/0308: 18 CLC
00/0309: FB XCE
00/030A: C2 20 REP \#20
00/030C: A9 C1 LDA \#C1
00/030E: 00 8D BRK 8D
00/0310: 3C 06 38 BIT 3806,X
00/0313: FB XCE
00/0314: 60 RTS

You may be surprised to see that the disassembly doesn't look quite right after the REP \#20 instruction at $\text{30A}$. This is because the disassembler
doesn’t recognize the REP instruction as such, and it continues to disassemble the instructions in the 8-bit mode. This shows what would happen if the Emulation or the \textit{m} or \textit{x} bits were set incorrectly and you were to enter your program at $30A$—the program would not interpret the bytes there as you had expected.

To properly disassemble the bytes from $30A$, type in

\begin{verbatim}
0=m
\end{verbatim}

and press Return. Now type 309L to list

\begin{verbatim}
0=m 1=x 1=Lbank (0/1)
00/0300: 20 58 FC JSR FC58
00/0303: A9 C1 8D LDA #8DC1
00/0306: BC 05 18 LDY 1805,X
00/0309: FB XCE
00/030A: C2 20 REP #20
00/030C: A9 C1 00 LDA #00C1
00/030F: 8D 3C 06 STA 063C
00/0312: 38 SEC
00/0313: FB XCE
00/0314: 60 RTS
\end{verbatim}

Notice that the \textit{m} and \textit{x} indicators at the top of the listing shows that \textit{m} is now set to 0. As you examine different parts of memory, you must specifically set these settings, when they’re not conditioned by hitting a BRK in a program. The Monitor does not make any assumptions about the condition of the various registers. This is so you can examine any part of memory at will and set the registers as you think they will be when that part of your program is executing. You can examine the current register settings at any time in the Monitor by typing Control-E (Examine registers) and pressing Return.

Looking at the listing starting at $30C$, you can now see the LDA with the two-byte operand. The STA instruction still takes only one byte. The 65816 only needs to know the starting address of the byte pair, so the instruction lists the same in either mode. What changes is how many bytes are written to memory starting at $63C$.

The converse situation in listing also applies. Now that the \textit{m} flag is set to 0 (16-bit Accumulator mode), try listing starting at $300$ again.

\begin{verbatim}
0=m 1=x 1=Lbank (0/1)
00/0300: 20 58 FC JSR FC58
00/0303: A9 C1 8D LDA #8DC1
00/0306: BC 05 18 LDY 1805,X
00/0309: FB XCE
00/030A: C2 20 REP #20
00/030C: A9 C1 00 LDA #00C1
00/030F: 8D 3C 06 STA 063C
00/0312: 38 SEC
00/0313: FB XCE
00/0314: 60 RTS
\end{verbatim}
Comparisons in Assembly Language

Notice that now the instructions at $303 and $306 list improperly. These were assembled as 8-bit instructions. You are now disassembling in the 16-bit mode.

Try rewriting the program using the long (16-bit; \(x = 0\)) X and Y mode, instead of the long accumulator mode, and see how the listing changes. Be sure to run your new program to make sure that it works, and that it returns to the 8-bit mode for Applesoft BASIC properly. You may wish to experiment with different combinations of \(e\), \(m\), and \(x\) modes to get the feel for what your options are in any given program.

**Notes for the APW Assembler**

If you have the APW assembler, you'll need to use the directives `LONGA ON/OFF` and `LONGI ON/OFF` to tell the assembler you're changing the \(m\) and \(x\) modes, respectively. The Merlin assembler automatically changes its

Program 7-5. \(e\)-, \(m\)-, and \(x\)-Bit Demo for the APW Assembler

```assembly
--------------------------------------------------------------------------------
* e-, m- and x-bit demo  *
* APW ASSEMBLER  *
--------------------------------------------------------------------------------

KEEP EMX.DEMO
ORG $300
LONGA OFF ; STARTING CONDITION
MSB ON ; HI BIT = ON

MAIN START
HOME EQU $FC58
SCRN1 EQU $5BC ; SCREEN LOCATION #1
SCRN2 EQU $63C ; SCREEN LOCATION #2

BEGIN JSR HOME ; CLEAR SCREEN
LDA "A" ; 1 BYTE LETTER "A"
STA SCRIN1 ; PUT ONE BYTE ON SCREEN
CLC ; C = 0
XCE ; E = 0 = NATIVE (16 BIT MODE)
REP #$20 ; %00100000 = M = 0
LONGA ON
LDA "A" ; 2 BYTE LETTER "A"
STA SCRIN2 ; PUT TWO BYTES ON SCREEN
SEC ; C = 1
XCE ; E = 1 = EMULATION (8 BIT MODE)
* AUTOMATIC SEP #$30

DONE RTS
END
```

137
mode when it sees the REP and SEP instructions in the listing. The APW, however, requires that the programmer include these assembler directives in the listing. Program 7-5 is the example program in the APW format.

LONGA OFF is required at the beginning because Applesoft BASIC CALLs our routine with all registers set to 8 bits. Then, after the REP #$20 instruction, the appropriate LONGA ON directive is again required to tell the assembler we have switched modes.

MSB (Most Significant Bit) ON is also required at the beginning to tell the assembler to use the hi-bit ASCII (value larger than $80) form when assembling the LDA #$"A" instructions.

More on Switching Modes
For those who may have previously done some programming on the 6502 or 65C02, it's important to mention that changing the Emulation bit doesn't have a dramatic effect on how the 65816 operates. Virtually all of the same instructions are available in both modes. Because the terms emulation and native modes are used, there is a tendency to think that one mode must exclude many of the instructions of the other. This is not the case. There are a few instructions, like REP and SEP that are meaningless in a given mode, but they are just ignored if encountered. REP and SEP are meaningless in the 8-bit mode because the A, X and Y registers are already limited to 8 bits, and REP and SEP cannot change the m or x bits at the same time as the Emulation bit.

The main hazard to the programmer in switching modes is the change in the length of certain instructions such as the LDA and STX instructions. As you have seen from the sample disassemblies of our demonstration program, the same segment of object code can be interpreted by the 65816 in very different ways, depending on the status of the e, m, and x bits. If your program sets these flags to one condition, and then jumps to another part of the program that was written assuming a different set of conditions, very strange things are likely to happen, and it can be very difficult to track down the problem. When debugging programs that use mixed register sizes and modes, this should be one of the first things you check when debugging a program.

Commands Learned So Far
Here are the new commands you've learned, plus the ones covered in previous chapters:

In this chapter:

BCC  BCS  CLC  CMP  CPX
CPY  REP  SEC  SEP  XCE
And in previous chapters:

<table>
<thead>
<tr>
<th>BEQ</th>
<th>BNE</th>
<th>BRA</th>
<th>BRK</th>
<th>BRL</th>
<th>JML</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
<td>JSL</td>
<td>JSR</td>
<td>LDA</td>
<td>LDX</td>
<td>LDY</td>
</tr>
<tr>
<td>NOP</td>
<td>RTL</td>
<td>RTS</td>
<td>STA</td>
<td>STX</td>
<td>STY</td>
</tr>
<tr>
<td>STZ</td>
<td>TAX</td>
<td>TAY</td>
<td>TXA</td>
<td>TXY</td>
<td>TYA</td>
</tr>
<tr>
<td>TYX</td>
<td>XBA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 8

Simple Math
Chapter 8

Simple Math

This chapter will introduce the basic math operations of addition and subtraction in assembly language. To some extent, you've already seen how this is done. In Chapter 6, you saw how the increment and decrement commands could be used to add and subtract. Unfortunately it performs these functions only by one each time (VALUE + 1 or VALUE - 1).

If you're really ambitious, you could, with the commands you know already, add or subtract any number by using a loop of repetitive operations, but this would be a bit slow. Fortunately, a better method exists.

You'll recall that a byte is an individual memory location that can hold a value from $00 to $FF (0-255 decimal). This number comes about as a direct result of the way the computer is constructed, and the way in which you count in base two. In base two, each position of the byte is called a bit, and the positions are numbered from right to left, from 0 to 7.

The pattern for counting is similar to normal decimal or hex notation. The value is increased by adding 1 each time to the digit on the far right, carrying as it becomes necessary. In base 10, you have to carry every tenth count; in hex, every sixteenth. In base two, the carry is done every other time.

The first few numbers look like this:

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>$01</td>
<td>1</td>
<td>00000001</td>
</tr>
<tr>
<td>$02</td>
<td>2</td>
<td>00000010</td>
</tr>
<tr>
<td>$03</td>
<td>3</td>
<td>00000011</td>
</tr>
<tr>
<td>$04</td>
<td>4</td>
<td>00000100</td>
</tr>
</tbody>
</table>

Notice that in going from 1 to 2, we would add 1 to the 1 already in the first position (bit 0). This generates the carry to increment the second position (bit 1). Here is the end of the series:

$FD 253 111111101
$FE 254 111111110
$FF 255 111111111
Observe what happens when the upper limit of the counter is finally reached. At $FF (255) all positions are full. When the next increment is done, we should carry 1 to the next position to the left; unfortunately that position doesn't exist.

When you use an increment instruction, the byte value wraps around to $00, and the extra bit just created is ignored. Suppose we were incrementing a two-byte address pointer. When the low-order byte reached $FF, we would want the next increment to add 1 to the high-order byte. For example:

<table>
<thead>
<tr>
<th>Address Value</th>
<th>High-Order Byte</th>
<th>Low-Order Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>$01FF</td>
<td>$01</td>
<td>$FF</td>
</tr>
<tr>
<td>+ 1</td>
<td>$00</td>
<td>$01</td>
</tr>
<tr>
<td>$0200</td>
<td>$02</td>
<td>$00</td>
</tr>
</tbody>
</table>

If you were using the INC instruction to add one each time, however, you would lose the newly created bit from the increment at $FF. The way this is usually handled in a loop is to use the BNE or BEQ test, like this:

```assembly
PTR EQU $06 ; $06,07
LOOP INC PTR ; PTR = PTR + 1
   BNE NEXT ; LESS THAN $00 SO CONTINUE
   INC PTR+1
NEXT ??? ; REGULAR PROGRAM HERE...
```

This program segment will increment the two byte pointer PTR, PTR+1 by one each time. It works by testing for the specific condition in which PTR (the low-order byte) has just been incremented from $FF to $00. In all other cases, the BNE just branches out of the increment segment. When PTR is $00, 1 is added to PTR+1 (the high-order byte) to properly increment the address value.

In the 16-bit mode of the 65816, this problem is temporarily avoided by allowing this increment command to operate on both bytes of a two-byte pointer. With the Emulation and Memory bits clear (e = 0, m = 0), the following instruction works for a two-byte pointer:

```assembly
INC PTR ; INCREMENTS PTR,PTR+1
```

This instruction automatically takes care of any bits that need to be carried from the low-order to the high-order byte.

However, the solution is only temporary. Remember that the Apple II GS allows up to three bytes to be used to define an address (for example, $01/300 = Bank 1, location $300). What to do when the two-byte pointer reaches $FFFF and we need to increment a bank byte?
Add with Carry

The answer is to use some general-purpose instructions that will add and subtract any value—not just 1—and condition the carry bit if appropriate.

The addition instruction we’ll use is ADC (Add with Carry). When an addition is done with this command, and the result would generate a bit in a new position wider than the byte width indicated by \( m \), the carry is set. ADC can only be used to add to the value already in the Accumulator, so the \( x \)-bit (for \( X \) and \( Y \) register sizes) is not a direct concern. The source of the value to be added to the contents of the Accumulator is limited only by the addressing modes available to the ADC instruction itself, which, as it happens, are rather extensive.

Because ADC always adds the value of the carry bit to the calculated sum, it’s important to clear the carry bit before doing the actual addition. For example, consider this program segment:

```
LDA #$05
ADC #$07
STA RESULT
```

As it stands, there are two possible results. If the carry happened to be clear when this was executed, the value in RESULT would be $0C (12 decimal). If, however, the the carry had been set (perhaps as the result of some previous operation), then RESULT would have be $0D (13 decimal).

This problem is avoided using the CLC (Clear Carry) instruction introduced in Chapter 7.

```
CLC     ; CLEAR CARRY IN PREPARATION FOR ADDITION.
LDA #$05   ; LOAD ACCUMULATOR WITH VALUE “5”
ADC #$07   ; ADD “7” TO IT.
           ; RESULT NOW = 12 ($0C)
```

This segment adds 5 and 7 using immediate values. Logically, the CLC could have been done after the LDA #$05, but before the ADC #$07 itself. Although it need only precede the ADC command, it has no effect on the LDA, so it’s put at the beginning of the routine for aesthetic purposes. It also helps identify the overall unit as a math routine. Most program segments that use CLC and ADC are usually written as shown.

The contents of two memory locations can also be added using the following instructions:

```
CLC
LDA MEM   ; 1ST MEMORY LOCATION.
ADC MEM2  ; 2ND MEMORY LOCATION.
STA RSLT  ; STORE RESULT SOMEWHERE.
```
Chapter 8

When the result of an ADC instruction is greater than $FF$ (or $FFFF$ in the two-byte mode: $m = 0$), the Carry bit will be set to 1, and this will automatically be added to the result of the next addition or subtraction. For example, suppose you wanted to add $120$ to the base address $1F0$. You would start with the low-order bytes:

```plaintext
Starting Lo Byte: $F0  Carry = 0
ADC #$20  $20
Result: $10  Carry = 1
```

And then add the high-order bytes:

```plaintext
Current Hi Byte: $01  Carry = 1
ADC #$01  $01
Result: $03  Carry = 0
```

This gives the correct two-byte result of $0210$, and is equivalent to:

```
$1F0
+ $120
$310
```

Notice that the Carry is also automatically cleared if the result of the addition did not result in a new carry bit being generated.

The steps above are equivalent to this source code segment:

```
CLC
LDA #$<1F0 ; LOW-ORDER BYTE OF $1F0 = $F0
ADC #$<120 ; ADD LOW-ORDER BYTE OF $120 = $20
STA RSLT ; STORE LOW-ORDER RESULT = $10
LDA #$>1F0 ; HIGH-ORDER BYTE OF $1F0 = $01
ADC #$>120 ; ADD HIGH-ORDER BYTE OF $120 = $01
STA RSLT+1 ; STORE HIGH-ORDER BYTE OF RESULT = $03
```

In a two-byte mode, the operation is similar. Let’s assume a base address of $01/$FF00, to which we will add $00/0200:

```plaintext
Starting Low Word: $FF00  Carry = 0
ADC #$0200  $0200
Result: $0100  Carry = 1
```

Next addition:

```plaintext
Current High Word: $0001  Carry = 1
ADC #$0000  $0000
Result: $0002  Carry = 0
```

146
This gives the correct four-byte result of $02/0100, and is equivalent to:

\[
\begin{align*}
$0001/FF00 \\
+ $0000/0200 \\
= $0002/0100
\end{align*}
\]

In this example, you'll notice that all four hexadecimal digits of the bank address are shown. In the Monitor listing, however, only one byte is shown for the bank byte, because that is all that is maintained by the 65816 itself. If you look at the long address operand generated by the mini-assembler in Chapter 3, or the assembler output from Merlin or APW, you'll see it also is only three bytes.

Because it's not convenient to switch between 8- and 16-bit modes for the ADC and other operations, most long addresses on the Apple IIGS are loaded and stored as four bytes. This is more practical than trying to handle the minimal three bytes that are required. Thus, although the high byte of the high word is always zero, it's still carried around in calculations for efficiency's sake.

Program 8-1 and 8-2 are sample programs using the ADC instruction. Note the use of the CLC before each ADC.

Program 8-1. Math Demo 1

```
BEGIN
  EQU $06
  EQU $08
  EQU $0A

  CLC
  LDA N1
  ADC N2
  STA RSLT

DONE
```

--End Merlin-16 assembly, 8 bytes, Errors: 0
Chapter 8

Program 8-2. Math Demo 2

1 *******************************************************
2 * MATH DEMO PROGRAM #2       *
3 * MERLIN ASSEMBLER           *
4 *******************************************************
5
6         ORG $300
7
8 =0006   8 N1   EQU $06
9
10 =000A  10 RSLT  EQU $0A
11
000300: 18 12 BEGIN  CLC ; GET READY FOR ADDITION
000301: A5 06 13 LDA  N1 ; GET 1ST NUMBER
000303: 69 80 14 ADC  #$80 ; ADD #$80 TO ACC.
000305: 85 0A 15 STA  RSLT ; STORE RESULT
000307: 60 16 DONE  RTS ; ALL DONE

--End Merlin-16 assembly, 8 bytes, Errors: 0

Program 8-3. Math Demo 3A

1 *******************************************************
2 * MATH DEMO PROGRAM #3A       *
3 * MERLIN ASSEMBLER           *
4 *******************************************************
5
6         ORG $300
7
8 =0006   8 N1   EQU $06
9
10 =000A  10 RSLT  EQU $0A
11
000300: 18 12 BEGIN  CLC ; GET READY FOR ADDITION
000301: A5 06 13 LDA  N1 ; GET 1ST NUMBER, LO BYTE
000303: 69 80 14 ADC  N2 ; ADD 2ND NUMBER, LO BYTE
000305: 85 0A 15 STA  RSLT ; STORE RESULT, LO BYTE
000307: 69 08 16 LDA  N1+1 ; GET 1ST NUMBER, HI BYTE
000309: 69 81 18 ADC  N2+1 ; ADD 2ND NUMBER, HI BYTE
00030B: 85 0B 19 STA  RSLT+1 ; STORE RESULT, HI BYTE
00030D: 60 21 DONE  RTS ; ALL DONE

--End Merlin-16 assembly, 14 bytes, Errors: 0
In the first program, the value in N1 is added to the contents of N2 and stored in RSLT. In the second program, N1 is added to the immediate value #$80, and is also stored in RSLT. Note the CLC before the ADC to insure an accurate result. This routine could be used either as a subroutine in part of a larger assembly language program, or called from BASIC after passing the values to locations 6 and 8.

The main disadvantage to all these programs is that we're limited to one-byte values for both the original values and the result of the addition. Thus, if the result exceeds $FF, the carry is ignored.

The solution is to use the Carry bit to create a two-byte addition routine (see Program 8-3).

Notice that N1, N2 and RSLT are all two-byte numbers, with the second byte of each pair being used for the high-order byte. This allows us to use values and results from $00 to $FFFF (0–65535).

Once the two low-order bytes of N1 and N2 are added, and the partial result is stored, the high-order bytes are added. If an overflow was generated in the first addition, the Carry will be set and an extra unit will be added in the second addition. Note that the Carry remains unaffected during the LDA N1+1 operation.

You may want to BLOAD the object code for Program 8-3, and then call it from this BASIC program, Program 8-4.

Program 8-4. Math Demo 3A BASIC CALL

0 REM MACHINE ADDITION ROUTINE
10 HOME
20 INPUT "N1,N2?";N1,N2
30 N1 = ABS (N1); N2 = ABS (N2)
40 POKE 6,N1 + INT (N1 / 256) * 256: POKE 7, INT (N1 / 256)
50 POKE 8,N2 + INT (N2 / 256) * 256: POKE 9, INT (N2 / 256)
60 CALL 768
70 PRINT : PRINT "RESULT IS: "; PEEK (10) + 256 * PEEK (11)
80 PRINT : GOTO 20

The ABS( ) statements on line 30 of Program 8-4 eliminate values less than zero. Although there are conventions for handling negative numbers, this routine is not that sophisticated.

Many times the number being added to a base address is known to always be $FF or less, so only one byte for N2 is needed. A two/one addition routine is shown in Program 8-5.

Notice that if the carry is set, the value in N1+1 gets incremented by one, even though the ADC says an immediate $00. The $00 acts as a dummy value to allow the carry to do its job.
Chapter 8

Program 8-5. Math Demo 3B

```
1 ******************************************
2 * MATH DEMO PROGRAM #3B    *
3 * MERLIN ASSEMBLER         *
4 ******************************************
5
6 ORG $300
7
8 NI EQU $06
9 N2 EQU $08
10 RSLT EQU $0A
11
12 BEGIN CLC ; GET READY FOR ADDITION
13 LDA NI ; GET 1ST NUMBER, BYTE
14 ADC N2 ; ADD 2ND NUMBER, BYTE
15 STA RSLT ; STORE RESULT, BYTE
16
17 LDA N1+1 ; GET 1ST NUMBER, HIGH BYTE
18 ADC #$00 ; ADD CARRY ONLY (NO 2ND HIGH BYTE)
19 STA RSLT+1 ; STORE RESULT, HIGH BYTE
20
21 DONE RTS ; ALL DONE
```

--End Merlin-16 assembly, 14 bytes, Errors: 0

Subtraction in Assembly Language

Subtraction is done very much like addition, except that a borrow is required. Rather than using a separate borrow flag for this operation, the computer uses the opposite of the Carry as a borrow. That is, a set Carry flag will be treated by the subtract command as a clear borrow (or no borrow taken); a clear Carry as a set borrow (borrow unit taken).

The command for subtraction is SBC (Subtract with Carry). The borrow is cleared with the command SEC, for Set Carry. (Remember, things look backward here). Program 8-6 is the subtraction equivalent of a our two-byte addition program presented earlier.

The program can be called with a slight variation on the Applesoft BASIC program we used for the addition programs (Program 8-7).

This assembly language routine will work fine for subtracting one positive number from another, but how can we handle negative numbers? Negative numbers can be thought of as a way of handling certain common arithmetic possibilities, such as when subtracting a larger number from a smaller one, (for example, 3 - 5 = -2), and when adding a positive number to a negative number (such as 5 + -8 = -3) to obtain a given result.
To be successful, then, what we must come up with is a system that will be consistent with the arithmetic of signed numbers as you now know it.

Program 8-6. Subtraction Example 1

```
ORG $300

; SET CARRY = 'CLEAR BORROW'
LDA N1 ; GET LOW BYTE OF 1ST VALUE
SBC N2 ; SUBTRACT LOW BYTE OF 2ND VALUE
STA RSLT ; PUT IN LOW BYTE OF 'RSLT'

; GET HIGH BYTE OF 1ST VALUE
LDA N1+1
SBC N2+1 ; SUBTRACT N2 WITH BORROW IF NEEDED
STA RSLT+1 ; PUT IN HIGH BYTE OF 'RSLT'

DONE RTS ; DONE!
```

--End Merlin-16 assembly, 14 bytes, Errors: 0

Program 8-7. Subtraction Example BASIC CALL

```
REM MACHINE SUBTRACTION ROUTINE

HOME
INPUT "N1,N2?";N1,N2
N1 = ABS(N1); N2 = ABS(N2); REM NO NEG. NUMBERS YET
IF N2 > N1 PRINT "WE CAN'T DO THAT YET!";END
POKE 6,N1 - INT (N1 / 256) * 256: POKE 7, INT (N1 / 256)
POKE 8,N2 - INT (N2 / 256) * 256: POKE 9, INT (N2 / 256)
CALL 768
PRINT : PRINT "RESULT IS: "; PEEK (10) + 256 * PEEK (11)
PRINT : GOTO 20
```

The Sign Bit

A good first approach to the problem is to just arbitrarily decide to use one of the 8 bits in a byte as a flag to indicate whether the number is positive or negative. If the bit is clear, the number will be positive. If the bit is set, the number will be regarded as negative. We'll use bit 7 (the eighth bit) for this. Thus +5 would be represented...
Chapter 8

00000101
While $-5$ would be shown as
10000101

Note that by sacrificing bit 7 to show the sign, we’re now limited to values from $-128$ to $+127$. When using two bytes to represent a number, such as an address, this means we’ll be limited to a range of $-32768$ to $+32767$. Sound familiar? If you’ve ever noticed the limits for integer variables in Applesoft BASIC, you’ll recognize this as the same range.

Although this new scheme is very pleasing in terms of simplicity, it does have one minor drawback—it doesn’t work. If we attempt to add a positive and a negative number using this result, we get disturbing results:

\[
\begin{align*}
+5 & : 00000101 \\
+(-8) & : 10001000 \\
-3 & : 10001101 = -13
\end{align*}
\]

Although we should get $-3$ as the result, using our signed bit system we get $-13$. There must be a better way. Well, with the help of what looks a lot like numeric magic, we can get something that works, although some of the conceptual simplicity gets lost in the process.

What we’ll invoke for this magic is the idea of number complements. You’ve probably heard of complementary angles (when dealing with two angles that add up to 180 degrees). In binary math, the simplest complement is called a ones complement. The ones complement of a number is obtained by reversing each 1 and 0 throughout the original binary number. Let’s try this as a negative number.

For example, the ones complement to 5 is

\[
\begin{align*}
00000101 & = +5 \\
11111010 & = -5
\end{align*}
\]

For 8, it is

\[
\begin{align*}
00001000 & = +8 \\
11110111 & = -8
\end{align*}
\]

This process is essentially one of definition, that is to say that we declare to the world that 1110111 will now represent $-8$ without specifically trying to justify it. Undoubtedly there are lovely mathematical proofs of such things, presenting marvelous ways of spending an afternoon; but, for our purpose, a general notion of the system will be sufficient. Fortunately, computers are very good at following arbitrary numbering schemes.
Now let’s see if we’re any closer to a working system.

\[
\begin{array}{c}
+5 & 00000101 \\
+ & -8 & 11110111 \\
\hline
-3 & 11111100 = -3 \\
(00000011 = +3)
\end{array}
\]

That worked. Let’s try another:

\[
\begin{array}{c}
-5 & 11111010 \\
+ & +8 & 00001000 \\
\hline
3 & 00000010 = 2 (plus Carry)
\end{array}
\]

Well, we seem to be closer. At least our answers will be right half the time. Don’t despair, there is a final solution, and that is to use what is called the twos complement system. The only difference between this and the ones complement system is that, after deriving the negative number by reversing each bit of its corresponding positive number, we add one.

Let’s see how it looks.

**For -5:**

5 = 00000101

**ones complement...**

11111010

**now add one...**

-5 = 11111011

**For -8:**

8 = 00001000

11110111

Now, let’s try the two earlier operations.

\[
\begin{array}{c}
+5 & 00000101 \\
+ & -8 & 11111000 \\
\hline
-3 & 11111101 = -3 \\
+8 & 000001000 \\
\hline
& 3 & 00000011 (plus Carry)
\end{array}
\]

Does 1111101 equal -3?

starting number: 00000011 = 3

ones complement: 11110110

add 1: +1

twos complement: 11111101 = -3

It works in both cases. It turns out that twos complement works in all cases. Most of the time, you probably won’t have much need for negative numbers, but hopefully you’ve at least gained a little insight to why integer variables are limited to the size that they are, and if you ever do have to deal with negative numbers, you’ll be prepared.

The best thing about this lesson, however, is that we can now use the
term sign bit. In the Status Register, a flag is provided for the easy testing of bit 7, and this flag is usually called the sign flag. Whenever a byte is loaded into a register, or any arithmetic operation is done, the sign flag will be conditioned according to what the final state of bit 7 (the sign bit) is. For example, LDA #$80 will set the sign flag to 1 (set), whereas a LDA #$40 will clear the flag. This is tested using the commands BPL (Branch on PLus) and BMI (Branch on MInus).

The Sign Bit as a Flag

Regardless of whether you’re using signed numbers or not, these instructions can be very useful for testing bit 7 of a byte. Many times, bit 7 is used in various parts of the Apple to indicate the status of something. For example, the keyboard location, $C000, sets the sign bit (we’ve also been calling it the high bit) whenever a key is pressed. Up until now, we’ve always tested by comparing the value returned from $C000 to the value #$80, such as in this partial listing:

```
LOOP LDA KYBD ; READ KEYBOARD
CMP #$80 ; KEY PRESSED?
BCC LOOP ; TRY AGAIN IF NOT . . .
```

This program will stay in a loop until a key is pressed. The keypress is detected by the value returned in $C000 being equal or greater to $80. A more elegant method is to use the BPL command as shown in Program 8-8.

Program 8-8. Keytest

```
1 ********************************************
2 * KEYTEST PROGRAM #2 *               
3 * MERLIN ASSEMBLER *                
4 ********************************************
5 ORG $300                           
6                                 
7 =C000  8 KYBD EQU $C000           
8 =C010  9 STROBE EQU $C010         
9                                 
10 000300: AD 00 C0 11 CHECK LDA KYBD ; GET VALUE FROM KYBD
10 000303: 10 FB =0300 12 BPL CHECK ; NO KEYPRESS, TRY AGAIN
13 000305: 8D 10 C0 14 CLR STA STROBE ; CLEAR KEYBOARD
14 000308: 60 15 D V 16 DONE RTS
15
```

--End Merlin-16 assembly, 9 bytes, Errors: 0
In this case, as long as the high bit stays clear (no keypress), the BPL will be taken and the loop continued. As soon as a key is pressed, bit 7 will be set to one, and the BPL will fail. The strobe is then cleared and the return done.

The Open Apple ($C061) and Option ($C062) keys (equivalent to push-buttons on joysticks) work in a similar way. If bit 7 of the corresponding memory locations are set, the button is being pushed. Program 8-9 shows an example.

Program 8-9. Button Test

```
***************
* BUTTON TEST  *
* MERLIN ASSEMBLER *
***************

ORG $300
=C061 PB0 EQU $C061 ; PUSH-BUTTON 0 OR OPEN APPLE KEY

000300: AD 61 C0
000303: 10 FB =0300

CHECK LDA PB0 ; GET STATUS BYTE
BPL CHECK ; AGAIN IF NO BUTTON PUSH

000305: 60 DONE RTS

--End Merlin-16 assembly, 6 bytes, Errors: 0
```

A variation on Program 8-9 is to check for the high bit set with the BMI instruction:

```
CHECK LDA PB0 ; GET STATUS BYTE
BMI DONE ; BRANCH IF PUSHED
BRA CHECK ; BPL WOULD WORK TOO...
DONE RTS
```

**Short Loops with BMI and BPL**

BPL and BMI are also used to terminate a loop of less than 128 cycles that must end when the Y register passes 0. For example:

```
ENTRY LDY #$50 ; STARTING VALUE FOR THE LOOP
LOOP DEY ; Y = Y - 1
BPL LOOP ; AS LONG AS Y IS POSITIVE (for example, < $80)
DONE RTS
```

or:

```
ENTRY LDY #$A0 ; STARTING VALUE FOR THE LOOP
LOOP INY ; Y = Y + 1
BMI LOOP ; AS LONG AS Y IS NEGATIVE (for example, > $7F)
DONE RTS
```

155
Chapter 8

In the first example, the Y register will wrap from $00 to $FF when it passes zero, causing the BPL to fail, thus terminating the loop. In the second example, the wrap is from $FF to $0, and BMI is used. The main drawback to this approach is that your loop counter must always be either positive or negative until the critical point. In other words, you couldn’t start at $FF and count down to zero because $FF is already negative, so the BPL wouldn’t work to keep the loop going.
Chapter 9

Logical and Shift Operators
Chapter 9

Logical and Shift Operators

In this chapter you'll see how to use two important types of commands: shift operators and logical operators. Shift operators are somewhat easier to understand, so we'll start with them.

Shift Operators: ASL

In the last chapter, you saw how to do simple addition and subtraction. This was done with the ADC and SBC instructions. Although the 65816 doesn't specifically have a multiply and divide instruction, there are instructions that come close and can be used to build an actual multiply or divide routine.

The shift commands give you the option of shifting each bit in the Accumulator or a given memory location one position to the left or right. The first two shift commands we'll look at are ASL (Arithmetic Shift Left) and LSR (Logical Shift Right).

ASL
(Arithmetic Shift Left)

In the case of ASL, each bit is moved to the left one position, with bit 7 going into the Carry, and bit 0 being forced to a zero. In addition to the Carry, the Sign and Zero Flags are also affected, depending on the resulting condition of bit 6 and whether the entire result is zero or not. Here are some examples showing the result of doing an ASL on given values, and the resulting status of the Carry, Sign, and Zero flags. The binary representation of each number has been spaced in the middle for easier reading.
ASL

<table>
<thead>
<tr>
<th>Value</th>
<th>Result</th>
<th>(C)</th>
<th>(N)</th>
<th>(Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td>Binary</td>
<td>Hex</td>
<td>Binary</td>
<td>Carry</td>
</tr>
<tr>
<td>$00</td>
<td>0000 0000</td>
<td>$00</td>
<td>0000 0000</td>
<td>0</td>
</tr>
<tr>
<td>$01</td>
<td>0000 0001</td>
<td>$02</td>
<td>0000 0010</td>
<td>0</td>
</tr>
<tr>
<td>$80</td>
<td>1000 0000</td>
<td>$00</td>
<td>0000 0000</td>
<td>1</td>
</tr>
<tr>
<td>$81</td>
<td>1000 0001</td>
<td>$02</td>
<td>0000 0010</td>
<td>1</td>
</tr>
<tr>
<td>$FF</td>
<td>1111 1111</td>
<td>$FE</td>
<td>1111 1110</td>
<td>1</td>
</tr>
</tbody>
</table>

In the first case, there is no net change to the Accumulator (or shifted memory location), although the Carry and Sign flags are cleared and the Zero flag is set (BCC, BPL and BEQ would work). The zero at each bit position was replaced by a zero to its right.

However, in the case of $01, the value doubles to become $02 as the 1 in bit 0 moves to the bit 1 position. In this case, all three flags are cleared.

When the starting value is $80 or greater, the Carry will be set. In the case of $80 itself, the value returns to 0 after the shift, since the only 1 in the pattern, bit 7, is pushed out into the Carry.

Notice that in the case of $FF, the Sign flag gets set as bit 6 moves into position 7. Remember that bit 7 is also called the Sign bit.

ASL has the effect of doubling the byte being operated on. This can be used as an easy way to multiply by any power of 2. By using multiple ASLs, you can multiply by 2, 4, 8, 16, and so on, depending on how many ASLs you use.

LDA $05 ; STARTING VALUE = 5
ASL ; ACC = 10 = 5 * 2
ASL ; ACC = 20 = 5 * 4
ASL ; ACC = 40 = 5 * 8

The examples assume that the m flag is in the 8-bit mode. If e and m = 0 (16-bit memory mode), then two bytes will be shifted at a time:

In the 16-bit mode, a zero is put in bit 0 of the low-order byte, bit 7 of the low-order byte is put in bit 0 of the high-order byte, and bit 7 of the high-order byte is put in the Carry. This still has the effect of doubling the two-byte value.
Logical Shift Right: LSR
The complement of the ASL command is LSR (Logical Shift Right). It behaves identically, except that the bits all shift to the right.

This can be used to divide by multiples of 2. It's also a nice way to test whether a number is even or odd: Even numbers always have bit 0 clear; odd numbers will always have it set. By doing an LSR followed by a BCC or BCS, you can test for this. LSR also conditions the Sign and Zero flags.

In both LSR and ASL, one end or the other always gets forced to a zero. Sometimes this is not desirable. The rotate commands—ROL and ROR (ROtate Left and ROtate Right)—are the solution to this.

With these commands, the Carry not only receives the pushed bit, but its previous contents are used to load the now-available end position.

ROL and ROR are used rather infrequently, but they do turn up occasionally in math functions such as multiply and divide routines.

Addressing modes for the shift operations include the absolute modes and indexed modes using the X register (with the exception of (MEM,X). The Y register cannot be used as an index in any of the shift operations.

Logical Operators
If you have ever used a statements like these:

IF X>5 AND Y=1 THEN 100
IF X=5 OR A$="WORD" THEN PRINT "HELLO"
you have already used an Applesoft BASIC version of what are called logical, or Boolean, operators. In Applesoft BASIC, logical operators are used for testing a group of conditions. In assembly language, the intuitive ideas of AND and OR are extended to produce a specific mathematical function that has uses beyond group conditional instructions.

Let's start with one of the most commonly used commands, AND. You're already familiar with the basic idea of this one from your daily speech. If this and that are a certain way, then I'll do something. This same way of thinking can be applied to your computer.

Let's consider a case where there are two things to be compared, A and B, and each can be a true or false. This will correspond to the statement "If A and B, then do something."

It's possible to draw a simple chart that illustrates all the possibilities.

**AND**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>No</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The chart shows four possibilities. If A is false, and B is false, then A AND B obviously aren't true, so we get a No answer. Likewise, if only one of either A or B is true, but not the other, then the result is still No. Only when both are true is the result Yes.

These results can also be written out as

- False AND False = No
- False AND True = No
- True AND False = No
- True AND True = Yes

For our purposes it would be more useful to rewrite the chart and sentences using 1 for True and Yes, and 0 for False and No.
Logical and Shift Operators

AND

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This table can be used to define a new mathematical function, AND. A mathematical function is just a set of rules for determining what numbers should result (the output) whenever a defined operation (function) is done on a given starting group of numbers (the input).

Many years ago, you learned four fundamental mathematical functions—multiply, divide, add, and subtract—and to do so you memorized a set of rules (or instant-recall answers) for each of the functions.

The Function AND

AND is a mathematical function also. In fact, the idea that mathematical functions could simulate logic was quite the rage in Lewis Carroll’s (of Alice in Wonderland fame) time, and a fellow named George Boole did quite a bit of work in the area. It’s Boole’s name that’s been given to this topic of ANDs and ORs—which is called Boolean math.

It’s actually quite an interesting subject. In the same way that multiplication and division have real-world examples with miles-per-hour and how many yards of cloth are needed for a dress, Boolean functions can be applied to areas outside BASIC programs and the logic of true and false. For example, consider this setup of a battery, a light, and two switches.

We’ll put the two switches in line with one another between the battery and the light:
In this setup, the light will only come on when both switch A AND B are on. You can quickly see that the physical analogy of switches and lights is identical to the mathematical function of AND. It also should give you a clue as to why AND is a very easy function to implement on a computer.

Once you’ve created a rule for dealing with 1’s and 0’s, you can apply those rules to the bits in a binary number.

Does 5 AND 3 have meaning? It turns out that it does, although the answer will not be 8. As we look at these numbers on a binary level, how to get the result of 5 AND 3 will be more obvious:

\[
\begin{align*}
A &= \text{5} \quad \text{0 1 0 1} \\
B &= \text{3} \quad \text{0 0 1 1} \\
A \text{ AND } B &= \text{0 0 0 1} = \text{1}
\end{align*}
\]

If you use the chart created earlier and apply it to each set of matching bits in A and B, you will obtain the result shown. Starting on the left, two 0’s in bit-position 3 (the fourth bit from the right) give zero as a result. For the next two bits, only a single 1 is present, in each case still giving zero as a result. Only in the last position do you get the necessary 1’s in both number’s 0-bits to yield one as the result.

Thus 5 AND 3 does have meaning, and the answer is 1.

AND is used for a variety of purposes. These include:

- To force 0’s in certain bit positions.
- As a mask to let only 1’s in certain positions “through.”

**Using ANDed in a Program**

When an AND operation is done, the contents of the Accumulator are ANDed with another specified value. The result of this operation is then put back in the Accumulator. The other value may either be given by way of the immediate mode, or held in a memory location. Here are some of the possible addressing modes for AND:

- LDA #$80
- AND #$7F
- AND $06
- AND $300,X
- AND ($06),Y

To better understand how AND is used, let’s consider this scenario:

Suppose you have a program that gets a key from the keyboard, and then checks to see if it’s a certain command. Let’s suppose for a moment that your looking for the letter A in an input command. Your program would probably look something like this:
Logical and Shift Operators

GETKEY LDA KYBD ; CHECK KEYBOARD
BPL GETKEY ; NO KEY YET
STA STROBE ; KEY GOT - CLEAR KYBD

CHECK CMP #"A" ; $Cl = "A"
BEQ ROUTINE ; THAT'S FOR US!
BNE GETKEY ; TRY AGAIN...

This seems to be a good start, but what about when the user types the A key with the CAPS LOCK key up (lowercase a)? You could check for both keys:

CHECK CMP #"A" ; $Cl = "A"
BEQ ROUTINE ; THAT'S FOR US!
CMP #"a" ; $E1 = "a"
BEQ ROUTINE ; THAT'S FOR US TOO!
BNE GETKEY ; TRY AGAIN...

This would do the job for this particular key, but what if your program has 50 commands? Do you want to do a double-check for each key? Probably not. It would be nice if there was a way to turn all lowercase input characters to uppercase in one step, before all the testing was done. The answer is to use the AND command. To see how this works, look at the binary values for the letters A and a:

A = $C1 = 1100 0001
a = $E1 = 1110 0001

^ bit 5

Notice that the only difference is that a lowercase a has bit 5 set to one. If we could clear just this bit, the value would correspond to an uppercase A.

The way to do this is with the AND command, and to create what is called a mask. A mask, in engineering and art terms, is something that only lets certain parts of an image through. If we treat the bit pattern for the letter a as an image, what we want is a mask that will not let the one in bit 5 through. That can be accomplished like this:

LDA VALUE ; $E1 = "a"
AND #$DF ; $DF = %1101 1111
; Result = "A"

Here's the process illustrated:

a = $E1 = 1110 0001
AND #$DF = 1101 1111
A = $C1 = 1100 0001
Notice that at each position, the binary version of $DF has a one (except bit 5). This means that ones and zeros in the input value, $E1, come through unchanged. However, the zero at bit 5 in the mask value, $DF, forces a zero in the output value at the same position.

AND is almost always used to force zeros in a given position, and to let the other data in the pattern through unchanged. The Merlin assembler lets you use binary numbers as an operand by putting a percent symbol ( % ) in front of the number, like this:

```
LOA VALUE
AND #%11011111 ; $DF
; Result = “A”
```

This makes it easier to see exactly what mask you’re using.

In general, AND is used to force zeros in a value. This is done using a mask with all bits set to 1 except for those which you wish to force to 0:

```
LDA VALUE ; $E1 = “a”
AND #%11011111 ; $DF
; Result = “A”
```

Clear Bits and Words

There is also an instruction specifically for clearing bits in a byte or word in memory, called TRB (Test and Reset Bits). This probably should have been named TCB for Test and Clear Bits, but those engineers like to keep you guessing. The T for test is a bit redundant too. It just means that it uses the bits in the Accumulator just like the AND instruction does.

TRB acts just like AND, except that it will only clear one or two bytes in memory. The result is not left in the Accumulator, although the mask is required to be in the Accumulator for its use.

Here’s a program segment that converts a lowercase a to A in a memory location:

```
LOA #%11011111 ; $DF = MASK
TRB MEM ; CONVERT MEMORY LOC. VALUE
```

BIT

The command somewhat related to AND is BIT. This is provided to allow a program to determine the condition of specific bits in a byte or word. It also has some secondary uses. When BIT is executed, quite a number of things happen. First, bits 6 and 7 of the memory location referenced are transferred directly to the Sign and Overflow bits. Since we’ve not discussed the Overflow
flag, let me say briefly that it is bit 6 in the Status Register, and has two associated branch instructions **BVC (Branch on Overflow Clear)** and **BVS (Branch on Overflow Set)**. After a **BIT** instruction, **BVC**, **BVS**, **BPL**, and **BMI** may be used to test the status of bits 6 and 7 in the referenced memory location. For example:

```
BIT MEM ; TEST A MEMORY LOCATION
BPL ROUTINE1 ; BRANCH IF BIT 7 = 0
BMI ROUTINE2 ; BRANCH IF BIT 7 = 1
BVC ROUTINE3 ; BRANCH IF BIT 6 = 0
BVS ROUTINE4 ; BRANCH IF BIT 6 = 1
```

The most frequent use of the **BIT** instruction is to either test a memory location, like the keyboard byte, $C000 without affecting the contents of the Accumulator, or to access a softswitch, like the keyboard strobe, again without affecting the contents of the Accumulator. This program illustrates both:

```
LDA #$05 ; CHARACTER "X"
KYBD BIT $C000 ; WAIT FOR KEYPRESS
BPL KYBD ; NONE YET.
BIT $C010 ; CLEAR KEYBOARD STROBE
JSR COUT ; PRINT "X" NO MATTER WHAT KEY PRESSED.
```

The other use of **BIT** is to test to see if one or more bits in the memory location match bits set in the Accumulator. If one or *more* do, the Zero flag will be cleared ($Z=0$). If no match is made, the Zero flag will be set ($Z=1$). This is done by ANDing the Accumulator and the memory location, and conditioning the zero flag depending on the result. The confusing part is that the test (BNE for a match) may seem backward. Alas, it’s unavoidable—it’s just one of those notes to scribble in your book so as to remember the quirk each time you use it.

Here’s an example to test for bits 0 or 2 set:

```
LDA #$05 ; 0000 0101
BIT MEM
BNE MATCH ; ONE OR BOTH BITS MATCH
BEQ NOMATCH ; NEITHER BIT IN MEM IS "ON"
```

**BIT** is usually used to test for a single bit being on. If you want to test for *all* of a group of bits being on, the **AND** instruction can be combined with a compare.

To test for both bits 6 and 7 being on:

```
LDA MEM
AND #$C0 ; 1100 0000 = BITS 6 & 7
CMP #$C0 ; COMPARE TO SAME VALUE
BEQ MATCH ; ONLY IF "BOTH" BITS ON
BNE NOMATCH ; IF BOTH NOT ON
```
Chapter 9

This last example is somewhat subtle, in that the result in the Accumulator after the AND will only equal the value with which it was ANDed if each bit set to one in the test value (the AND operand) has an equivalent bit on in the Accumulator (loaded from the memory location).

Take note that TRB and TSB (discussed below) both condition the Zero flag in the same way as the BIT instruction, and so could be used for a test, but they also re-write the tested bits in memory.

ORA and EOR
These two commands bring up an interesting error of sorts in the English language, and that is the difference between and inclusive OR and the exclusive OR. When you say “I’ll go to the store if it stops raining OR a bus comes by,” it has two possible interpretations. The first is that if either event happens, then the result will happen. This also includes the possibility that both may happen. This is called an inclusive OR type statement.

The other possibility is that the conditions to be met must be one or the other but not both. This might be called the most pure form of an or statement. It’s either night OR day, but never both together. This would be called an exclusive OR statement.

In assembly language, the inclusive or function is called ORA for OR Accumulator. The other is called EOR for Exclusive OR. Here are the charts for both functions:

ORA

<table>
<thead>
<tr>
<th>Acc.</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

EOR

<table>
<thead>
<tr>
<th>Acc.</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
First, consider the chart for ORA. If either or both corresponding bits in the Accumulator and the test value match, then the result will be a 1. Only when neither bit is 1 does a 0 value for that bit result. The main use for ORA is to force a 1 at a given bit position. In this manner it is the complement to the AND operator (which forces 0’s).

Here are some examples of the effect of the ORA command:

<table>
<thead>
<tr>
<th>Example #1:</th>
<th>Example #2:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Accumulator:</strong></td>
<td><strong>ORA Value:</strong></td>
</tr>
<tr>
<td>$80 1000 0000</td>
<td>#$03 0000 0011</td>
</tr>
<tr>
<td><strong>ORA Value:</strong></td>
<td><strong>Result:</strong></td>
</tr>
<tr>
<td>#$03 0000 0011</td>
<td>$83 1000 0011</td>
</tr>
</tbody>
</table>

Use of ORA also conditions the Sign and Zero flags, depending on the result, which is automatically put into the Accumulator.

In this setup, the switches are in parallel, giving the electricity a choice of paths to the light. The light will only come on when either switch A OR B are on (inclusive).

To specifically set bits in a memory location to ones, there is a special purpose instruction TSB (Test and Set Bits). This is used by loading the Accumulator with a mask where the bits are set at each position that you want forced to one in the memory location. For example, if you wanted to set the high bit of a memory location, the following would do it:

```
LDA #$80 ; %1000 0000 = HIGH BIT SET
TSB MEM ; SET HIGH BIT IN MEMORY
```

Like TRB, the result is left only in memory, and the Accumulator is not changed.

**Exclusive OR: EOR**

The EOR command is somewhat different in that the bits in the result are set to 1 only if one or the other corresponding bits in the Accumulator and test value are set to 1, but not both.

EOR has a number of uses. The most common is in encoding data. An
interesting effect of the table given is that for any given test value, the Accumulator will flip back and forth between the original value and the result each time the EOR is done. For example:

**Example #1:**

First pass:

| Accumulator: | $80 1000 0000 | $83 1000 0011 |
| ORA Value:   | #$03 0000 0011 | $0A 0000 1010 |
| Result:      | $83 1000 0011 | $89 1000 1001 |

Second pass:

| Accumulator: | $83 1000 0011 | $89 1000 1001 |
| ORA Value:   | #$03 0000 0011 | $0A 0000 1010 |
| Result:      | $80 1000 0000 | $83 1000 0010 |

**Example #2:**

Program 9-1. Hi-Res Screen Inverter

```
1  ******************************************************************************
2  *   HI-RES SCREEN INVERTER     *
3  *   MERLIN ASSEMBLER         *
4  ******************************************************************************
5
6  ORG $300
7
8  =0006
9  PTR EQU $06 ; $06,07
10 =2000
11 SCREEN EQU $2000 ; HIRES PAGE 1
12
13 000300: A9 20 11 ENTRY LDA #$>SCREEN ; HIGH ORDER BYTE OF $2000
14 000302: 85 07 12 STA PTR+1 ; SET HIGH BYTE OF PTR
15 000304: A9 00 13 LDA #$<SCREEN ; LOW ORDER BYTE OF $2000
16 000306: 85 06 14 STA PTR ; SET LOW BYTE OF PTR
17
18 000308: A0 00 18 START LDY #$00 ; INIT Y-REGISTER
19
20 00030A: B1 06 20 LOOP LDA (PTR),Y ; GET EXISTING BYTE
21 00030C: 49 FF 21 EOR #$FF ; FLIP BITS
22 00030E: 91 06 22 STA (PTR),Y ; PUT BACK IN MEMORY
23 000310: C8 23 INY ; Y = Y + 1
24 000311: D0 F7 =030A 24 BNE LOOP ; BRANCH WHILE Y = $1 TO $FF
25
26 000313: E6 07 26 NXT INC PTR+1 ; PTR GOES FROM $2000 TO $2100, ETC.
27 000315: A5 07 27 LDA PTR+1
28 000317: C9 40 28 CMP #$40 ; STOP WHEN PTR = $4000
29 000319: 90 ED =0308 29 BCC START ; NOT THERE YET
30
31 00031B: 60 31 EXIT RTS
```

--End Merlin-16 assembly, 28 bytes, Errors: 0
Logical and Shift Operators

This phenomenon is used extensively in Apple IIGS graphics to allow images to overlay each other, without destroying the image below.

**EOR** can also be used to reverse specific bits. Simply place 1's in the positions you wish reversed. EOR #$FF reverses all the bits in a byte.

Program 9-1 uses EOR to reverse, and then restore the entire Hi-Res screen. It's just a variation on Screen Clear #1A in Chapter 10. Use Program 9-2 to draw the screen (or BLOAD your favorite picture), and then change the image.

**Program 9-2. Hi-Res Screen Inverter Loader**

```plaintext
10 PRINT CHR$(4);"BLOAD HIRES.INVERT,A$300"
20 HGR
25 HCOLOR = 3
30 HPLT 0,0
40 FOR I = 1 TO 50
45 HCOLOR = RND (1) * 7: HPLT X,Y
50 X = RND (1) * 279
55 Y = RND (1) * 159
60 HPLT TO X,Y
70 NEXT I
100 HOME: VTAB 22: PRINT "PRESS A KEY, ESCAPE TO END"
105 GET A$  
110 IF A$ = CHR$(27) THEN TEXT: END
120 CALL 768: GOTO 100
```

Program 9-3 is a final example of how to use a shift operator. This routine prints a number in the binary form to the screen. It works by successively shifting each bit in the byte into the Carry. At that point, depending on whether it was a one or a zero, the routine prints a one or a zero plus a space. You can delete the part that prints spaces if you want a more compact display.

**Program 9-4** will load the object file, and input the numbers to be printed.
Chapter 9

Program 9-3. Binary Number Printer

1 **********************************************
2 *           BINARY NUMBER PRINTER          *
3 *           MERLIN ASSEMBLER              *
4 **********************************************
5
6 ORG $300
7
8 =0006 NUM EQU $06
9 =0007 TEMP EQU $07 ; TEMPORARY WORK AREA
10
11 =FDED COUT EQU $FDED
12
13 000300: A5 06 14 PRBIT LDA NUM ; GET VALUE TO PRINT
14 000302: A2 08 15 LDX #$08 ; START COUNTER FOR # OF BITS
16
17 000304: 0A 17 TEST ASL ; GET A BIT FROM THE BYTE
18 000305: 85 07 18 STA TEMP ; SAVE THE ROTATED BYTE FOR A WHILE
19 000307: 90 0C ==0315 19 BCC PZ ; GOTO PRINT ‘O’ IF BIT IS CLEAR
20
21 000309: A9 B1 21 PO LDA #$’1’ ; PRINT A ‘1’
22 00030B: 20 ED FD 22 JSR COUT ; ‘SPC’
23 00030E: A9 A0 23 LDA #$’A0’
24 000310: 20 ED FD 24 JSR COUT ; PRINT THE ‘SPACE’
25 000313: B0 0A ==031F 25 BCS NXT ; GO FOR THE NEXT BIT IN THE BYTE
26
27 000315: A9 B0 27 PZ LDA #$’0’
28 000317: 20 ED FD 28 JSR COUT ; PRINT THE ‘0’
29 00031A: A9 A0 29 LDA #$’A0’
30 00031C: 20 ED FD 30 JSR COUT ; ‘SPC’
31
32 00031F: A5 07 32 NXT LDA TEMP ; GET THE ROTATING BYTE BACK
33 000321: CA 33 DEX ; X = X - 1
34 000322: D0 E0 ==0304 34 BNE TEST ; SHIFT IT AGAIN IF WE’RE NOT DONE
35
36 000324: 60 36 EXIT RTS
37
38
39
40
41

--End Merlin-16 assembly, 37 bytes, Errors: 0

Program 9-4. Binary Number Printer Loader

10 PRINT CHR$(4);“BLOAD BINARY.PRINT,A$300”
20 INPUT “NUMBER TO CONVERT?”;N
25 IF N = 0 THEN END
30 POKE 6,N: REM STORE NUM FOR ROUTINE
35 PRINT N;“ = “;
40 CALL 768: REM CONVERT AND PRINT
50 PRINT : PRINT : GOTO 20
Trouble Shooting

Computer repair specialists use the binary nature of numbers to help track down the cause of a hardware-related computer errors. For example, suppose you had a parallel printer that always printed an at sign ( @ ) where spaces should be. The decimal ASCII values for a space is 96 and 64 for the @. This information doesn’t appear to tell you very much.

Now look at the numbers in binary:

Space: 0110 0000
@:       0100 0000

In examining the bit pattern, you can see that the patterns are identical, except for bit 5. This indicates that the signal for bit 5 is not coming through, and that checking the printer cable wire or connector that is associated with bit 5 would be a good idea.

Most disk- and memory-related data errors are caused by a similar problem, that is, a given bit flips from a one to a zero or vice versa. In modem communications, where data is transmitted over a phone line, this type of error is very likely. It’s easy to miss a signal and drop a one to a zero, or for unexpected static to turn a zero into a one. To correct errors like this in transmission, a system of checksums has evolved. A checksum is just a number value used to verify the accuracy of transmitted, or even stored, data.

For example, we could design a system where after every 10 bytes were transmitted, we would then send the sum of those ten bytes. If the receiving computer added up the values for the bytes received, and got a different sum, it would know an error had occurred, and the data could be retransmitted. There are a lot of different error-checking systems, but most follow this general principle.

You can also build a checksum into your own programs, so that the running program can check to make sure that no damage has occurred to the program or the data it uses. The Merlin assembler has a special pseudo-op, CHK (CHecKsum) that stores a checksum byte in the program at the point where the instruction is used. The checksum in Merlin is generated by doing successive EORs on each byte of the program, and carrying the result of each EOR along for the next one, until the entire program has been scanned. A simple version would look something like this:

```
LDA   BYTE1 ; GET 1ST BYTE
EOR   BYTE2 ; EOR WITH 2ND BYTE, KEEP RESULT
EOR   BYTE3 ; EOR WITH 3RD BYTE, KEEP RESULT
EOR   BYTE4 ; ETC.
```
Chapter 9

The final result makes up the checksum byte. Program 9-5 is an example that checks itself to make sure everything's the way it should be. Program 9-6 is an Applesoft BASIC that checks the checksum program.

On the first pass, you should get the message "Program Checks OK." Line 30 of the BASIC program then POKEs a foreign byte into the program, in the middle of the error message, just so you can see something's changed. On the second run of the program, the program detects that it has been changed, and prints out the error message.

If you're writing programs that others will have to type in, such as in a club newsletter or magazine article, you may want to include the Merlin checksum at the end of each your listings so that the person typing the program in can make sure there have been no errors. On longer listings in the remainder of this book, a CHK byte will be included at the end so you too can make sure there are no errors in your listing. (COMPUTE! Publication's MLX and Automatic Proofreader programs use a similar checksum technique to the one described here except that MLX and the Automatic Proofreader check each line as it is entered.)

Your Apple IIGS also has a checksum, which is in the last byte of the Applesoft BASIC ROM area at $F7FF. This byte is checked by the internal diagnostic routines that run when you press Control-Option-Open-Apple-Reset. Some copy-protected programs do a checksum on the entire Applesoft BASIC/Monitor ROMs when they run to make sure they're not on a non-standard machine. This really isn't a good idea though, the programs then stop running when Apple updates a ROM, or brings out a new machine.

Program 9-5. Checksum Demo

```
1  ******************************************************
2  * PROGRAM CHECKSUM DEMO  *
3  * MERLIN ASSEMBLER        *
4  ******************************************************
5
6      ORG $300
7
8      =0006 8 PTR EQU $06
9      =0008 9 TEMP EQU $08 ; TEMPORARY WORK AREA
10
11     =FDED
12     11 COUT EQU $FDED
13
14      000300: A9 00 13 CHECK LDA #<CHECK ; LOW BYTE OF BEG OF PROG.
15      000302: 85 06 14 STA PTR
16      000304: A9 03 15 LDA #>CHECK ; HIGH BYTE OF BEG OF PROG.
17      000306: 85 07 16 STA PTR+1
18
19      000308: 64 08 18 STZ TEMP ; STORE STARTING VALUE
```
Logical and Shift Operators

00030A: A5 08 20 LOOP
00030C: 52 06 21 LDA TEMP
00030E: 85 08 22 EOR (PTR) ; EOR WITH MEMORY, RESULT IN ACC.
000310: E6 06 23 STA TEMP ; STORE RESULT
000312: B0 02 =0316 24 21 INC PTR ; PTR = PTR + 1
000314: E6 07 25 BNE NEXT2 ; NO WRAP-AROUND
000316: A5 07 26 21 INC PTR + 1
000318: C9 03 27 NEXT2
00031A: 90 EE =030A 28 LDA PTR + 1
00031C: A5 06 29 CMP #>CHKSUM ; AT END OF PROGRAM YET?
00031E: C9 6C 30 BCC LOOP ; NOT YET...
000320: 90 E8 =030A 31 LDA PTR
000322: A5 08 32 CMP #<CHKSUM
000324: CD 6C 03 33 BCC LOOP ; STOPS AT BYTE JUST BEFORE 'CHKSUM'
000326: DO OE 34 TEST
000328: A0 00 35 LDA TEMP ; GET CHECKSUM
00032A: 89 45 03 36 CMP CHKSUM ; COMPARE TO STORED VALUE
00032C: FO 13 37 BNE ERROR ; UH OH, SOMETHING'S CHANGED!
00032E: 20 ED FD 38 NEXT CHARACTER
000331: CS 80 FS 39 PROGRAM NOP ; YOUR PROGRAM HERE...
000334: 80 45 03 40 PRINT LDY #$00
000335: 60 55 DONE 41 LOOP1 LDA MSSG1,Y
000337: A0 00 42 BEQ DONE ; END OF MESSAGE
000339: B9 45 03 43 LOOP2 LDA MSSG2,Y
00033B: F0 13 =0344 44 JSR COUT
00033D: 20 ED FD 45 INY ; NEXT CHARACTER
00033F: 80 F5 =032C 46 BRA LOOP1
000341: C8 45 47 ERROR LDY #$00
000343: 80 5F =0339 48 LOOP2 LDA MSSG2,Y
000345: 32 61 CHK ; STORE CHECKSUM
000347: 60 55 DONE RTS
000349: D0 F2 EF E7 57 MSSG1 ASC "Program Checks Ok."
00034A: F2 E1 ED A0 C3 E8 E5 E3 58 8D,00
00034C: EB F3 A0 CF EB AE 8D 00 59 "Error in Program!"
00034E: C5 F2 F2 EF 59 MSSG2 ASC "Program Checks Ok."
000352: 50 8D 00 5A 8D,00
000354: E7 F2 E1 ED A1 8D 00 5B NEXT CHARACTER
000356: 32 61 CHKSUM CHK ; STORE CHECKSUM
000358: 61

--End Merlin-16 assembly, 109 bytes, Errors: 0

175
Chapter 9

Program 9-6. Checksum Demo Loader

10 PRINT CHR$ (4);"LOAD CHECKSUM.TEST,A$300"
20 CALL 768: REM RUN THE TEST
30 POKE 861, ASC ("x") + 128: REM CREATE ERROR
40 CALL 768: REM RUN TEST AGAIN
Chapter 10

Addressing Modes and Improved Printing
An *addressing mode* is a term that simply refers to the different ways a given 65816 instruction can locate the data or memory location it needs. Addressing modes is a rather fundamental concept to programming that has been used throughout this book, we just haven’t called it by name.

Flexibility in the ways in which you can address memory is the key to even greater power in your own programs. Consider this chart of just a some of the addressing modes available on the Apple IIgs:

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Example</th>
<th>Hex Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>LDA #$A0</td>
<td>A9 A0</td>
</tr>
<tr>
<td>Absolute</td>
<td>LDA $7FA</td>
<td>AD FA 07</td>
</tr>
<tr>
<td>Absolute Long</td>
<td>LDA $0107FA</td>
<td>AF FA 07 01</td>
</tr>
<tr>
<td>Direct (Zero) Page</td>
<td>LDA $80</td>
<td>A5 80</td>
</tr>
<tr>
<td>Implicit/Implied</td>
<td>TAY</td>
<td>A8</td>
</tr>
<tr>
<td>Relative</td>
<td>BCC $3360</td>
<td>90 0F</td>
</tr>
<tr>
<td>Indexed</td>
<td>LDA $200,X</td>
<td>BD 00 02</td>
</tr>
<tr>
<td>Indexed Long</td>
<td>LDA $010200,X</td>
<td>BF 00 02 01</td>
</tr>
<tr>
<td>Indirect</td>
<td>LDA ($80)</td>
<td>B2 80</td>
</tr>
<tr>
<td>Indirect Long</td>
<td>LDA [$80]</td>
<td>A7 80</td>
</tr>
<tr>
<td>Indirect Indexed</td>
<td>LDA ($80),Y</td>
<td>B1 80</td>
</tr>
<tr>
<td>Indirect Indexed Long</td>
<td>LDA [$80],Y</td>
<td>B7 80</td>
</tr>
<tr>
<td>Indexed Indirect</td>
<td>LDA ($80,X)</td>
<td>A1 80</td>
</tr>
<tr>
<td></td>
<td>also: JSR ($300,X)</td>
<td>FC 00 03</td>
</tr>
</tbody>
</table>

In looking at the examples, you should find all but the last six very familiar. We have used each of them in previous programs presented in this book. This list does not present all of the available addressing modes on the 65816, but they are the ones we will use most frequently. Let’s take a look at each more closely.

**Immediate mode.** The *immediate* mode was used to load a register with a specific value. In most assemblers this is indicated by the use of the # sign
preceding the value to be loaded. This contrasts the *absolute* mode in which the value is retrieved from a given memory location. In this mode, the exact address you’re interested in is given. This line will load the Accumulator with the value $A0:

LDA #$A0

Loads the value $A0 into the Accumulator.

**Absolute addressing mode.** In contrast to the immediate mode, the absolute mode retrieves a value from a given memory location. In this mode the exact address you’re interested in is given.

LDA $7FA

Copies the value found in location $7FA into the Accumulator.

**Direct page mode.** *Direct page* addressing is really just a variation on the absolute mode. The main difference is that because the addresses referenced are always in the range of $00 to $FF, direct page addressing only takes two bytes for the complete instruction (see the third column), whereas in the more general case of absolute addressing, three or even four bytes per instruction are required.

A *page* of memory is $100 (256 decimal) bytes. Starting at $0, the first page of memory is called the zero page. Addresses $100 to $1FF are called page one, and so on. Direct page addressing is sometimes called zero page addressing because the default location for the direct page is, in fact, page zero. The direct page area can be moved anywhere in the first 64K of memory, and this is described in the next chapter.

LDA $80

Copies the value found in location $80 into the Accumulator.

**Implicit addressing.** *Implicit* (or *implied*) is certainly the most compact instruction in that only one byte of object code is generated by the assembler. The TAY command (*Transfer Accumulator to the Y register*) needs no additional address bytes because the source and destination of the data are implied by the very instruction itself.

TAY

Copies the value found in the Accumulator into the Y register.

**Relative addressing.** *Relative* addressing is done relative to where the instruction itself is found. Although the Example column (as listed by the Monitor L command) shows it as a branch to a specific address, you’ll notice that the actual hex code is merely a plus or minus displacement from the branch point. This was discussed in a previous chapter.
Indexed addressing. At this point you should be able to create your own simple programs. The problem with the four modes discussed so far is that the programs created are rather inflexible in dealing with data from the outside world (such as in input routines), and in doing things like accessing tables and large blocks of data.

Indexed addressing is necessary to write input routines. In the pure form, the contents of the X or Y register are added to the address given in the instruction to determine the final address. In the example given (LDA $200, X), if the X register holds 0, the Accumulator will be loaded with the contents of location $200. If the X register instead holds a 4, location $204 will be accessed. Indexing is ideal for accessing tables of data, or any range of memory.

This addressing mode works fine as long as you know exactly where in memory the data table is. What happens when the table could be in a movable location, such as when dealing with an interface card that could be in any slot? Another problem area would be in dealing with a large variety of possible base addresses, such as the starting address of each line on the text or graphics screens. To print a character or to plot a point, the computer uses the base address of each line and then adds an appropriate horizontal offset to get to the desired position. Using indexed addressing, there would have to be a specific instruction for each possible screen line since the address is built into the instruction itself.

Indirect addressing. The solution to this is to use the indirect indexed mode. This really is an elegant method. First the 65816 goes to the given direct page location (the base address must be a direct page address). Indirect addressing is indicated in the assembly source listing by the use of parentheses. In the example, LDA ($80), the 65816 goes to locations $80 and $81 to get the low- and high-order bytes of the address stored there. Then it adds the value of the Y-register to that address (see Figure 10-1).

Figure 10-1. Indirect Indexed Addressing

65816: LDA ($80),Y (Y-register=$04)

Location: ($80 $81)

Contents: $00 $02

(Address = $200 + $04 = $204) → <ACCUMULATOR>
These two-byte direct page address pairs are often called pointers, and you will often hear them referred to in dealing with various programs on the Apple. In fact, by looking at various reference books available for the Apple, you’ll observe quite a number of these byte pairs used by Applesoft BASIC, the Monitor, ProDOS and just about everything on the Apple. Pointers are used to keep track of all sorts of continually changing things, like where the program is, the locations of strings and other variables, and many other nifty items. Pointers find such frequent use because of the tremendous power of indirect addressing.

If you want to simulate the LDA $200,X command with the indirect mode, you first store #$00 in $80 and #$02 in $81 (00 and 02 are the low and high order bytes of the address $200). Then you use the command LDA ($80),Y.

**Indirect long addressing mode.** There is a long addressing mode for indirect indexed addressing, also. In that case, three bytes are retrieved starting at the direct page address. Long direct page addressing is indicated in the assembly source listing by using the square brackets instead of parentheses:

LDA [$80],Y

In this situation, data is loaded into the accumulator from the long address pointed to by $80,81,82, and the Y register is added to this data, as shown in Figure 10-2.

Figure 10-2. Indirect Long Addressing

65816: LDA ($80),Y (Y register=$04)

Location: ($80 $81 82) ($010204)

Contents: $00 $02 $01 $??

(Address = $010200 + $04 = $204) → <accumulator>

You don’t have to define the pointer as three bytes just because the 65816 only uses three bytes to determine the final address. In fact, doing so would create problems trying to load and store bytes there using the usual LDA/STA method. This is because the load/store of the low-order word would have to be in 16-bit mode, and the high-order word would have to be in 8-bit mode. The other alternative would be to do all three load/store operations in 8-bit mode.
A better solution is to just define four bytes in your source listing equates. Then it will be much easier to just to do two load/store operations in the 16-bit mode to maintain a pointer.

There is also a special case of indirect indexed addressing, simply called _indirect_. Because there are times when you just want to examine a variable memory location, without necessarily scanning an entire table, you can omit the use of the Y register in the instruction:

```
LDA ($80)
```

This is equivalent to the instructions:

```
LDY #00
LDA ($80),Y
```

but it saves an instruction and does not affect the Y register. This mode may also be used for long addresses:

```
LDA [$80]
```

**Use of the X and Y index registers.** You may have noticed that the X register was used in one case and the Y register in the other. It turns out that the X and Y registers cannot always be used interchangeably. The difference shows up depending on what actual instruction and which addressing mode you're using. As it happens, indirect indexed addressing can only be done using the Y register. To know what is legal, you should make use of Appendix A to see which addressing modes can be used with any given command.

**Indexed indirect addressing.** The last addressing mode, _indexed indirect_, is probably the most unusual. In this case, the contents of the X register (Y cannot be used for this mode) are used at the beginning of the address calculation.

In the case of the sample instruction in the chart, if the X register held 0, a LDA ($80,X) would go to $80 and $81 for the two-byte address, and then load the Accumulator with the contents of the indicated location. If instead the X register held a 04, the memory address would be determined by the contents of $84 and $85.

Usually, then, the X register is loaded with multiples of 2 to access a series of continuous pointers in zero page.
The most common use of this command is for creating tables of entry address for subroutines. You might at first think that since the location of subroutines in your programs or in the Monitor seem to be fixed, such a process would not be necessary. Consider, however, the case of jumping to a routine on an interface card such as a printer card. There are also routines in the mouse firmware that may move in future revisions to the Apple IIgs. Because of this, it is handy to have a way to execute a JMP or JSR using an indirect address.

This addressing mode also provides for an elegant command processor. Assume for a moment that your program will have an input of the values 1, 2, or 3, depending on a menu choice from the user. One method of handling the commands is to do a number of CMP and conditional branch instructions, like this:

```
CHECK LDA CMD ; VALUE FOR COMMAND
         ; ASSUME = "1", "2" OR "3"
       CMP #"1"
       BNE CHK2
       JSR CMD1 ; LOC OF ROUTINE #1
       JMP NEXT ; GO TO NEXT PART OF PROGRAM

CHK2  CMP #"2"
      BNE CHK3
      JSR CMD2 ; LOC OF ROUTINE #2
      JMP NEXT ; GO TO NEXT PART OF PROGRAM

CHK3  CMP #"3"
      BNE AGAIN ; CMD NOT HERE . . . TRY AGAIN
      JSR CMD2 ; LOC OF ROUTINE #2
      JMP NEXT ; GO TO NEXT PART OF PROGRAM

NEXT  NOP ; MORE OF YOUR PROGRAM HERE . . .
```

An alternative approach uses a table of all your subroutine entry points, and the indexed indirect addressing mode. Before you can see exactly how it's
done, though, you’ll need to know how to have the assembler create a data table in the first place.

**Assembler Data Storage Pseudo-Ops**

Before all this can be put to work, there is still one more question to answer: How do you store just pure data within a program?

All the commands discussed so far are actual commands for the 65816. There is no data command, as such. What is available are the pseudo-ops of your particular assembler. You’ll recall that directives are commands used by the assembler itself (such as ORG and EQU) during the assembly of a source listing to tell the assembler to do something, like save a file to disk, or set the default BLOAD address.

Pseudo-ops are assembler commands that look very much like assembly language instructions. The difference is that instead of generating a specific 65816 instruction, they create a group of bytes which are not necessarily executable instructions—they’re usually data.

Assembler directives and pseudo-ops are also a little like porpoise and dolphin, or assembly language and machine language: Although they technically have different meanings, the words are used rather interchangeably by many people.

The rule-of-thumb is: Pseudo-ops generate assembled bytes of object code; directives tell the assembler to save a file, to print a new page, or to do some other housekeeping (but non-code generating) operation.

Specific directives and pseudo-ops vary from one assembler to another, so you’ll have to consult your own manual to see how your assembler operates.

In general, the most common use of pseudo-ops is for data storage. The general procedure is to define a block of one or more bytes of data, and then either to put that data at the end of the program or to skip over that block with a branch or jump instruction when executing your program. Data can usually be entered either as hex bytes or as the ASCII characters you wish to use. The assembler will, in that case, automatically translate the ASCII characters into the proper hex numbers.

**Printing a String of Characters**

To print a string of characters on the screen, like Hello there, you’ll need to use both an indirect addressing mode and an assembler pseudo-op to define some data.

In the Merlin assembler, there is a HEX command for directly entering the hex bytes of a data table. Program 10-1 is a sample program using the indexed address mode.
Chapter 10

Program 10-1. Print Demo 1

1 ******************************************************
2 * SAMPLE 'PRINT' PROGRAM #1  *
3 * MERLIN ASSEMBLER          *
4 ******************************************************

5 ORG $300
6
7 =FDED
8 COUT EQU $FDED
9
00300: A2 00
10 BEGIN LDX #$00 ; START WITH X = 0

00302: BD 13 03 12 LOOP LDA DATA,X ; READ A BYTE OF DATA
00305: 20 ED FD 13 JSR COUT ; PRINT ASCII CHARACTER
00308: E8 14 INX ; X = X + 1
00309: E0 0B 15 CPX #11 ; DONE WITH LIST
0030B: 90 F5 =0302 16 BCC LOOP ; X < 11 MEANS NO...
0030D: A9 8D 17 LDA #$8D ; #$8D = CARRIAGE RETURN
0030F: 20 ED FD 18 JSR COUT ; PRINT IT

20 DONE RTS

00312: 60

21 DATA

00313: C8 E5 EC EC 22 DATA HEX C8,E5,EC,EC,EF,A0,F4,E8,E5,F2,E5
00317: EF A0 F4 E8

23 * DATA = 'Hello there'

---End Merlin-16 assembly, 30 bytes, Errors: 0

The hex values in the data table are the ASCII values for each letter plus $80. This sets the high bit of each number, which is what the Apple expects to have the letter printed out properly when using COUT.

This program uses the X register as both a counter for the number of characters to print and as a pointer to the place in the data block from which the next character will be read. COUT does not affect the X register, so you don’t have the concern about it being altered, as we did with the accumulator and COUT in the last chapter.

Line 10 starts the program by setting the X register to 0. The main loop then uses the indexed addressing mode to add the value of the X register to the address of the DATA table at $313. As X is incremented after each character is printed, it’s checked against the number of character to be printed—the number of characters in the string. Notice that line 15 compares against the decimal value 11 by not using the dollar sign in front of the number. The pound sign is still needed to show it’s an immediate mode value.
Note that there are 11 characters in the string, but the use of 11 in the CPX instruction is almost coincidence. Remember that the first character read from the table is done with X starting at 0. On this basis, when X has reached 10, it has reached the last character of the string. In the program, however, X is incremented before the check. Thus, X will be 11 after the last character has been printed.

When writing programs you must keep all of the following in mind: the starting value of the indexed register, which test you want to use to determine the end of the loop (BCC, BEQ, BNE, BCS, or another test), and whether the test occurs before or after an increment or decrement has been done.

After printing, the program ends with a carriage return. Remember that in assembly language you must usually do everything yourselves. This means you cannot assume an automatic carriage return at the end a printed string.

You’ll notice the data table is put at the end of the program. This is because the computer, as such, can’t tell the difference between data and a program. It’s up to you to keep your data tables from being executed as a program. For a closer look, BLOAD the assembled file and list it from the Monitor.

```
*300L
1=m  1=x  1=LCbank (0/1)
00/0300: A2 00  LDX #00
00/0302: BD 13 03  LDA 0313,X
00/0305: 20 ED FD  JSR FDED
00/0308: E8  INX
90/0309: E0 0B  CPX #0B
00/030B: 90 F5  BCC 0302 {-0B}
00/030D: A9 8D  LDA #8D
00/030F: 20 ED FD  JSR FDED
00/0312: 60  RTS
00/0313: C8  INY
00/0314: E5 EC  SBC EC
00/0316: EC EF A0  CPX A0EF
00/0319: F4 E8 E5  PEA E5E8
00/031C: F2 E5  SBC (E5)
```

Notice that after the RTS at $312, the data table lists as instructions also. This is because the Monitor disassemble command (like the 65816 while running a program) has no way of knowing that the bytes from $313 to $31D are ASCII text (or any other kind of data for that matter).
Chapter 10

Now, while still in the Monitor, type

300.320

The screen will display

*300.320

00/0300: A2 00 BD 13 03 20 ED FD -",= .. m{}
00/0308: E8 E0 0B 90 F5 A9 8D 20-h'..u).
00/0310: ED FD 60 C8 EC ES EC EF-m'}Hello
00/0318: A0 F4 E8 ES F2 ES 06 DO- there.P
00/0320: 24-

To the right of the hex values, you can see the ASCII equivalents. The Monitor dump command normally only shows as printable characters those bytes whose high bit is set (value is greater than $7F). At the beginning of the dump, the assembly language instructions do not list as any intelligible characters. At the bottom of the group you can see the phrase Hello there.

By using both the Monitor list and dump commands, you can sometimes tell whether the part of memory you’re looking at has a program or data in it. Of course, certain opcodes (like $AD for LDX) can appear like ASCII text in a dump, and ASCII text (like $CB = INY) can appear like an instruction. It’s also possible that the data stored in a program is not necessarily ASCII text. Perhaps it’s the sizes of different objects, the quantity of an item, or some other information. In general, though, a little thoughtful examination can decipher many programs, even without the source code.

Some assemblers, like the Merlin 8/16, also have utilities to create actual text file source listings from object code in memory. Even these, though, require thoughtful use to yield intelligent results.

Sample Print Program #1 with APW

Program 10-2 is the same print demo as Program 8-1 for the APW assembler.

The main differences with the APW listing are the START, END, LONGA and LONGI directives, and the way that stored data is defined with the DC pseudo-op. START and END were discussed in Chapter 5. LONGA and LONGI are directives that tell the assembler not to create object code that expects 16-bit operations.

Sixteen-bit refers to an operating mode of the 65816 where most of the registers are two bytes (16 bits) in size, instead of the 8-bit size we’ve been looking at so far. Because Applesoft BASIC was written on older Apples that only had one-byte registers, we’ve been limiting the programs written thus far to this same mode. In the APW assembler, you must start a source listing with
Addressing Modes and Improved Printing

Program 10-2. Print Demo 1 for the APW Assembler

******************************************************************************
* SAMPLE ‘PRINT’ PROGRAM #1  *
* APW ASSEMBLER               *
******************************************************************************

KEEP D.PROG1
ORG $300
LONGA Off
LONGI Off

MAIN START
COUT EQU $FD3D

BEGIN LDX #$00 ; START WITH X = 0
LOOP LDA DATA,X ; READ A BYTE OF DATA
JSR COUT ; PRINT ASCII CHARACTER
INX ; X = X + 1
CPX #11 ; DONE WITH LIST
BCC LOOP ; X < 11 MEANS NO...
LDA #$8D ; #$8D = CARRIAGE RETURN
JSR COUT ; PRINT IT

DONE RTS

DATA DC H’C8 E5 EC EF A0 F4 E8 E5 F2 E5’

* DATA = ‘Hello there’

END

LONGA OFF and LONGI OFF if you want the following accumulator and index register (X and Y) instructions to be compatible with Applesoft BASIC. In just a few chapters, you’ll see how to use the extended register size in other programs. For the time being, though, just start any APW source listing with LONGA OFF and LONGI OFF.

APW doesn’t have a HEX pseudo-op. Instead, it has the pseudo-op DC (for Defined Constant), which is then followed by a modifier to tell the assembler what kind of number you want to define. For our program, H stands for Hex and tells the assembler to use the values that follow to determine how many bytes of hex data to include in the data block. The single apostrophe is call a delimiter, and it is used to contain the list of data.
Chapter 10

ASCII Data Pseudo-Ops

You might now be thinking that it’s not very convenient to have to look up the ASCII values of each character that you want to print. It’s also not much fun to have to count how many characters there are in the string. Well, you’re in luck. There is a way of writing your program to eliminate both these inconveniences. Program 10-3 is the new program.

Program 10-3. Print Demo 2

```
1 *****************************************************
2 * SAMPLE 'PRINT' PROGRAM #2  *
3 * MERLIN ASSEMBLER            *
4 *****************************************************
5
6       ORG $300
7
8       =FDED
9
000300:A2 00
10 BEGIN  LDX #$00       ; START WITH X = 0
11
000302:BD 0E 03
12 LOOP   LDA DATA,X    ; READ A BYTE OF DATA
000305:F0 06 =030D
13 BEQ    DONE         ; 0 = END OF STRING
000307:20 ED FD
14 JSR    COUT          ; PRINT ASCII CHARACTER
00030A:E8
15 INX    ; X = X + 1
00030B:80 F5 =0302
16 BRA    LOOP          ; NEXT CHARACTER
17
18 DONE   RTS
19
00030D:60
20 DATA   ASC "Hello there",8D,00
000312:EF A0 F4 E8 E5 F2 E5 8D
21
00031A:00

--End Merlin-16 assembly, 27 bytes, Errors: 0
```

This program has two improvements. First, the pseudo-op ASC (ASCII) is used to automatically translate the text we want to print into the proper hex bytes in memory. The Merlin 8/16 uses quotes (" ) to signify bytes with the high bit on, and it uses the apostrophe to signify bytes with the high bit clear. In addition, Merlin’s ASC command lets us end the string of characters with any hex values, by just using commas.

We’ll use this first to include the carriage return as part of the string, and then to put a zero at the end of the string that we can detect with a BEQ instruction.

You can see that, on line 13 of the program, when a character is loaded
Addressing Modes and Improved Printing

with a value of zero, the BEQ will detect it and branch to DONE. This means that we don’t have to check for a specific length of a string, which makes editing the source listing much easier if you decide to change a printed message in a program.

If you’re using the APW assembler, the equivalent source listing is shown in Program 10-4.

Program 10-4. APW Print Demo 2

******************************************
* SAMPLE 'PRINT' PROGRAM #2    *
*    APW ASSEMBLER        *
******************************************

KEEP  D.PROG2
ORG  $300
LONGA  OFF
LONGI  OFF
MSB  ON
MAIN  START
COUT  EQU  $FDED
BEGIN  LDX  #$00  ; START WITH X = 0
LOOP   LDA  DATA,X  ; READ A BYTE OF DATA
       BEQ  DONE  ; 0 = END OF STRING
       JSR  COUT  ; PRINT ASCII CHARACTER
       INX  ; X = X + 1
       BRA  LOOP  ; GET NEXT CHARACTER
DONE   RTS
DATA   DC  C'Hello there',H'8D 00'

The APW assembler uses C (for Character) to designate text data in the DC command. You can also mix data types on one line, so you can put the H'8D 00' bytes at the end of the string.

**VTAB and HTAB in Assembly Language**

Being able to print something on the screen is nice—you know how to clear the screen (HOME = $FC58) and print a string (COUT = $FDF0)—but what about positioning the cursor to control where the string is printed?

The easiest way, for now, is to use the Monitor routine VTAB for the
vertical position, and to directly control the horizontal cursor location, CH (for Cursor Horizontal = memory location $24).

Program 10-5. Print Demo 3

```
1  ******************************************
2  * SAMPLE 'PRINT' PROGRAM #3 *
3  * MERLIN ASSEMBLER *
4  ******************************************
5  6  ORG $300
6  7
7  =FC58 8 HOME EQU $FC58
8  =FDED 9 COUT EQU $FDED
9  =FC22 10 VTAB EQU $FC22 ; VTAB TO CV
10  =0025 11 CV EQU $25 ; VERTICAL POSITION
11  =0024 12 CH EQU $24 ; HORIZ. CURSOR
12 BEGIN 15 JSR HOME
13
14 000300: 20 58 FC 15 BEGIN JSR HOME
000303: A9 0B 16 LDA #11
000305: 85 25 17 STA CV ; CURSOR VERT. POSN.
000307: 20 22 FC 18 JSR VTAB ; VTAB 12
00030A: A9 09 19 LDA #9 ; HTAB 10
00030C: 85 24 20 STA CH ; HORIZ. POSITION
21
22 00030E: A2 00 22 LDX #$00 ; START WITH X = 0
23
24 000310: BD 1C 03 24 LOOP LDA DATA,X ; READ A BYTE OF DATA
000313: F0 06 =031B 25 BEQ DONE ; 0 = END OF STRING
000315: 20 ED FD 26 JSR COUT ; PRINT ASCII CHARACTER
000318: E8 27 INX ; X = X + 1
000319: S0 F5 =0310 28 BRA LOOP ; NEXT CHARACTER
29
30 00031B: 60 30 DONE RTS
31
32 00031C: C8 E5 EC EC 32 DATA ASC "Hello there",8D,00
000320: EF A0 F4 E8 E5 F2 E5 8D
000328: 00
33
--End Merlin-16 assembly, 41 bytes, Errors: 0
```
As a working example, let's first create a program that will position the cursor somewhere on the screen and print a string starting at that position. Here's an equivalent Applesoft BASIC program:

```
10 HOME
20 VTAB 12
30 HTAB 10
40 PRINT "THIS IS A TEST"
50 END
```

Type in the third Print demo, Program 10-5.

This program is fairly equivalent to the earlier one, except that it first clears the screen with a JSR HOME. Then it stores the desired vertical cursor position in the part of memory that Applesoft BASIC uses for the cursor's vertical position, CV (for Cursor Vertical = $25). To put the cursor in the proper vertical position, we need to do a JSR VTAB. VTAB ($FC22) is a routine that uses the value stored in CV and puts the cursor at the corresponding vertical position on the screen. CV counts from 0 to 23 (as opposed to Applesoft BASIC's 1 to 24), so if we want the twelfth line, we must store 11 in CV.

Finally, a 9 is stored in CH to do the equivalent of HTAB 10. CH also counts starting at 0, so we must use a value that's one less than the normal position value.

From there, the rest of the program is identical to SAMPLE PRINT PROGRAM #2.

**Indirect Addressing**

The *indirect addressing* mode is used when you want to access one memory location depending on what you've stored in a different location, or when the range of data is greater than the 256-byte range we can access by incrementing the X register.

Let's consider the problem of clearing the screen. In this case, we want to put a space character in every memory location in the screen block ($400-$7FF). One way of doing this is shown in Program 10-6.

First, locations $06 and $07 are initializing to hold the base address of $400, the address of the first byte of the screen memory area. The label PTR is used for location $06. Line 11 shows how the assembler can use STA PTR + 1 to create STA $07. Doing it this way helps the source listing remind us that locations $06,07 form a pair. The ASCII value for A is then put in the accumulator, and Y is initialized to $00 to begin the upcoming loop.

Then we enter a loop which runs the Y register from $00 to $FF. Since this is added to the base address in $06,07 ($400), this stores an $A0 (a space)
in every location from $400 to $4FF. When Y is incremented from $FF, it goes back to $00, a move that’s detected by the BNE on line 22. At zero, it falls through, and location $07 is incremented from $04 to $05, giving a new base address of $500.

This whole process is repeated until location $07 reaches a value of $08 (corresponding to a base address of $800), at which point we return from the routine. Notice that we don’t have to reinitialize the Y register to zero for each new pass through the loop. This is because the BNE test guarantees that the Y register will be zero when the new pass is started.

Program 10-6. Clear Screen Demo 1A

```
Chapter 10

ORG $300

ENTRY LDA #$04 ; HIGH-ORDER BYTE OF $400
STA PTR+1 ; SET HIGH BYTE OF PTR
LOA #$00 ; LOW-ORDER BYTE OF $400
STA PTR ; SET LOW BYTE OF PTR

* SETS PTR (6,7) TO $400

START LDA #$A0 ; ASCII FOR 'SPACE' CHARACTER
LDY #$00

LOOP STA (PTR),Y ; PUT 'SPACE' IN MEMORY
INY ; Y = Y + 1
BNE LOOP ; BRANCH WHILE Y = $1 TO $FF

NEXT INC PTR+1 ; PTR GOES FROM $400 TO $500, ETC.
LDA PTR+1
CMP #$08 ; STOP WHEN PTR = $800
BCC START ; NOT THERE YET

EXIT RTS

--End Merlin-16 assembly, 26 bytes, Errors: 0
```
Clear to a Character
By changing the value of the #$A0 to some other character, you can clear the screen to any character you wish. In fact, you can get the value from the keyboard as we've done in earlier programs.

Let's take this opportunity to show some new assembler tricks. Program 10-7 is the revised version.

The first change to notice is that the beginning of the screen memory area has been given a label, SCRN (line 10). In general, it's a good idea to avoid specific addresses in the body of your programs. By assigning a label, you accomplish two things. First, if you ever want to change the memory area affected, you only need to change one line, rather than many individual uses of the address. Second, you make it possible for the assembler to help you discover typing errors.

It works like this: Suppose you used the address $313 in a program in a dozen different places, and in one of those uses you accidentally typed $314. The assembler has no way of knowing this is not what you intended, and it could take a long time to track down the mistake when debugging your program. On the other hand, if you assign $313 = LABEL, and then somewhere accidentally type LABEL, the assembler will automatically generate a Label Not Defined (or similar) error, which you can quickly remedy.

Since the value of SCRN could now be almost anything, we can't use the LDA #$04 and LDA #$00 for the high and low bytes. Fortunately, the assembler will do the calculation for us. By using the < and > symbols after the pound sign and before the label, the assembler will automatically calculate the high- and low-order bytes of the address defined by SCRN, and it will use them when assembling the line (see lines 14 and 16). The low-order byte is indicated by <; the high-order byte is indicated by >.

There's another change you might not see at first glance. Notice that lines 16 and 17 now use LDY, STY. As long as you know that SCRN will start at a page boundary ($400, $500, and so forth), you also know that the low-order byte will always be zero. You can use this fact to save an instruction to set the Y register to zero. Since it makes no difference which register is used on line 16 and 17 to set up PTR, we can take care of the Y register at the same time.

We've also added a check for the Escape key, so you can exit the program when you wish.

If the Escape key is not pressed, the value is then temporarily held in the variable CHAR so that it can be retrieved each time after incrementing PTR in the NEXT section.

With the screen display in the 40-column mode, BLOAD this program and run it from BASIC with a CALL 768. Each keypress will clear the screen.
Program 10-7. Clear Screen Demo 1B

```
**************
* SCREEN CLEAR PROGRAM #1B *
* MERLIN ASSEMBLER *
**************

ORG $300

PTR EQU $06 ; PTR = $06,07
CHAR EQU $08
SCRN EQU $400
KYBD EQU $C000
STROBE EQU $C010

ENTRY LDA #$>SCRN ; HIGH-ORDER BYTE OF $400
STA PTR+1 ; SET HIGH BYTE OF PTR
LDY #$<SCRN ; LOW-ORDER BYTE OF $400 AND Y = 0
STY PTR ; SET LOW BYTE OF PTR

READ LDA KYBD ; GET KYBD CHARACTER VALUE
CMP #$80 ; KEYPRESS
BCC READ ; NO, THEN TRY AGAIN.

CHECK CMP #$9B ; ESCAPE
BEQ DONE ; YES

CLEAR LDA CHAR ; GET CHAR TO 'CLEAR' TO
(PTR),Y ; PUT CHAR IN MEMORY
INY ; Y = Y + 1
BNE CLEAR ; BRANCH WHILE Y = $1 TO $FF

NEXT INC PTR+1 ; PTR GOES FROM $400 TO $500, ETC.

LOOP STA (PTR),Y ; PUT CHAR IN MEMORY
INY ; Y = Y + 1
BNE LOOP ; NOT THERE YET

AGAIN BRA ENTRY ; GO BACK FOR MORE

DONE RTS ; ALL DONE!
```

--End Merlin-16 assembly, 43 bytes, Errors: 0

196
Addressing Modes and Improved Printing

to a different character. The screen should also clear to the same character as
the key you press, including space bar and special characters. This program
shows you how fast machine language is. Clearing the screen requires over
1000 different locations to be set to the given value. In Applesoft BASIC, this
would be quite slow by comparison. Here you’ll find that the screen will clear
to different characters just as fast as you can type them.

An interesting variation on this is to enter the graphics mode by typing
in GR before calling the routine. Then the screen will clear to various colors
and different line patterns.

See what other variations you can make on this, try modifying the pro­
gram to clear the hi-res screen ($2000–$3FFF).

Using Indirect Indexed Addressing for a Better Screen Clear

If you multiply the number of lines on a 40-column screen (24) by the number
of characters (40), you get 960 (24 * 40 = 960). This is a smaller number than
the actual memory allocated to the screen memory, $400 to $7FF = $400 bytes
= 1024.

It turns out that the Apple IIGS uses an unusual system of assigning each
character position on the screen to a byte of memory. You might think it would
have been most logical to have the first 40 bytes of the first line correspond to
the first 40 bytes of memory, and so on, but that’s not the way it works. It
turns out that such a linear calculation would take much longer, and would be
harder to implement in the hardware circuitry of the computer, than the
method that is actually used. At first look, the system seems to be rather
chaotic:

<table>
<thead>
<tr>
<th>Screen Line</th>
<th>Address Range: Hex</th>
<th>Address Range: Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$400–$427</td>
<td>1024–1063</td>
</tr>
<tr>
<td>2</td>
<td>$480–$4A7</td>
<td>1152–1191</td>
</tr>
<tr>
<td>3</td>
<td>$500–$527</td>
<td>1280–1319</td>
</tr>
<tr>
<td>4</td>
<td>$580–$5A7</td>
<td>1408–1447</td>
</tr>
<tr>
<td>5</td>
<td>$600–$627</td>
<td>1536–1575</td>
</tr>
<tr>
<td>6</td>
<td>$680–$6A7</td>
<td>1664–1703</td>
</tr>
<tr>
<td>7</td>
<td>$700–$727</td>
<td>1792–1831</td>
</tr>
<tr>
<td>8</td>
<td>$780–$7A7</td>
<td>1920–1959</td>
</tr>
<tr>
<td>9</td>
<td>$428–$44F</td>
<td>1064–1103</td>
</tr>
<tr>
<td>10</td>
<td>$4A8–$4CF</td>
<td>1192–1231</td>
</tr>
<tr>
<td>11</td>
<td>$528–$54F</td>
<td>1320–1359</td>
</tr>
<tr>
<td>12</td>
<td>$5A8–$5CF</td>
<td>1448–1487</td>
</tr>
<tr>
<td>13</td>
<td>$628–$64F</td>
<td>1576–1615</td>
</tr>
<tr>
<td>14</td>
<td>$6A8–$6CF</td>
<td>1704–1743</td>
</tr>
<tr>
<td>15</td>
<td>$728–$74F</td>
<td>1832–1871</td>
</tr>
</tbody>
</table>
Chapter 10

<table>
<thead>
<tr>
<th>Screen Line</th>
<th>Address Range: Hex</th>
<th>Address Range: Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>$7A8–$7CF</td>
<td>1960–1999</td>
</tr>
<tr>
<td>17</td>
<td>$450–$477</td>
<td>1104–1143</td>
</tr>
<tr>
<td>18</td>
<td>$4D0–$4F7</td>
<td>1232–1271</td>
</tr>
<tr>
<td>19</td>
<td>$550–$577</td>
<td>1360–1399</td>
</tr>
<tr>
<td>20</td>
<td>$5D0–$5F7</td>
<td>1488–1527</td>
</tr>
<tr>
<td>21</td>
<td>$650–$677</td>
<td>1616–1655</td>
</tr>
<tr>
<td>22</td>
<td>$6D0–$6F7</td>
<td>1744–1783</td>
</tr>
<tr>
<td>23</td>
<td>$750–$777</td>
<td>1872–1911</td>
</tr>
<tr>
<td>24</td>
<td>$7D0–$7F7</td>
<td>2000–2039</td>
</tr>
</tbody>
</table>

As you look at the list, a certain pattern does emerge, but it is not necessary to go into any detail about the actual mechanics of the calculations. Suffice it to say that determining these addresses is precisely the purpose of the VTAB routine used in Program 10-3.

As you study the table, you’ll notice that the last eight bytes of memory used by the last eight screen lines are not used (64 bytes total). For example, the range of bytes from $478 to $47F (screen line 17) are never used in the screen display itself. These gaps in memory usage are called “screen holes,” and they have been reserved for use by hardware devices assigned to each slot. The assignments are as follows:

<table>
<thead>
<tr>
<th>Slot Number</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$478</td>
<td>$479</td>
<td>$47A</td>
<td>$47B</td>
<td>$47C</td>
<td>$47D</td>
<td>$47E</td>
<td>$47F</td>
</tr>
<tr>
<td></td>
<td>$4F8</td>
<td>$4F9</td>
<td>$4FA</td>
<td>$4FB</td>
<td>$4FC</td>
<td>$4FD</td>
<td>$4FE</td>
<td>$4FF</td>
</tr>
<tr>
<td></td>
<td>$578</td>
<td>$579</td>
<td>$57A</td>
<td>$57B</td>
<td>$57C</td>
<td>$57D</td>
<td>$57E</td>
<td>$57F</td>
</tr>
<tr>
<td></td>
<td>$5F8</td>
<td>$5F9</td>
<td>$5FA</td>
<td>$5FB</td>
<td>$5FC</td>
<td>$5FD</td>
<td>$5FE</td>
<td>$5FF</td>
</tr>
<tr>
<td></td>
<td>$678</td>
<td>$679</td>
<td>$67A</td>
<td>$67B</td>
<td>$67C</td>
<td>$67D</td>
<td>$67E</td>
<td>$67F</td>
</tr>
<tr>
<td></td>
<td>$6F8</td>
<td>$6F9</td>
<td>$6FA</td>
<td>$6FB</td>
<td>$6FC</td>
<td>$6FD</td>
<td>$6FE</td>
<td>$6FF</td>
</tr>
<tr>
<td></td>
<td>$778</td>
<td>$779</td>
<td>$77A</td>
<td>$77B</td>
<td>$77C</td>
<td>$77D</td>
<td>$77E</td>
<td>$77F</td>
</tr>
<tr>
<td></td>
<td>$7F8</td>
<td>$7F9</td>
<td>$7FA</td>
<td>$7FB</td>
<td>$7FC</td>
<td>$7FD</td>
<td>$7FE</td>
<td>$7FF</td>
</tr>
</tbody>
</table>

Since there really is no slot 0 on the Apple II GS, the first group of bytes can be used by any slot. This means that a card using these locations should not assume they will be maintained if another card also has to use them.

Generally speaking, it’s a good idea to never use the first group. The groups for slots 1 through 7 are considered private memory, and cards may use these locations to store any required data. For the printer and communications ports (slots 1 and 2), this can include baud rate, parity, or other information. The 80-column firmware, for example, which always looks like it’s in slot 3, stores its current horizontal cursor position in location $57B.

If you write your own programs that manipulate the horizontal cursor
Addressing Modes and Improved Printing

position, as does Program 10-3, you must control the contents of $57B (CH80 is the common assembly label) in addition to $24 (CH40).

The mouse firmware make extensive use of the screen holes for storing the current $x,y$ position, the clamping values, and more.

You may wonder why the slot/address assignments didn’t group all bytes for a given slot together, for example, $4F8$ to $4FF$ for slot 1. The reason was to make indexed addressing easier. A printer card, for example, could be placed in any slot.

```
TAY ; PUT SLOT # IN Y (1-7)
LDA $478,Y ; GET A BYTE
STA $4F8,Y ; PUT IT IN ANOTHER SCREEN HOLE
LDA $578,Y ; AND SO ON.
STA $5F8,Y
LDA $678,Y
STA $6F8,Y
LDA $778,Y
STA $7F8,Y
```

Based on the above, you may have noticed that Program 10-6 cleared all the invisible screen holes—not a wise thing to do. If you run the program with DOS 3.3 active and then do a CATALOG, you’ll hear the disk drive grind as it starts back up and discovers some of its information stored in the screen holes has been altered.

A better screen clear program uses the VTAB routine and indirect indexed addressing to clear just those bytes you see on the screen. Program 10-8 is the improved listing.

Notice that the VTAB routine sets up BASL (BASE address Low byte) with the address of the beginning of the line. We then use indirect indexed addressing to store the character at each character position on a given line.

There are two main loops in the program. Line 39 checks the $Y$ Register to keep it in the range of 0 to 39 for the horizontal character loop. Line 44 checks to see if the current line counter has reached the bottom of the screen. Remember that the horizontal and vertical counts go from 0 to 39 and from 0 to 23, respectively. BCC is thus the proper test to use, since this will branch as long as each counter is less than its prescribed limit.
Chapter 10

Program 10-8. Screen Clear Demo 1C

* SCREEN CLEAR PROGRAM #1C *
* MERLIN ASSEMBLER *

ORG $300

LINE EQU $06 ; WHAT LINE WE'RE ON
CHAR EQU $08
KYBD EQU $C000
STROBE EQU $C010
CV EQU $25 ; VERTICAL CURSOR POSN
VTAB EQU $FC22
BASL EQU $28 ; $28,29 = BASE ADDRESS

000300: AD 00 C0 READ LDA KYBD ; GET KYBD CHARACTER VALUE
000303: C9 80 CMP #$80 ; KEYPRESS?
000305: 90 F9 BCC READ ; NO, THEN TRY AGAIN.
000307: SD STA STROBE ; CLEAR KYBD STROBE.

000310: 64 06 INIT STZ LINE ; SET LINE = 0
000312: A5 06 FINDV LDA LINE ; GET LINE VALUE
000314: 85 25 STA CV ; VERTICAL CURSOR POSN
000316: 20 22 JSR VTAB ; CALCULATE BASE ADDRESS

000319: A5 08 CLEAR LDA CHAR ; GET CHAR TO 'CLEAR' TO
00031B: A0 00 LDY #$00 ; ZERO Y REGISTER

00031D: 91 28 LOOP STA (BASL),Y ; PUT CHAR IN MEMORY
00031F: C8 INY ; Y = Y + 1
000320: C0 28 CPY #40 ; END OF LINE?
000322: 90 F9 BCC LOOP ; NOPE: NEXT POSITION

000324: E6 06 NXTLN INC LINE ; LINE = LINE + 1
000326: A5 06 LDA LINE
000328: C9 18 CMP #24 ; DONE YET?
00032A: 90 E6 BCC FINDV ; NOPE

00032C: 80 D2 AGAIN BRA READ ; GET ANOTHER CHARACTER!

00032E: 60 DONE RTS ; ALL DONE!

--End Merlin-16 assembly, 48 bytes, Errors: 0
Addressing Modes and Improved Printing

Command Processing with Indexed Indirect Addressing

The final example for this chapter is a demonstration of how indexed indirect addressing can be used to easily process a command. The principle behind the operation is to use the command as the X register index into a table of entry points to the desired routines.

In the early example of indexed indirect addressing, a direct page address was used as the base address for a LDA instruction. For the JSR and JMP instructions, a complete 2-byte address can be used, like this:

JSR ($0300,X)

or

JMP ($0300,X)

What is needed now is a way to store, in the program itself, the addresses of the routines that we want to call. The program can then load the X register with the value for a command and jump directly to the subroutine. Program 10-9 is an example.

The first thing to notice is the use of a new assembler pseudo-op, DA (Defined Address) on lines 44-46. This instruction tells the assembler to create a pair of bytes in memory, as part of the object file, that have values equal to the label specified in the instruction. On line 44, the statement DA CMD1 tells the assembler to first evaluate the label CMD1, which in this assembly is equal to $32E (see line 48). The assembler then stores the address, low-order byte first, at the beginning of the data table, location $328. The next two DA statements store the address for CMD2 and CMD3. You can use the DA instruction in a program whenever you want to create two bytes of data that form a pointer to somewhere else in memory.

Now, let's look at the program as a whole. After first clearing the screen, the program gets a keypress and then checks for the Escape key. Lines 26-29 then check to see if the key is in the acceptable range—this program accepts keys 1, 2, and 3. The first test on lines 26, 27 compare against the value for 1. A BCC instruction will branch back if the value is less than 1.

Because the next test will be done with a BCS instruction, which tests for greater than or equal to, we need to compare with a value one larger than the last allowable value. Lines 28, 29 do this test.

At this point the accumulator holds the value $B1, $B2, or $B3. This must first be converted to a command number of 0, 1, or 2. Lines 31-33 subtract #$B1 to get a result in this range. The table is made up of two-byte
Chapter 10

groups; thus the proper entry points are stored at TABLE, TABLE+2 and TABLE+4 ($328, $32A and $32C). This means that the X register must be loaded with 0, 2, or 4 for the indirect JSR to work properly.

Lines 35,36 add the value for CMD to itself (equivalent to multiplying by 2), and then put this value in the X register. Line 40 then does the actual indirect JSR to the appropriate routine, which prints the characters A, B, or C, depending on which number key is pressed.

This may seem a little more complicated than the alternative method of a CMP and direct JSR, but for large numbers of commands you may find this method of using the indirect JSR more efficient.

**APW user please note.** In the APW assembler, lines 44-46 used the defined constant pseudo-op, and should look like this:

```
DC 12'CMD1'
DC 12'CMD2'
DC 12'CMD3'
```

Program 10-9. Command Processor Example

```
1 ******************************************************
2 * COMMAND PROCESSOR EXAMPLE *
3 *                  MERLIN ASSEMBLER                  *
4 ******************************************************

5
6 ORG $300
7
8 =C000 KYBD EQU $C000
9 =C010 STROBE EQU $C010
10
11 =FC58 HOME EQU $FC58 ; CLEAR SCREEN
12 =FDED COUT EQU $FDED
13
14 =0006 CMD EQU $06
15
16 000300: 20 58 FC ENTRY JSR HOME
17
18 000303: AD 00 C0 READ LDA KYBD ; GET KYBD CHARACTER VALUE
19 000306: C9 80 CMP #$80 ; KEYPRESS?
20 000308: 90 F9 =0303 BCC READ ; NO, THEN TRY AGAIN.
21 00030A: 8D 10 C0 STA STROBE ; CLEAR KYBD STROBE.
22
23 00030D: C9 9B CHECK CMP #$9B ; ESCAPE?
24 00030F: F0 2F =0340 BEQ DONE ; YES
25
26 000311: C9 B1 CMP #"1" ; 1ST ALLOWABLE CHARACTER
27 000313: 90 EE =0303 BCC READ ; NOPE TRY AGAIN
28 000315: C9 B4 CMP #"4" ; LAST CHAR + 1
```

202
Addressing Modes and Improved Printing

000317: B0 EA = 0303 29 BCS READ
30
000319: 38 31 SEC ; GET READY TO SUBTRACT
00031A: E9 B1 SBC #$B1 ; VALUE FOR 'I'
00031C: 85 06 STA CMD ; SAVE CMD VALUE
34
00031E: 18 35 CLC
00031F: 65 06 ADC CMD ; ACC = 2 * CMD VALUE
37
000321: AA 38 TAX ; PUT CMD IN X REGISTER
39
000322: FC 28 03 PROCESS JSR (TABLE,X) ; GO TO APPROPRIATE ROUTINE
40
000325: 4C 03 03 JMP READ ; BACK FOR MORE!
43
000328: 2E 03 TABLE DA CMD1 ; LOC OF ROUTINE #1
00032A: 34 03 DA CMD2 ; LOC OF ROUTINE #2
00032C: 3A 03 DA CMD3 ; LOC OF ROUTINE #3
47
00032E: A9 C1 48 CMD1 LDA #“A”
000330: 20 ED FD 49 JSR COUT
000333: 60 50 RTS
51
000334: A9 C2 52 CMD2 LDA #“B”
000336: 20 ED FD 53 JSR COUT
000339: 60 54 RTS
55
00033A: A9 C3 56 CMD3 LDA #“C”
00033C: 20 ED FD 57 JSR COUT
00033F: 60 58 RTS
59
000340: 60 60 DONE RTS ; ALL DONE!
61
000341: EB 62 CHK ; CHECKSUM FOR LISTING

--End Merlin-16 assembly, 66 bytes, Errors: 0
Chapter 11

Data Storage and Program Control
This chapter starts with a question: When your program does a JSR to some other part of memory, how does the 65816 microprocessor keeps track of where to return?

It’s possible that there is some as-yet-unmentioned register in the 65816 itself that could hold the returning address, but suppose your called subroutine does a JSR itself? With room for only one stored address, the microprocessor register would already be busy keeping track of the first address, and things would come to a screeching halt.

The Stack

The answer is to set aside an area of RAM in the computer itself to store a large number of temporary values. This area is called the stack, and, for Applesoft BASIC and simple related routines, it occupies all of page one ($100–$1FF) of memory. The area can actually be reassigned anywhere in the first 64K of memory (Bank 0), but you’ll see how to do that a little later.

The area is called the stack because it acts something like a physical stack of items. If you put three items in a stack, one at a time, you can’t get back to the first item until you remove the last two placed there. The stack in the Apple is the same way; and is often refered to as a last in, first out stack (LIFO). If you place the value 1, then 2, then 3 on the stack in memory, you get the values back in the reverse order, 3, 2, 1.

You’ve already used the stack in a program by using the JSR instruction. When the 65816 encounters a JSR in a program, it first determines the address of the instruction which follows JSR. This address is then put on the stack for temporary storage, and the JSR to the target address is done. When the RTS at the end of the subroutine is reached, the 65816 takes the stored return address off the stack and goes back to executing the program at the instruction which follows the JSR.

The stack can be used for more than maintaining JSRs however. You
can put the contents of any of the A, X, or Y registers on the stack with the appropriate push command: PHA (Push Accumulator), PHX (Push X), and PHY (Push Y). These place a copy of whatever value is in the referenced register onto the stack. Whether this consists of one or two bytes depends on the condition of the e, m, and x bits, which control register sizes.

To get a value off the stack, you use the corresponding pull instructions: PLA (Pulls Accumulator), PLX (Pull X), and PLY (Pull Y).

As with the push instructions, the number of bytes removed from the stack also depend on the settings of the e, m, and x bits.

You don’t have to use the same register to remove data from the stack as you used to put it there. The only crucial requirement is that the number of bytes put on the stack be the same as the number removed. Here’s a sample program that uses the stack:

```
LDA #"A" ; VALUE FOR "A" IN ACCUMULATOR.
PHA ; PUT VALUE (ONE BYTE) ON STACK.
PLX ; RETREIVE VALUE INTO X REGISTER.
STX $SBC ; PUT IT ON SCREEN.
```

The program begins by loading the accumulator with the single-byte ASCII value for the letter A. This is then pushed onto the stack. Next, PLX pulls a single byte from the stack, and puts it in the X register. It’s then stored in screen memory to make the byte visible.

**Keep It Balanced**

In using the stack, it’s of critical importance that you keep all pushes and pulls balanced. That is, if your routine pushes two bytes onto the stack, your routine should also remove those same two bytes before your final RTS. All information in the stack is organized only by its position there. There are no labels or other identifiers. When Applesoft BASIC does a CALL to a routine, the return address for your BASIC program is put on the stack. If your program looked like these three lines, the computer will probably crash or lock up when you run it:

```
LDA #"A" ; VALUE FOR "A"
PHA ; PUT ONE BYTE ON STACK
RTS ; RETURN TO WHERE?
```

This is because you have left a byte on the stack that the computer will use as a return address. Remember it’s expecting that the last two bytes on the stack are a return address from a JSR somewhere. You’ve now added a byte of your own, and have not removed it. Likewise, if you do a PHA with m=1 (8-bit mode = 1 byte pushed on stack), and then later you do a PLA with m=0 (16-bit mode = 2 bytes removed from stack), your final RTS won’t work because part of the
return address will be missing. Like left and right parentheses in a BASIC program, it's up to you to see that all pushes and pulls in a machine language program are balanced.

The Stack Pointer
The 65816 uses a pointer, called the stack pointer, to keep track of data currently on the stack. This pointer is abbreviated $S$ (Stack) and is another register within the 65816 itself (see Figure 11-1).

When the Monitor displays the registers following a BRK, or with Control-E, you can see the value of the stack pointer:

\[
\begin{align*}
A &= 00EF \\
X &= 0000 \\
Y &= 0000 \\
S &= 01DD \\
D &= 0000 \\
P &= B0 \\
B &= 00 \\
K &= 00 \\
M &= 0C \\
Q &= 80 \\
L &= 1 \\
m &= 1 \\
x &= 1 \\
e &= 1
\end{align*}
\]

For a routine called from Applesoft BASIC, the 65816 has the $e$ bit set to emulation ($e = 1$), and the stack is confined to page one ($100-1FF$). This is done by forcing the high byte of the stack pointer to $01$, and letting the low byte range from $00$ to $FF$.

In the native mode, however, the high byte can have any value in the range of $00$ to $FF$ also, which means the stack can be of any size and can be located anywhere in the first 64K (Bank 0) of memory.
Chapter 11

For efficiency purposes, the stack is operated by building down in memory, with the stack pointer indicating the next available memory location. For example, with nothing on the stack in the emulation mode (such as Applesoft BASIC routines), a diagram of the the stack would look like this:

$1FF: $?? ← S = $01FF

This shows nothing on the stack, and S is set to $1FF. At this point, the actual contents of $1FF can be anything.

Suppose now that the following program were executed:

8000: 20 58 FC JSR $FC58
8003: A5 C1 LDA #C1
8005: 8D BC 05 STA $5BC
8008: 60 RTS

At the point at which the program does the JSR to $FC58, for example, the stack would look like this:

$1FF: $80 High Byte = $80
$1FE: $02 Low Byte = $02 ($8002)
$1FD: $?? ← S = $01FD

Note that the address for the instruction following the JSR $FC58, $8003 minus one ($8002) has been placed on the stack. When the microprocessor resumes execution with the next RTS, it adds one to the address on the stack, and proceeds from there. This may not seem to be the cleanest possible system, but is presumably a result of the original processor design.

Using JSL and JSR. You’ll recall from Chapter 3 that there is also another form of the JSR instruction that will jump to any location in memory, JSL (Jump Subroutine Long). Because JSR has only a 2-byte operand, it can only jump to an address in the current bank of memory. JSL is a four-byte instruction, and it looks like this in a program:

JSL $E1A55C

This tells the microprocessor to go to the subroutine located at address $E1A55C. It can also be expressed as location $A55C in bank $E1. A routine called by a JSL must be terminated by a RTL (ReTurn from subroutine Long).

Whether you use a JSL or a JSR affects how many bytes are stored, and later retrieved, from the stack. For a JSL, three bytes are pushed onto the stack; a JSR pushes two bytes onto the stack. An RTL removes three; RTS removes two.
As an example, consider this program, running in Bank 0:

```
00/8000: 22 5C A5 E1 JSL $E1A55C
00/8004: A9 C1 LDA #C1
```

The stack will look like this while the routine at $E1A55C is being executed:

| $1FF | $00 | Bank = $00 |
| $1FE | $80 | High Byte = $80 |
| $1FD | $03 | Low Byte = $03 ($008003) |
| $1FC | ?? | ← S = $01FD |

**Setting the stack pointer.** There are a number of commands that can be used to set the stack pointer, or to transfer its value to another register for examination or manipulation. These are as follows:

- **TSX** Transfer Stack pointer to X register.
- **TXS** Transfer X register to Stack pointer.
- **TSC** Transfer Stack pointer to C (full) Accumulator.
- **TCS** Transfer C (full) Accumulator to Stack pointer.

Basically, your choices are to use the Accumulator or the X register to either receive from or send a value to the Stack pointer. For example, if, for some reason you wanted to set the Stack pointer to $1300, the following program would do it:

```
CLC
XCE ; SET NATIVE MODE IF NOT THERE ALREADY.
SEP #$20 ; 16-BIT ACCUMULATOR
LDA #$1300 ; ACC (C) = $1300
TSC ; SET S = $1300
```

This program assumes it is starting in emulation (e=1) mode, as would be the case if it were called from Applesoft BASIC. It’s not really a very realistic example, since there’s not much point to moving the stack for a routine called from Applesoft BASIC, but it shows that such a feat is possible. You could also have used the TXS command to set S to $1300 if desired.

**Stack Relative Addressing**

With just the push and pull instructions, once a byte was on the stack, you couldn’t access it until any bytes on top of it were removed. Fortunately, there is an addressing mode specifically for the stack that lets you retrieve any of the last 256 bytes pushed on the stack. It’s called the *stack relative* addressing...
mode, and looks like this:

```
BEGIN  LDA  #$0100  ; 16-BIT MODE
       PHA  ; PUT IT ON STACK
       LDA  #$0200  ; ANOTHER VALUE
       PHA  ; PUT IT ON STACK
GET    LDA  1,S  ; ACC WILL = $0200
       LDA  3,S  ; ACC WILL = $0100
```

In this addressing mode, you can retrieve a value off the stack by indicating the offset from the current stack pointer that you want to retrieve the data from. There are two things in particular to notice here.

The first is that the first byte stored on the stack is at relative position one. That is because the stack pointer itself (relative position zero) always points to the next available byte, not the last stored.

Second, remember to use the proper offset value. Because each push in the sample program was in the 16-bit mode, two bytes were put on the stack for each PHA. Therefore, the second LDA used an offset of 3 to get the second pair of bytes.

**Stack Relative Indirect Indexed Addressing**

There is an even more interesting addressing mode variation, called *stack relative indirect indexed*. It looks like this:

```
LDY  #2
LDA  (1,S),Y
```

This tells the 65816 to first go to the stack and get the address it finds at the first position on the stack (1,S). Then, it uses this as an indirect pointer to an address to which the offset in the Y register is added. This may seem rather convoluted, but it comes in very handy for following a JSR with data to be handled by a subroutine.

Program 11-1 demonstrates how to create a subroutine that prints whatever string follows the JSR to the routine. It works by first using the stack relative-indirect indexed mode. Because the return address for the JSR is on the stack when the print routine is called, that address can be used as a base address from which to read the characters that follow. The print loop itself is based on the print program from the last chapter. When the $00 at the end of the string is encountered, the print loop is terminated.

At this point, an RTS in the print routine would return to the byte right after the JSR, namely the beginning of the string just printed. To skip over this data, lines 36–43 add the length of the string just printed to the value for the return address. The result is then replaced on the stack so that when the RTS at
DONE is executed, program control resumes at PROGRAM, the next instruction after the end of the printed string.

This technique is not limited to just print routines. You can use this method to pass any kind of data to a subroutine. The main restriction is that you must make sure that the subroutine has a way of knowing exactly how many bytes are passed to it. If you add too much or too little to the return address, program control will resume either past the desired instruction or in the middle of the data to be passed, both of which will have unpredictable results.

Program 11-1. Stack Indirect Indexed Sample

```assembly
; ************
* STACK INDIRECT INDEXED SAMPLE *
* MERLIN ASSEMBLER *
; ************

ORG $300

-FC58 EQU $FC58
-FDED EQU $FDED
-0006 EQU $06

BEGIN JSR HOME

PRINT1 JSR PRINT
ASC "THIS IS THE FIRST STRING",SD,00
PROGRAM NOP

PRINT2 JSR PRINT
ASC "THIS IS THE SECOND STRING",SD,00
PROGRAM NOP

EXIT RTS

PRINT LDY #$01

LOOP LDA (1,S),Y
BEQ FIX
; 0 = END OF STRING
JSR COUT
; PRINT ASCII CHARACTER
INY
; Y = Y + 1
```

213
### Chapter 11

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00034B: 80 F6 0343 34</td>
<td>BRA LOOP</td>
<td>; NEXT CHARACTER</td>
<td></td>
</tr>
<tr>
<td>00034D: 84 06 36</td>
<td>FIX</td>
<td>STY LEN</td>
<td>; SAVE LEN OF STRING</td>
</tr>
<tr>
<td>00034F: A3 01 37</td>
<td>LDA 1,S</td>
<td>ADC LEN</td>
<td>; ADD TO LENGTH OF STRING</td>
</tr>
<tr>
<td>000351: 65 06 38</td>
<td>STA 1,S</td>
<td>ADC</td>
<td>; PUT BACK IN PLACE</td>
</tr>
<tr>
<td>000353: 83 01 39</td>
<td>STA 2,S</td>
<td>LDA 2,S</td>
<td>; GET HIGH BYTE OF RETURN ADDRESS</td>
</tr>
<tr>
<td>000355: A3 02 40</td>
<td>LDA 2,S</td>
<td>ADC #$00</td>
<td>; ADD CARRY IF NEEDED</td>
</tr>
<tr>
<td>000357: 69 00 42</td>
<td>ADC</td>
<td>STA 2,S</td>
<td>; PUT BACK IN PLACE</td>
</tr>
<tr>
<td>000359: 83 02 43</td>
<td>STA 2,S</td>
<td>LDA 2,S</td>
<td>; GET HIGH BYTE OF RETURN ADDRESS</td>
</tr>
<tr>
<td>00035B: 60 44</td>
<td>DONE RTS</td>
<td>; RETURN TO END OF STRING + 1!</td>
<td></td>
</tr>
</tbody>
</table>

---

**End Merlin-16 assembly, 92 bytes, Errors: 0**

### Push Effective Instructions

Along with the push instructions already described, there is a special group of instructions that push address data, rather than register contents, onto the stack. These instructions are **PEA (Push Effective Address)**, **PEI (Push Effective Indirect address)**, and **PER (Push Effective Relative address)**.

PEA pushes an absolute address onto the stack. It can also be used to push a data value. PEA always pushes two bytes onto the stack no matter what the setting of the \( e \) and \( m \) bits are (8-or 16-bit mode).

The following instructions would push the values \$0100 and \$0200 onto the stack:

PEA \$0100
PEA \$0200

This is equivalent to:

LDA #$0100 ; 2 BYTES IN 16-BIT MODE
PHA
LDA #$0200
PHA

PEA can be used to pass any constant or address to a subroutine using the stack. Notice that in the PEA instruction, the pound sign ( \# ) is not used in front of the value pushed on the stack.

PEI uses the contents of the indicated direct page locations as the data to put on the stack. This instruction:

PEI ($06) ; PUSH TWO BYTES ON STACK
Is equivalent to:

```
LDA $07 ; GET HIGH BYTE BYTE
PHA    ; PUSH ON STACK
LDA $06 ; GET LOW BYTE
PHA    ; PUSH ON STACK
```

Notice that PEI pushes the high byte (LABEL + 1) first, then the low-order byte (LABEL). Like PEA, PEI always pushes two bytes, regardless of the condition of e and m.

PER is the most exotic of the three instructions. It pushes a relative offset from the current point in the program to the referenced address. This is very much like the calculation done for a branch instruction. In your source listing, you specify the target address. However, the assembler calculates a relative distance, and uses this as the operand of PER in the actual assembly.

PER can be used to reference data when you don't know exactly where in memory your program will run. In previous chapters, we discussed the impact of JMPs and JSRs to absolute addresses in a program. If a program has a JSR, for example, to $340, and that program is moved to $8000, then the JSR $340 will jump to a non-existent program at that point. Likewise, load and store instructions like LDA and STA that address absolute memory location will access non-existent data if the program is moved from its original location.

One solution to this problem is to use the stack relative indexed indirect addressing mode and to use PER to put the correct address on the stack. Program 11-2 is another print program that will run at any location in memory.

On line 12, the 65816 first uses the offset value in the operand of PER and adds that to the current program counter, such as the address of that instruction. The result, namely the current address of DATA, is then pushed on the stack. Line 15 can then access that pointer on the stack.

When using any of the push instructions, you must remember to keep the number of bytes pushed and pulled from the stack balanced. Lines 21, 22 provide the balancing pulls in this example program.
Chapter 11

Program 11-2. PER Sample Print Demo

1. **********************************************
2. • 'PER' SAMPLE PRINT PROGRAM •
3. • MERLIN ASSEMBLER •
4. **********************************************

=FC58 6 HOME EQU $FC58
=FDED 7 COUT EQU $FDED

008000: 20 58 FC 10 BEGIN JSR HOME
008003: 62 10 00 12 PRINT PER DATA ; PUSH ADDRESS OF 'DATA'
008006: A0 00 13 LDY #$00 ; POSN OF 1ST CHARACTER
008008: 83 01 14
008009: FO 06 =8012 15 LOOP LDA (1,$),Y ; GET A CHARACTER TO PRINT
00800A: 00 06 =8012 16 BEQ FIX ; 0 = END OF STRING
00800C: 20 ED FD 17 JSR COUT ; PRINT ASCII CHARACTER
00800F: C8 18 INY ; Y = Y + 1
008010: 80 F6 =8008 19 BRA LOOP ; NEXT CHARACTER
008012: 68 20 21 FIX PLA ; FIX STACK (PULL 1 BYTE)
008013: 68 22 PLA ; PULL SECOND BYTE
008014: EA 23 24 PROGRAM NOP ; MISC. PROGRAM STUFF HERE...
008015: 60 25 26 EXIT RTS
008016: D4 C8 C9 D3 28 DATA ASC "THIS IS A TEST",8D,00
00801A: A0 C9 D3 A0 C1 A0 D4 C5
008022: D3 D4 8D 00

—End Merlin-16 assembly, 38 bytes, Errors: 0

The Direct Page Register: D

In the previous chapter, we noted that although the direct page is usually the first $100 bytes from $00 to $FF, this area can be reassigned by a running program for its own use. This means that there can be several different programs in the computer at the same time, each with its own direct-page area, thus eliminating potential memory-use conflicts.

Several programs in the computer at the same time may seem rather exotic, but even a running Applesoft BASIC program fits this category. Not only is Applesoft BASIC active, but usually ProDOS and certain Monitor routines are active, as well. These three were designed before the 65816 (for the 6502 microprocessor), so each was designed to specifically avoid using bytes used by one of the others. With the Apple IIIGS, you can design new programs without
Data Storage and Program Control

Program 11-3. PHD Example 1

```
SAVE PHD ; PUSH DIRECT PAGE ON STACK
PLA ; GET LOW BYTE (ASSUMES 8-BIT MODE)
STA TEMP ; SAVE IT SOMEWHERE
PLA ; GET HIGH BYTE
STA TEMP+1 ; SAVE IT
LDA #>PAGE ; GET HIGH BYTE OF NEW LOCATION
PHA ; PUT IT ON STACK
LDA #<PAGE ; GET LOW BYTE OF NEW LOCATION
PHA
PLD ; SET NEW DIRECT PAGE LOCATION
NOP ; YOUR PROGRAM HERE . . .
```

```
RESTORE LDA TEMP+1 ; GET OLD DIRECT PAGE LOC. HI BYTE
PHA ; PUT IT ON STACK
LDA TEMP ; GET OLD DP LOW BYTE
PHA
PLD ; RESTORE DP LOCATION
EXIT RTS ; DONE
```

having to worry about what direct-page bytes may be used by someone else.

There is a specific 65816 command, PLD (Pull Direct page register), that lets the programmer reassign the direct page to any location in the first 64K of memory ($0000 to $FFFF). This command can be very useful because page zero is heavily used by Applesoft BASIC, the Monitor, and ProDOS. There aren’t very many free bytes left. The example programs shown so far have used locations $06-$09 because these happen to be free, but what happens if you need more bytes? Using the PLD command is a way to define an entirely new direct page for your program’s own use, thus allowing 256 bytes just for you.

In the 65816 itself, there is a register, called the Direct Page Register, abbreviated D, that keeps track of the current direct-page location. This defaults to $0000, which indicates the range from $00 to $FF, but you can change D to define a new direct page any time you’d like (see Figure 11-1).

There is a hazard, however. Remember that, if you’re calling your routine from Applesoft BASIC, you must restore the direct page back to page zero when you’re finished or before you call any Applesoft BASIC, Monitor, or ProDOS 8 routine.

You can determine the current direct page at any time by using the command PHD (Push Direct page register). This pushes two bytes onto the stack that correspond to the current direct-page setting. For example, the program
Chapter 11

segment shown in Program 11-3 assumes that the routine is operating in the 8-bit mode (e and m = 1), as would be the starting case for a ProDOS 8 program or a routine called from Applesoft BASIC. However, if your program has shifted to the 16-bit mode (e and m = 0), the process is somewhat simpler as shown in the segment Program 11-4.

Program 11-4. PHD Example 2

```
SAVE PHD ; PUSH DIRECT PAGE ON STACK
PLA    ; GET DP LOC (ASSUMES 16-BIT MODE)
STA TEMP ; SAVE IT SOMEWHERE
LDA #PAGE ; GET NEW LOCATION
PHA    ; PUT IT ON STACK
PLD    ; SET NEW DIRECT PAGE LOCATION
NOP    ; YOUR PROGRAM HERE...
RESTORE LDA TEMP ; GET OLD DIRECT PAGE LOCATION
PHA    ; PUT IT ON STACK
PLD    ; RESTORE DP LOCATION
EXIT RTS ; DONE
```

The Data Bank and Program Bank Registers: B and K

In previous chapters, we've mentioned that the 65816 can run a program and access data in any of the memory banks. Remember that the bank is the first byte of the full three-byte address for a memory location. For example:

```
01/0300
```

would signify bank 1, address $300. For a running program, the Program Counter includes the Program Bank Register to make up the complete address for where the 65816 is currently executing an instruction (see Figure 11-2).

The Program Bank Register is a single byte register that determines which bank the currently active program is in. When the Monitor prints out the registers, like this:

```
A=00EF X=0000 Y=0000 S=01DD D=0000 P=B0
B=00 K=00 M=0C Q=80 L=1 m=1 x=1 e=1
```

the Program Bank is abbreviated K. The display here indicates the Program Bank Register is set to bank 0. The Program Bank Register is only modified when a JML, JSL, or RTL is executed. You can, however, determine it's current
setting with a stack instruction, PHK (Push program bank register). This instruction always pushes a single byte, namely the current Program Bank value, onto the stack.

The main purpose for the PHK command is to condition another important register, the Data Bank Register.

**The Data Bank Register: B**

So far, whenever you had a pair of instructions like

\[
\text{LDA MEM1} \\
\text{STA MEM2}
\]

it was taken for granted that the bytes accessed would all be in the same bank. In every program so far, this has been limited to bank 0. This need not be the
case, however. By changing the Data Bank Register, B, your program can run in one bank and manipulate data in another.

In the 65816 model (Figure 11-2), the Data Bank Register can be seen to be a single byte, like the Program Bank Register. Unlike the Program Bank Register, however, the Data Bank Register can be both read and set at any time. The two commands used are PHB (Push data Bank register) and PLB (Pull data Bank register). PHB puts the current value of the data bank register on the stack. This would be used to save the current setting, so your program could restore it before it quit.

For a routine called from Applesoft BASIC, BRUN, or a ProDOS 8 System file, both the Program and Data Bank Registers are already set to the default of bank 0. However, in a ProDOS 16 System file, the program can be loaded anywhere into memory. This means that when your program starts running, the Program Data Bank most likely won’t be set to the correct value. Fortunately, you do know that the Program Bank Register is correct—otherwise your program wouldn’t be running.

A program can properly set the Data Bank Register with the PLB command, which sets the Data Bank Register, B, by pulling a value off the stack:

```
BEGIN PHK ; PUSH PROGRAM BANK VALUE ON STACK
PLB         ; SET DATA BANK TO SAME VALUE
```

You’ll see these instructions at the beginning of almost every ProDOS 16 program. ProDOS 16 is described in more detail in Chapter 14.

**Bank 1 Access: An 80-Column Screen Clear**

If you try the Screen Clear Program 1C, Program 10-8, with 80 columns active, you’ll notice it only clears every-other character on the screen. That’s because the 80-column screen is made up of two $400-byte blocks—the first in bank 0, and the other (in the same address range) in bank 1. Thus, the 80-column screen is made up from memory with the addresses $00/400 to $00/7FF and $01/400 to $01/7FF. Screen Clear 1C only clears the memory in bank 0.

To create an 80-column screen clear, we can use the Data Bank Register to change which bank is being cleared (see Program 11-5). The main difference between Program 11-5 and Screen Clear 1C, Program 10-8, is on lines 42–56. Line 42 first pushes the current Data Bank value on the stack. Even though we know it’s equal to bank 0 for a routine called from Applesoft BASIC, this illustrates the technique for saving it if the need should arise. It also illustrates how the stack itself can be used to save a value temporarily. The value to be saved is pushed onto the stack, and then is pulled off later (see line 56) to restore it.

Lines 43–46 push the value 1 on the stack so the Data Bank Register can be set with a PLB. At this point, all 16-bit address access will be done in bank
1—not where the program is running in bank 0. It’s important to notice that we’re dealing with 16-bit addresses. What this means is that direct page and stack references are still in bank 0. This makes sense because the stack and direct page can only be in bank 0, so the Data Bank Register has no effect on these addresses or on instructions like LDA ($80),Y that reference them.

Because the Accumulator was used to push the value on the stack, the value for the character to clear to is no longer there. Line 48 resets the Accumulator to the proper value. Line 49 resets the Y register to 0. Because it has just fallen through the test on 39, the Y register is equal to 40 without the reset.

Lines 51–54 duplicate the loop used for bank 0, after which the Program Data Bank value is restored to its original value on line 56.

More About Multi-Bank Access

The Data Bank Register is not the only way to access other banks of memory from a running program. You’ve already seen how the LDA and STA instructions can use a long address to access any byte directly. As an exercise, you should try rewriting Screen Clear Program 1D using the Indirect Indexed Long instruction, STA [BASL],Y, to clear bank 0. Instead of using the Data Bank Register, modify BASL+2 to be the bank byte.

Program 11-5. Screen Clear Demo 1D

```
000300: AD 00 CO
000303: C9 80
000305: 90 F9 = 0300
000307: 8D 10 C0

000300: AD 00 CO
000303: C9 80
000305: 90 F9 = 0300
000307: 8D 10 C0
```
Chapter 11

00030A:C9 9B  23 CHECK CMP #$9B ; ESCAPE?
00030C:F0 31 -033F 24 BEQ DONE ; YES
25
00030E:85 08  26 STA CHAR ; SAVE CHARACTER VALUE
27
000310:64 06  28 INIT STZ LINE ; SET LINE = 0
29
000312: A5 06  30 FINDV LDA LINE ; GET LINE VALUE
000314: 85 25  31 STA CV ; VERTICAL CURSOR POSN
000316: 20 22 FC  32 JSR VTab ; CALCULATE BASE ADDRESS
33
000318:A5 08  34 CLEAR LDA CHAR ; GET CHAR TO 'CLEAR' TO
00031B:A0 00  35 LDY #$00 ; ZERO Y REGISTER
36
00031E:91 28  37 LOOP1 STA (BASL),Y ; PUT CHAR IN MEMORY
000320: A9 01  38 LDA #$01 ; BANK 1 VALUE
000322:48 45  39 PHA ; PUT IT ON STACK
000323:AB 46  40 PLB ; SET DATA BANK = 1
000325:A5 08  41 LDA CHAR ; PUT CHAR IN ACC AGAIN ...
000328:A0 00  42 LDY #$00 ; ZERO Y REGISTER AGAIN ...
43
00032A:91 28  44 LOOP2 STA (BASL),Y ; PUT CHAR IN MEMORY
00032C: C8  45 INY ; Y = Y + 1
00032E: C0 28  46 CPY #40 ; END OF LINE?
000330: 90 F9 =031D 47 BCC LOOP1 ; NOPE: NEXT POSITION
48
000332:B0 06  49 NXTLN INC LINE ; LINE = LINE + 1
000334: 90 C9  50 CMP #18 ; DONE YET?
000336:90 D5 =0312 51 BCC FINDV ; NOPE
52
000338:80 C1 =0300 53 AGAIN BRA READ ; GET ANOTHER CHARACTER
54
00033A: E6 06  55 NXTLN INC LINE ; LINE = LINE + 1
00033C: A5 06  56 LDA LINE
00033E: C9 18  57 CMP #24 ; DONE YET?
000340: B0 D5 =0312 58 BCC FINDV ; NOPE
59
000342: 80 C1 =0300 60 AGAIN BRA READ ; GET ANOTHER CHARACTER
61
000344: F6 60  62 DONE RTS ; ALL DONE
63
000346: 0C 67  64 CHK ; CHECKSUM FOR LISTING
65
—End Merlin-16 assembly, 65 bytes, Errors: 0
Chapter 12

Adding Machine Language Programs
Chapter 12

Adding Machine Language Programs

One of the first useful applications of your knowledge of assembly language programming can be the enhancement of your existing Applesoft programs. Such a combination of two languages is sometimes called a hybrid program, and many commercial programs are written by combining modules written in a variety of languages. In this way, particular functions in a program can be written in the language best suited to the task.

For example, if you had to write a program that asked the user for 10 names, just about the fastest way to do it would be with a short and simple Applesoft program that looked like this:

```
10 FOR I = 1 TO 10
20 INPUT N$(I)
30 NEXT I
```

This is much simpler than the equivalent program in assembly language. In cases where the tools already exist in Applesoft BASIC, and blinding speed is not a requirement, Applesoft BASIC is a completely acceptable solution.

However, if you had to sort a list of 1000 names, speed would become a concern, and it would be worth considering whether the job could best be done in assembly language.

As you've seen in previous chapters, it's possible to call a single machine language routine from within an Applesoft BASIC program with the CALL command. All of the example programs presented in this book so far assume that you're testing them with a CALL from Applesoft BASIC. As you have seen, by using the CALL statement, you can jump to any address in memory you want, and the routine there can perform virtually any function, subject only to the design of the routine.
Why Add Anything to Applesoft BASIC?

You might ask, "Why add any routines at all to Applesoft BASIC?" You might think that there are only two choices when writing a program: Write the entire program in Applesoft BASIC, or write the entire program in assembly language. As you've seen, though, there is another option. By adding special machine language routines to your Applesoft BASIC programs, you can solve those problems that Applesoft BASIC can't quite manage.

There are two fundamental reasons to add new functions to a BASIC program: speed and simpler programming. Applesoft BASIC has a little more than 100 commands like PRINT, VTAB, HOME, and so on. What happens when you want to sort a list? You have to combine a large number of these simpler commands to create a sort function. Usually, the resulting subroutine is not as fast as you'd like it.

To be fair, it's not that BASIC itself is really slow. As first discussed in Chapter 1, when your program has a statement like PRINT 5 * 3, the command is ultimately executed at machine language speed, because Applesoft BASIC is a collection of machine language routines. It's when things slow down that you have to put together hundreds of these commands to do a complete subroutine like sort a list, format dollars and cents numbers, and so forth.

If there were a way to replace all those BASIC commands with a machine language routine that sorted an array, and a way to add a command named SORT to BASIC, it would be just as fast as a pure machine language program—because it would be in machine language. Best of all, your BASIC program listing would look simpler itself, since all you'll see is a single CALL statement to do the sort or whatever. This isn't to say, of course, that you won't still have to write hundreds of lines of assembly language source code to create your sort subroutine—it just won't be visible as part of your BASIC program.

Passing Data Between Applesoft BASIC and Machine Language

All of the sample programs you've seen so far are independent of any Applesoft BASIC program that calls them. That is to say, there is no data in the form of variables passed either to or from the machine language routine that is called.

To create any really useful routines, you'll want to be able to pass variables back and forth between Applesoft BASIC and the machine language subroutines. The only problem is, how? You should suspect by now that an Applesoft BASIC program does not run, nor store data, in the same way as a machine language program. Then how can they talk to one another and share data?

Remember for a moment that at some point Applesoft BASIC executes a machine language routine to carry out a BASIC command, so it seems only logical that routines must already exist within Applesoft BASIC to convert the
Adding Machine Language Programs

Applesoft BASIC variables like X and A$ to a form that a machine language routine can deal with. To see how this is done, let’s first take a brief look at how an Applesoft BASIC program is actually stored in memory.

**The Internal Structure of Applesoft BASIC**

Consider this simple program:

```
10 HOME: PRINT “HELLO”
20 END
```

An interesting question arises: How does the computer actually store, and then later execute this program?

To answer that, we’ll have to go to the Monitor and examine the program data directly. Type in the BASIC program exactly as shown, then go to the Monitor with a CALL -151.

The first question to answer is, exactly where in the computer is the program stored? This can be found by entering the Monitor and typing in:

```
67 68 AF B0 (and pressing RETURN)
```

The computer should respond with:

```
00/0067:01-.
00/0068:08-.
00/00AF:19-.
00/00B0:08-.
*
```

The first pair of numbers is the pointer for the program beginning, bytes reversed of course. They indicate that the program starts at $801.

(You should be able to put this observation together with what you just learned about indirect addressing, and see why almost every important address value—like the beginning of a program, a variable list, and so forth—is stored as a pair of zero-page bytes. This allows us to access the data table, program, and so on with LDA ($67).Y, for example. By seeing how other programs are written, including Applesoft BASIC itself, you will get ideas about how to structure programs of your own.)

The second pair of bytes is the program end pointer, and they show it ends at $818. Using this information, let’s examine the program data by typing in:

```
801L
```

You should get:

```
1=m 1=x 1=L.Cbank (0/1)
00/0801: 10 08 BPL 080B {+08}
```
Although at first you might think this looks like a machine language program, those BRK instructions are a clue that it's really not directly executable code. Now type in:

801.819

This will give:

00/0801: 10 08 OA 00 97 3A BA-:::;
00/0808: 22 48 45 4C 4F 22 00-"HELLO".
00/0810: 16 08 14 00 80 00 00-::::::;
00/0818: 00 00-...

To understand this, let's break it down one section at a time. When the Apple stores a line of BASIC, it encodes each keyword as a single byte token. Literally, a token is any single marker of a larger quantity. In this case, the token is a single byte that stands for the five or six bytes that make up the Applesoft BASIC command. Thus, the word PRINT is stored as a $BA. This does wonders for conserving space. In addition, there is some basic overhead associated with packaging the line, namely a byte at the end to signify the end of the line, and a few bytes at the beginning of each line to hold information related to the length of the line, and also the line number itself.

To be more specific:

00/0801: 10 08 OA 00 97 3A BA-:::;
00/0808: 22 48 45 4C 4F 22 00-"HELLO".
The first two bytes of every line of an Applesoft BASIC program are an index to the address of the beginning of the next line. At $801,802 we find the address $810 (bytes reversed). This is where line 20 starts. At $810 we find the address $816. This is where the next line would start, if there were one. The double 00 at $816 tells Applesoft BASIC that this is the end of the BASIC listing.

The next information within a line is the line number itself:

```
00/0801: 10 08 OA 00 97 3A BA-.......;
00/0808: 22 48 45 4C 4C 4F 22 00-"HELLO".
00/0810: 16 08 14 00 80 00 00 00-.......;
00/0818: 00 00-..
```

The OA 00 is the two-byte form of the number 10, the line number of the first line of the Applesoft BASIC program. Likewise, the 14 00 is the data for the line number 20. The bytes are again reversed. After these four bytes, we see the actual tokens for each line.

```
00/0801: 10 08 OA 00 97 3A BA-.......;
00/0808: 22 48 45 4C 4C 4F 22 00-"HELLO".
00/0810: 16 08 14 00 80 00 00 00-.......;
00/0818: 00 00-..
```

All bytes with a value of $80 or greater are Applesoft BASIC keywords in token form. Bytes less than $80 represent normal ASCII data (letters of the alphabet, numbers, and so forth). Examining the data here, we see a $97 followed by $3A—$97 is the token for HOME, and $3A is the token for the colon. Next, $BA is the token for PRINT. This is followed by the quote ($22) and the text for HELLO (48 45 4C 4C 4F) and the closing quote ($22). Last of all, the 00 indicates the end of the line.

In line number 20, the $80 is the token for END. As before, the line is terminated with 00.

Again, remember how indirect addressing worked: Applesoft BASIC limits the length of each stored line to 255 bytes. This is so the Y register can be incremented from 0 to $FF as it scans the line in memory. The 0 at the end is used so that a BEQ test will detect the end of the line. If you were writing your own Applesoft BASIC interpreter, the code shown in Program 12-1 would be a good place to start.

When a program is executed, the interpreter scans through the data. Each time it encounters a token, such as the PRINT token, it looks up the value in a table to see what action should be taken. In the case of PRINT, this would be to output the characters following the token, namely HELLO.
This constant translation is the reason for the use of the term *interpreter* for Applesoft BASIC.

Machine code, on the other hand, is directly executable by the 65816 microprocessor so is much faster since no table lookups are required.

In Applesoft BASIC, a SYNTAX ERROR is generated whenever a series of tokens is encountered that is not consistent with what the interpreter expects to find.

Take the time to look over the program that reads an Applesoft BASIC line. You have learned all the commands and addressing modes necessary to write a program like this, and taking a moment to make sure you understand
what it's doing will help reinforce what you've learned already. And it will cement the new facts about Applesoft BASIC in place.

As another aside, armed with what you've already learned, and a chart of the Applesoft BASIC tokens, you could write a number of interesting programs, including a utility to renumber the line numbers in an Applesoft BASIC program or one to make a list of all the variable names used in a program.

**Passing Variables from Applesoft BASIC to Machine Language**

The easiest way to pass data to a machine language routine is to simply POKE the appropriate values into unused memory locations, and then to retrieve them when you get to your machine language routine. To illustrate this, we'll use the speaker location ($C030) and your knowledge of loops to write a simple tone routine.

To use this, enter and assemble Program 12-2, and BLOAD the final object code at $300. Then enter the accompanying Applesoft BASIC program, Program 12-3.

**Program 12-2. Sound Routine 1**

```
1 ********************
2 *      SOUND ROUTINE #1 *
3 *      MERLIN ASSEMBLER *
4 ********************
5
6 ORG $300
7
8 PITCH EQU $06
9 DURTN EQU $07
10 SPKR  EQU $C030

000300: A6 07
12 BEGIN LDX DURTN

000302: A4 06
14 LOOP  LDY PITCH ; STARTING VALUE FOR PITCH
000304: AD 30 C0
15 LDA  SPKR ; CLICK SPEAKER
16
000307: 88
17 DELAY DEY ; COUNTDOWN DELAY (PITCH)
000308: D0 FD =0307
18 BNE  DELAY
19
00030A: CA
20 DRTN DEX ; COUNTDOWN DURATION
00030B: D0 F5 =0302
21 BNE  LOOP ; CONTINUE NOTE
22
00030D: 60
23 DONE  RTS
24
```

—End Merlin-16 assembly, 14 bytes, Errors: 0
Chapter 12

This Applesoft BASIC program is used to call it:

Program 12-3. Sound Routine 1 Loader

10 INPUT “PITCH, DURATION “;P,D
20 POKE 6,P: POKE 7,D
30 CALL 768
40 PRINT
50 GOTO 10

The Applesoft BASIC program works by first requesting values for the pitch and duration of the tone from the user. These values are then POKEd into locations 6 and 7, and the tone routine is called. The tone routine uses these values to produce the desired sound, and then it returns to the calling program for another round (or sound).

The tone is created by two loops, one within the other. The outer loop, from LOOP to DURTN, controls how many times the entire note-producing cycle will last, thus controlling the length, or duration of the note. The inner loop, at DELAY, cycles through the delay loop to waste a little time each time the speaker is clicked. The faster you click the speaker, the higher the pitch of the note played. The two loops together create a system where you can control both the pitch of the note and its duration.

The technique used for passing the variables for pitch and duration works fine for limited applications, but having to POKE all the desired parameters into various corners of memory is not very flexible, and strings are nearly impossible. There must be an alternative.

Better Variable Passing

The key to passing variables to your own machine language routines is to work with Applesoft BASIC in terms of routines already present in the machine. That way, you can name the variable you’re dealing with right in the CALL statement.

The secrets here are the identities of two components of the Applesoft BASIC interpreter: TXTPTR (TeXT PoinTeR) and CHRGET (CHaRacter GET). TXTPTR and CHRGET are names given to pointers and routines already present in the computer, like COUT. It’s a good idea to use these names yourself in your own listings, but it’s not required to have the program work.

TXTPTR is the two-byte pointer ($B8, B9) that points to the next token to be analyzed. This is equivalent to the pointer PTR ($06,07) in the example line-scanning listing earlier. CHRGET ($B1) is a very short routine that actually resides on the zero page, and which will read a given token into the Accumulator. This is equivalent to the line-scanner itself. In addition to occasionally being called directly, many other routines use CHRGET to process a string of data in an Applesoft BASIC program line.
Adding Machine Language Programs

Program 12-4 is the revised tone routine; Program 12-5 is its Applesoft BASIC loader.

Program 12-4. Sound Routine 2

```
1                                                        ***************
2 * SOUND ROUTINE #2                                      *
3 * MERLIN ASSEMBLER                                       *
4                                                        ***************
5
6 ORG $300
7
8 =0006
9 PITCH EQU $06
10 =0007
11 DURTN EQU $07
12 =C030
13 SPKR EQU $C030
14
15 =E74C
16 COMBYTE EQU $E74C
17
18 000300: 20 4C E7                                          GETVARS      JSR COMBYTE GET COMMA & EXPRESSION
19 000303: 86 06                                             STX PITCH STORE VALUE < 256
20 000305: 20 4C E7                                          JSR COMBYTE GET NEXT COMMA & EXPRESSION
21 000308: 86 07                                             STX DURTN ; STORE VALUE < 256
22
23 00030A: A6 07                                            BEGIN LDX DURTN
24
25 00030C: A4 06                                            LOOP LDY PITCH ; STARTING VALUE FOR PITCH
26 00030E: AD 30 C0                                          LDA SPKR ; CLICK SPEAKER
27
28 000311: 88                                              DELAY DEY ; COUNT DOWN DELAY (PITCH)
29 000312: D0 FD =0311                                      BNE DELAY
30
31 000314: CA                                              DRTN DEX ; COUNT DOWN DURATION
32 000315: D0 F5 =930C                                      BNE LOOP ; CONTINUE NOTE
33
34 000317: 60                                              DONE RTS
35
36
37 End Merlin-16 assembly, 24 bytes, Errors: 0
38
39 Program 12-5. Sound Routine 2 Loader
40
41 10 INPUT "PITCH, DURATION ";P,D
42 20 CALL 768,P,D
43 30 PRINT
44 40 GOTO 10

This is a much more elegant way of passing the values, and it also doesn't require miscellaneous memory locations as such (although for purposes of simplicity the tone routine itself still uses the same zero page locations).

The secret to the new technique is the use of another new routine,
Chapter 12

**COMBYTE** ($E74C = COMma and BYTE processor). This is an Applesoft BASIC routine which checks for a comma and then evaluates whatever expression follows the comma in the Applesoft BASIC statement. It then returns the result as a single-byte value between $00 and $FF (0–255) in the X register.

It’s normally used for evaluating commands such as POKE, HCOLOR =, and others, but it does the job very nicely here. It also leaves TXTPTR pointing to the end of the line (or to a colon if there was one) by using CHRGET to advance TXTPTR appropriate to the number of characters following each comma. Note also that any legal expression such as $(X-5)/2$ can be used to pass the data.

To verify the importance of managing TXTPTR, try putting a simple RTS ($60$) at $300$. Calling this, you’ll get a SYNTAX ERROR—upon return Applesoft BASIC’s TXTPTR will be on the first comma, and the phrase ”,P,D” is not a legal Applesoft BASIC expression.

Now what about two-byte quantities (values greater than 256), or strings? There are other routines in Applesoft BASIC that can be used for these, as well.

The most concise way to explain the basic routines you’ll need is with another example, Program 12-6.

Program 12-6. Passing Variables

```
1 ******************************************************************************************
2 * BASIC TO ASSEMBLY LANGUAGE *
3 * VARIABLE PASSING DEMO. *
4 *
5 * &N1,N2,N$ *
6 * where N = Real variable, *
7 * with value up to 65535, *
8 * N2 is < 256, and N$ is any *
9 * string. *
10 *
11 * Note that N1, N2 and N$ can *
12 * also be expressions. *
13 *
14 * Merlin 8/16 Assembler *
15 *
16 ******************************************************************************************
17
18 BUFFER EQU $280 ; INPUT BUFFER FOR WORK AREA
19 MEM EQU $270 ; SOME STORAGE BYTES ($270-272)
20
21 FRMNUM EQU $DD67 ; EVALUATE NUMERIC EXPRESSION
22 GETADR EQU $E752 ; CONVERT PAC TO INTEGER
23 LINNUM EQU $50 ; $50,51
24
25 FRMEVL EQU $DD7B ; EVALUATE ANY EXPRESSION
26 FRESTR EQU $E5FD
27 INDEX EQU $5E ; $5E,5F
```
Adding Machine Language Programs

28  ILDIR  EQU  $E306 ; CHECK FOR DIRECT MODE
29
30  COMBYTE EQU  $E74C ; GET COMMA AND VALUE < 256
31
32  CHRGOT  EQU  $B7
33  CHKCOM  EQU  $DEBE ; CHECK FOR COMMA
34
35  8000: 20 06  E3 36 BEGIN  JSR  ILDIR ; MAKE SURE WE'RE NOT IN IMMED MODE
36  8003: 20 B7  00 37 JSR  CHRGOT ; CHECK CHAR AT TXTPTR
37  8006: C9 2C 38 CMP  #','; CHECK FOR COMMA
38  8008: D0 03 39 BNE  REAL ; NO COMMA
39  800A: 20  BE  DE 40 JSR  CHKCOM ; ADVANCE TXTPTR
40
41  800D: 20 67  DD 42 REAL  JSR  FRMNUM ; EVALUATE EXPRESSION
42  8010: 20 52  E7 43 JSR  GETADR ; CONVERT TO INTEGER
43  8013: A5 50 44 LDA  LINNUM ; LOW BYTE OF RESULT
44  8015: 8D 70  02 45 STA  MEM ; STORE IT
45  8018: A5 51 46 LDA  LINNUM+1 ; HI BYTE OF RESULT
46  801A: 8D 71  02 47 STA  MEM+1 ; STORE IT
47
48  801D: 20 4C  E7 49 SINGLE  JSR  COMBYTE ; CHECK COMMA AND EVALUATE
49  8020: 8E 73  02 50 STX  MEM+3 ; STORE IT
50  8023: 20  BE  DE 51 STRING  JSR  CHKCOM ; CHECK FOR NEXT COMMA
51  8026: 20 7B  DD 52 JSR  FRMEVL ; EVALUATE STRING EXPRESSION
52  8029: 20 FD  E5 53 JSR  FRESTR ; MAKE SURE IT'S A STRING AND
53  802C: A8 57 COPY  TAY ; PUT LEN IN Y REG
54  8030: 88 58 DEY ; FIX LEN FOR XFER LOOP
55  8032: B1 5E 59 LOOP  LDA  (INDEX),Y ; GET CHAR OF NAME STRING
56  8034: C0 FF 62 CPY  #$FF ; DONE WITH LOOP
57  8036: D0 F6 63 BNE  LOOP ; NOPE
58  8038: 60 65 DONE  RTS ; ALL DONE!
59  8039: 84 67 CHK ; CHECKSUM FOR LISTING
60
61
62
63
64
65
66

—End assembly, 58 bytes, Errors: 0

This program will read a large (greater than 255) floating-point number stored in a real variable, convert it to a two-byte integer, and then store it somewhere. We'll also read a string and a number value less than 256, and will store the data of these variables as well.
Chapter 12

Program 12-7. Passing Variables, Applesoft BASIC Loader

10 PRINT CHR$(4);"BLOAD VARIABLE.1.DEMO,A$768"
15 POKE 1014,0: POKE 1015,3: REM ($3F6,$3F7) = $300
20 N1 = 513: N2 = 127: N$ = "TEST"
30 & N1,N2,N$
40 PRINT PEEK (624) + 256 * PEEK (625)
45 PRINT PEEK (627)
50 FOR I = 0 TO 3
55 PRINT CUR$ (PEEK (640 + I));
60 NEXT I: PRINT

Program 12-7 includes a few new concepts. The first is to have the BASIC program automatically BLOAD the object code file when it first runs. This isn’t really all that new—Chapters 4 and 5 showed how to manually BLOAD a file after assembling it and saving the object code to disk. This program just adds the ProDOS BLOAD command as the first line of the program so you don’t have to load it manually each time you run the program. This is a good idea for programs that use added machine language routines.

Using &

The other new feature of the BASIC program is using the ampersand character ( & ). This is actually just a very limited version of the CALL command that you’ve already been using. The only differences are that the ampersand always does the equivalent of a CALL 1014 whenever it’s encountered in a program, and, because the ampersand always CALLs 1014, you must remember to POKE the proper destination address at 1014, 1015 before the ampersand is first used. Location 1014 is equivalent to $3F5, and if you go to the Monitor and list starting at $3F5, you see the following:

00/03F5: 4C 03 BE JMP BE03
00/03F8: 4C 00 BE JMP BE00
00/03FB: 4C 59 FF JMP FF59

You can see that all there is at $3F5 is a JMP command that you’re re-writing with your POKEs. This JMP is called a vector, and is used to direct the running machine language program to a new place in memory. After running the BASIC program, go to the Monitor again, and you’ll see $3F5 has changed to point to $300:

00/03F5: 4C 00 03 JMP 0300
00/03F8: 4C 00 BE JMP BE00
00/03FB: 4C 59 FF JMP FF59

The only advantage of using the ampersand is that it saves a few key-strokes compared to typing CALL 768 each time you call your routine. Other
than that, it's identical to a CALL statement.

The BASIC program itself simply loads the demo assembly language routine at $300 and then sets the ampersand vector to point to $300.

A Closer Look
Let's look at the assembly language program itself. In order of appearance, starting on line 36, here are the routines that are called, with a brief explanation:

Applesoft BASIC almost always uses the memory range from $200 to $2FF, called the input buffer, to store the characters you’re typing as part of an INPUT, or when you’re entering a new line of your program. Because any INPUT command writes into this area, you can’t store anything there, but it is available as a temporary area for your routines.

We’ll use this in our program as a temporary place to work with some string data, but the first thing to do is to make sure we’re not in immediate mode, typing in the very area we want to store to. JSR ILDIR ($E306 = check for Illegal DIRect error) does this. It will generate an Applesoft BASIC ILLEGAL DIRECT ERROR if our routine is not being called from within a running program.

When the ampersand is first called on line 30, the memory pointer (TXTPTR) is pointing to the first variable (or character) following the ampersand.

First, because a comma follows the CALL statement, TXTPTR must be advanced past the comma. However, because you might later decide to use the ampersand, we can’t be sure a comma will always be there. Therefore, the routine is made more versatile by beginning it with a call to the routine CHRGOT ($B7 = CHaRacter GOT). This is similar to CHRGET in that the accumulator is loaded with whatever character is pointed to by TXTPTR, but TXTPTR is not advanced after the read. This means you can check to see what’s there before putting everything in motion.

In this case, we check for a comma. If it’s there, we advance TXTPTR using CHKCOM ($DEBE = CHecK for COMma), which moves TXTPTR past a comma and at the same time checks to make sure that it was in fact a comma being skipped. If there isn’t a comma there (as will be the case if the routine is called with the ampersand syntax), it leaves everything alone and goes directly to the actual routine on line 36. All of this is so that you have the option of calling the routine either with a statement like CALL 768,N1,N2,N$ or &N1,N2,N$. Remember, if you use the CALL method, you don’t have to set up the ampersand vector.

FRMNUM ($DD67 = evaluate a FoRMula for a NUMber) is a routine which will evaluate any numeric value, variable, or expression, and which will
return the result in the Floating Point Accumulator (FAC). You don’t have to deal with the FAC directly, however, because there is another routine, GETADR ($E752 = \text{GET ADdResss}$), that will convert a floating-point number in the FAC into a two-byte integer, placing the result in LINNUM, LINNUM+1 ($\$50,51 = \text{LINeNUMber}$). You might correctly guess that these two routines are used by Applesoft BASIC for handling line numbers and addresses.

Our program then stores these two bytes in MEM, MEM+1 ($\$270,271$). During the evaluation, FRMNUM advances TXTPTR to the next character after the expression, in this case a comma before the next variable. FRMNUM will produce an automatic TYPE MISMATCH ERROR if you should try to use a string expression instead of a numeric expression in the first position.

Sometimes you may want to pass a relatively small number (less than 256); in this case you can use COMBYTE ($E74C$) to check the comma, advance TXTPTR, and then evaluate any numeric expression (remember an expression can be just single numbers and variables). It then returns the result in the X register. If the result is greater than 255, it will automatically generate an ILLEGAL QUANTITY ERROR in the calling Applesoft BASIC program. This saves you a lot of testing and message printing in the BASIC program. The result is stored at MEM+3 just to set it off from the first two data bytes stored earlier.

Finally, starting on line 52, we again check for an expected comma, but now we call the routine FRMEVL ($DD7B = \text{general FoRMula EVaLuation}$). FRMEVL is like FRMNUM, except that it will evaluate any expression, string or numeric. To check for a string, we need to call FRESTR ($E5FD = \text{FREe STRing}$), which checks to make sure we just evaluated a string and also leaves us with a pointer to the string data in INDEX, INDEX+1 ($\$5E,5F$) and the length of the string in the accumulator.

Lines 57 through 63 then use this information to copy the string to the middle of the input buffer (BUFFER = $280$), just to demonstrate how the data can be moved or manipulated.

**Passing Variables from Machine Language to Applesoft BASIC**

Now that you can send any kind of information you want to a machine language routine, how will that routine send its answers or other information back to Applesoft BASIC? The answer, again, is to use built-in routines. Since the statement $X = Y + 2 \times (5/2)$ exists in Applesoft BASIC, there must be routines that, after doing the calculation, convert the result to the Applesoft BASIC variable X.

Program 12-8 is a demonstration program that sends data from a machine language program back to an Applesoft BASIC program as real, integer,
and string variables. Program 12-9 is the Applesoft BASIC loader used with Program 12-8.

Program 12-8. Passing Variables to an Applesoft BASIC Program

```
1 **************************************************
2 * *
3 * MACH. LANG. TO FP VAR DEMO *
4 * SYNTAX: CALL 768,X%,Y,Z$ *
5 * WHERE: *
6 * X% = VALUE AT $270,271 *
7 * Y = VALUE AT $272,273 *
8 * Z$ = STRING AT $280+ *
9 * *
10 * MERLIN 8/16 ASSEMBLER *
11 * *
12 **************************************************
13 
14 FORPNT EQU $85 ; $85,86
15 STR EQU $280 ; STRING BUFFER
16 DATA1 EQU $270 ; 1ST VALUE
17 DATA2 EQU $272 ; 2ND VALUE
18 
19 CHRGOT EQU $B7
20 CHKCOM EQU $DEBE
21 PTRGET EQU $DFE3
22 CHKSTR EQU $DD6C
23 E3E9 EQU $E3E9
24 SAVD EQU $DA9A
25 ILDIR EQU $E306
26 
27 CHKNUM EQU $DD6A
28 EB9D EQU $EB9D
29 QINT EQU $EBF2
30 VARTYPE EQU $11 ; STR$=$FF, NUM=$00
31 NUMTYPE EQU $12 ; INT =$80, REAL = $00
32 LET2 EQU $DA63
33 LET3 EQU $DA6B
34 FAC EQU $9D
35 
36 BEGIN JSR ILDIR ; MAKE SURE WE'RE NOT IN IMMED MODE
37 JSR CHRGOT ; CHECK CHAR AT TXTPTR
38 CMP #',' ; CHECK FOR COMMA
39 BNE SEND1 ; NO COMMA
40 JSR CHKCOM ; ADVANCE TXTPTR
41 JSR PTRGET ; FIND OR CREATE VARIABLE
42 SEND1 JSR PTRGET ; FIND OR CREATE VARIABLE
43 STA FORPNT ; FOR USE BY LET2/LET3
44 STA FORPNT+1 ; AS ADDR OF VARIABLE DATA
45 STY FORPNT+1 ; AS ADDR OF VARIABLE DATA
46 
```
Chapter 12

008017: AC 70 02 48   LDY DATA1 ; LO BYTE OF RETURN VALUE
00801A: AD 71 02 49   LDA DATA1+1 ; HI BYTE
00801D: 85 9E 50     STA FAC+1
00801F: 84 9F 51     STY FAC+2
008021: A2 90 52     LDX #$90
008023: 25 12 53     AND NUMTYPE
008025: 20 9D EB 54   JSR GIVAYF2
008028: A5 12 55     LDA NUMTYPE
00802A: 30 06 =8032 56 BMI S1A ; INTEGER VARIABLE
00802C: 20 63 DA 57   JSR LET2 ; MAKE A REAL VAR
00802F: 18 58     CLC
008030: 90 06 =8038 59 BCC SEND2 ; ALWAYS BRANCH
008032: 20 F2 EB 61   S1A JSR QINT ; XVERT TO INTEGER
008035: 20 6B DA 62   JSR LET3 ; THAT'S ALL.
008038: 20 BE DE 64   SEND2 JSR CHKCOM ; MOVE TXTPTR PAST COMMA
00803B: 20 E3 DF 65   JSR PTRGET ; FIND OR CREATE VARIABLE
00803E: 20 6A DD 66   JSR CHKNUM ; VAR = NUM
008041: 85 85 67     STA FORPNT ; FOR USE BY LET2/LET3
008043: 84 86 68     STY FORPNT+1 ; AS ADDR OF VARIABLE DATA
008045: AC 72 02 70   LDY DATA2 ; LO BYTE OF RETURN VALUE
008048: AD 73 02 71   LDA DATA2+1 ; HI BYTE
00804B: 85 9E 72     STA FAC+1
00804D: 84 9F 73     STY FAC+2
00804F: A2 90 74     LDX #$90
008051: 25 12 75     AND NUMTYPE
008053: 20 9D EB 76   JSR GIVAYF2
008056: A5 12 77     LDA NUMTYPE
008058: 30 06 =8060 78 BMI S2A ; INTEGER VARIABLE
00805A: 20 63 DA 79   JSR LET2 ; MAKE A REAL VAR
00805D: 18 80     CLC
00805E: 90 06 =8066 81 BCC SENDSTR ; ALWAYS BRANCH
008060: 20 F2 EB 83   S2A JSR QINT ; XVERT TO INTEGER
008063: 20 6B DA 84   JSR LET3 ; THAT'S ALL.
008066: 20 BE DE 87   SENDSTR JSR CHKCOM ; MOVE PAST COMMA
008069: 20 E3 DF 88   JSR PTRGET
00806C: 20 6C DD 89   JSR CHKSTR
00806F: 85 85 90     STA FORPNT
008071: 84 86 91     STY FORPNT+1
008073: A9 80 93     LDA #$<STR ; LOC OF STRING BUFFER
008075: A0 02 94     LDY #$>STR
008077: A2 0D 95     LDX #$0D ; TERMINATOR CHARACTER
008079: 20 E9 E3 97   JSR MAKSTR ; CREATE STRING DESCRIPTOR
00807C: 20 9A DA 98   JSR SAVD ; PUT DATA AT A,Y INTO VARIABLE

240
Program 12-9. Applesoft BASIC Loader

0 REM ML TO FP VAR DEMO
5 PRINT CHR$(4);"BLOAD VARIABLE.2.DEMO,A$300"
10 A% = 200
20 B = 45123
30 C$ = "THIS IS A TEST"
40 POKE 624,A% - INT (A% / 256) * 256: REM LO BYTE
45 POKE 625, INT (A% / 256): REM HI BYTE
50 POKE 626,B - INT (B / 256) * 256: REM LO BYTE
55 POKE 627, INT (B / 256): REM HI BYTE
60 FOR I = 1 TO LEN (C$)
65 POKE 639 + I, ASC (MID$(C$,I,1))
70 NEXT I
75 POKE 639 + 1, 13: REM 'RETURN'
100 REM DISPLAY & CONVERT DATA
105 PRINT "ORIGINAL DATA: A% = ";A%,"B = ";B
110 PRINT "C$ = ";C$: PRINT
115 CALL 768,X%,Y,Z$
120 PRINT "NEW DATA: X% = ";X%,"Y = ";Y
125 PRINT "Z$ = ";Z$: PRINT
130 END

Just for variety, Program 12-9 uses the CALL method and shows how the variables follow a the CALL statement and a comma.

A Detailed Look

As with the first variable passing example, we first check for a comma, and move TXTPTR past it, if necessary.

Then, PTRGET ($DFE3 = PoinTeR GET), first called on line 43 of the listing, locates the variable in the Applesoft BASIC calling line. If a variable with that name has not yet been defined in the calling program, PTRGET creates one. The JSR CHKNUM ($DD6A = CHeck for NUMber) on line 44 verifies that this variable is a numeric—not a string—variable.

At this point, PTRGET has left the address of the variable data in the Accumulator and Y register, and lines 45 and 46 store this address in an Applesoft BASIC pointer, FORPNT ($85,86 = FORmula PoiNTer), to be used later when the variable data is actually sent back.

Lines 48 and 49 transfer the data, which for this demo has already been
stored at locations $270, 271 into the Floating Point Accumulator (FAC = $9D). The remaining lines (52–62) identify whether the receiving variable was an integer or a real variable type, and then they send the data back using the appropriate internal Applesoft BASIC routines accordingly.

For the purposes of this demo, the data is arbitrarily placed in memory by the Applesoft BASIC program. But, under normal circumstances, this data would have been already created by your assembly language program and could be located anywhere in memory.

Lines 64–84 basically duplicate the same variable passing procedure for the second variable. You might think the program should be written to send only integers back to the first variable, and only reals back to the second. In practice, however, you should use the routine as presented here so that you don’t have to remember whether to use a specific numeric variable type. The Applesoft BASIC demo uses both integers and reals just to prove it works, but the assembly language routine shown sends data to any numeric variable.

Lines 87–98 assume that string data, ending with a carriage return ($0D = 13), is in memory starting at $280. Again, PTRGET is used to identify the proper variable in the calling Applesoft BASIC program. This time, CHKSTR ($DD6C = CHeckSTRING) is used to make sure it’s a string variable. Sending the data back to Applesoft BASIC is even easier than it was for numbers—you need only load A and Y with the address of the beginning of the string, and X with the terminator character, and then call MAKSTR ($E3E9 = MAKe STRing). MAKSTR assumes that FORPNT has already been set up with the address of the variable data (lines 83 and 84), and it creates the preliminary string descriptor that SAVO ($DA9A = SAVe Descriptor) ultimately turns into a true Applesoft BASIC string variable.

The Applesoft BASIC program (Program 12-9) is designed only to prove the routine works. Lines 10–30 define three variables, lines 40–75 POKE the data into memory so our routine will have something to work with, and line 115 actually does all the work of returning the data, in the form of three new Applesoft BASIC variables, back to the calling program.

In normal practice, this data would have been created by the machine language subroutine itself, and no POKEs would be used.

How to Add Your Own Routines

Our examples so far have all loaded the routines at $300. This is because the area from $300 to $3CF is not used by Applesoft BASIC or ProDOS for anything, so is available for short routines.

As you create your own subroutines and programs, you’ll have to decide where you want them to be located in the machine. Where you put them depends on how big the routine is (number of bytes), how many routines you’re
Adding Machine Language Programs

using (there may be room for one short routine at $300), and whether the routine has to be put at a specific location in memory to work properly. Most large routines that are listed in computer magazines have to be loaded at a certain spot, and the article that explains them will give instructions on how to set things up.

However, if you write the routine yourself, or you want to move a published routine to a new location, you'll have to decide where to put it. First, let's look at how a normal Applesoft BASIC program sits in memory (see Figure 12-1).

Figure 12-1. Normal Applesoft BASIC Program Memory Map

When an Applesoft BASIC program is running, it uses all of the memory from $800 (2048 decimal) to $9600 (38400 decimal). The program itself starts at $800. LOMEM: defaults to the end of the BASIC program and determines where Applesoft BASIC will start storing all the variable names and numeric variable data that your program uses. HIMEM: defaults to just below the part of memory used by ProDOS (usually $9600), and the bytes for string data start here in memory, building down as new strings are defined. As more data is defined in the program as a whole, the two groups, numeric and string data, grow toward the middle of memory.

If you want to add a routine to the system, there are three usual places to put it, as shown in Figure 12-2.

Figure 12-2. Applesoft BASIC with Routines Memory Map
Chapter 12

If the routine is less than 200 bytes long, you can put it at $300 (768). Starting at $3D0 are some important ProDOS pointers, so your program can’t be larger than the space between $300 and $3D0 if you want to put it here.

Another option is to find the end of your program in memory, to BLOAD the routine there, and to set LOMEM: to a larger value to create a gap for your program. The listing for a BASIC program that did this might look like this:

```basic
10 PRINT CHR$(4);"BLOAD ROUTINE,A";
   PEEK(175) + 256 * PEEK(176)
20 REM ASSUME ROUTINE IS 200 BYTES LONG...
30 LOMEM: PEEK(175) + 256 * PEEK(176) + 200
```

Note that no variables are used in the calculations. Since you’ll be changing LOMEM:, no variables may be used before LOMEM: is changed, since they will lose their values as soon as LOMEM: is moved (Applesoft BASIC won’t know where to look for the old values).

Locations 175, 176 ($AF,BO hex) are Applesoft BASIC’s pointer to the end of the program, and PEEKing here tells us where the end of the BASIC program is.

This last technique requires that your program is position independent (there are no JMPs or JSRs in the program to other parts of itself). You’ll recall from the discussion in Chapter 6 that if there is a JMP or JSR in the program to another location within the program, and you move the program’s position in memory to any other location other than the ORG address, it will crash when you try to run it.

If you want to write programs that are not position independent, you can always pick an arbitrarily large memory address for the ORG and BLOAD, like $6000 (the top of hi-res page two), and set LOMEM: to, say, 28672 ($7000 hex). $1000 hex is about 4000 bytes, which should be enough room for many routines. You just have to make sure your BASIC program never gets so large as to go past $6000, or that your machine language routine isn’t larger than $1000 bytes.

All this might seem like a lot to learn at once, but you’ll find that including the lines necessary to pass variables back and forth fairly simple to use, once you try them a few times.

By using these techniques in your own subroutines that work with Applesoft BASIC programs, I think you’ll find it’s a lot easier to use what you’ve learned about assembly language right away.
Chapter 13

ProDOS
Chapter 13

ProDOS

ProDOS, which stands for *Professional Disk Operating System*, is a set of routines loaded into RAM when the computer first starts up from the disk. The routines are responsible for opening and reading files and for writing data to the disk. Without a disk operating system in the machine, the Apple II GS doesn’t inherently know how to read a disk. You can prove this to yourself by turning on the computer with no disk in the drive, and then pressing RESET to go to Applesoft BASIC without starting up a disk. If you type CATALOG at this point, you’ll get a SYNTAX ERROR because the computer, without a DOS loaded, doesn’t know anything about talking to the disk.

There are two main versions of ProDOS, ProDOS 8 and ProDOS 16. ProDOS 8 was originally designed for the Apple IIe and IIc, which use the 65C02 microprocessor. In this processor, the Accumulator, registers, and memory are always accessed one byte at a time, so these are called 8-bit machines. ProDOS 8 will also run on the Apple II GS, and is the required disk operating system for Applesoft BASIC.

ProDOS 16 is the latest incarnation of ProDOS, and it’s designed specifically for the Apple II GS. It will not run on a IIe or IIc, nor with Applesoft BASIC. Fortunately, the user doesn’t have to worry about which operating system is required by a given program, because the Apple II GS automatically loads the correct operating system when a program is loaded.

**ProDOS 8**

To get an idea of how ProDOS 8 is set up in the computer, let’s first consider a hypothetical disk, with ProDOS 8 (named PRODOS), BASIC.SYSTEM, and an Applesoft BASIC program named STARTUP on the disk. This disk can be booted on either a II GS, IIe, or IIc.

When the disk is first booted, the disk drive hardware is preprogrammed to read in the first two blocks of data on the disk (blocks 0 and 1) and to execute this data as a program. This very small program then reads in more information, and the process continues until an entire application program (in this case STARTUP) is loaded and running. To the original designers of disk systems,
this process of one bit of code loading some more that loads even more had an almost magical feeling to it, reminding them of the phrase *pulling yourself up by your bootstraps*. This inspired the term *booting* for starting up a disk.

The entire ProDOS 8 boot process goes like this: First, the code stored on the disk in blocks 0 and 1 is loaded and run. This miniprogram looks for a file in the main (root) directory named PRODOS. If it can’t find exactly that name, or if some other problem occurs while it’s trying to load and run that file, you’ll get the Unable to Load ProDOS error message.

On our hypothetical disk, the file PRODOS is ProDOS 8, and the disk operating system itself is loaded into the computer. This isn’t the end, though. ProDOS 8 doesn’t understand CATALOG either. In fact, it has no user interface at all, let alone a friendly one. It has various routines that access the disk, but they all expect to be called with a JSR from some machine language program. Many Applesoft BASIC programs, on the other hand, are written with statements like PRINT CHR$(4);“CATALOG”, that the programmer expects will make something happen.

ProDOS 8 is called a *kernel*, in that it’s a central part of the operating computer, but it contains none of the niceties that make up a complete program. To bridge this gap, ProDOS 8 on our disk loads and runs BASIC.SYSTEM, which in turn looks for a file called STARTUP that is an Applesoft BASIC file.

BASIC.SYSTEM need not be run. ProDOS 8 in general just looks for the first file in the main directory whose name ends in .SYSTEM and whose file type is SYS ($FF). Although BASIC.SYSTEM was the system file on our hypothetical disk, it need not be. A program like AppleWorks, or any other application program, has its own system file as the first system file on the disk, and it doesn’t need BASIC.SYSTEM or any Applesoft BASIC program at all on the disk.

### Writing .SYSTEM Files

So, how do you write your own stand-alone program that’s a SYSTEM file under ProDOS 8? It’s actually quite simple. There are only a few things you need to know:

1. SYSTEM files are automatically loaded starting at memory location $2000. If your program is not totally position independent (if it has even *one* internal address reference), it must be assembled with an ORG $2000.
2. The filetype of the object file must be $FF (SYS). In Merlin, the DSK and TYP directives will take care of this. In APW, you must use the MAKEBIN and FILETYPE commands.
3. Instead of ending with an RTS, you must execute a call (JSR) to ProDOS, telling it you’re finished. This Quit Call lets ProDOS return control to a program selector or any other program that started your application. You should
never end a program by telling the user to reboot, or, worse yet, by clearing memory and forcing a reboot.

The discussion that follows assumes you're familiar with ProDOS directory structure (root directories and subdirectories), the meaning of different file types, and so forth. This discussion will briefly introduce you to some of the basic rules for working with ProDOS, and it will give you an idea of what's involved in dealing with ProDOS from assembly language.

**The Machine Language Interface: MLI**

To make using ProDOS from machine language as easy as possible, the designers came up with a standard procedure to make any given call (JSR) to ProDOS. This common calling point (and way of using ProDOS) is called the Machine Language Interface (MLI). The general procedure is:

1. Call the MLI entry point. This is always done with a JSR $BF00.
2. The JSR is immediately followed by three bytes that encode the desired ProDOS command. The first byte is the command code itself, followed by two more bytes that form a pointer to a larger block called the parameter list or parameter table (parm table). The parameter list may contain things like the name of the file to open, and how many bytes to read. When the ProDOS call returns, it always resumes execution immediately after these three bytes. Program 11-1, the Stack Indirect Indexed Sample, showed a way of displacing the return address after a JSR, and this is similar to what ProDOS does with an MLI call.
3. After the three data bytes, there is usually a BCS ERROR instruction. ProDOS sets the Carry if an error occurs, and it stores the error code in the Accumulator. The BCS test will then branch to your own error routine to take appropriate action. If no error occurs, the carry will be clear and your program will continue. Notice that you are on your own now, and no longer enjoy the support of Applesoft BASIC and BASIC.SYSTEM to handle errors and print messages. If you want a CATALOG in your own SYSTEM type program, you’ll have to write it yourself.

**The Simplest Program**

The first SYSTEM program (Program 13-1) we’ll write that uses ProDOS 8 will be just about the simplest possible: It will clear the screen, ask for a keypress, and then do the Quit Call. The command code for a Quit is $65.
Program 13-1 can be tested by going to a program selector like the Apple Program Launcher, DeskTop, or your favorite program selector, and then choosing P8.SYSTEM in the selector menu. When you press a key, you automatically should be returned to the selector program. The name of this file, P8.SYSTEM, has no connection to the file P8 on your Apple IIgs System Disk, other than we are just using the characters P8 in these examples to indicate that a ProDOS 8 file is being used.

Notice that every ProDOS 8 SYSTEM file starts up in the total 8-bit mode, just as would a routine you were calling from Applesoft BASIC.

Looking at the source listing, lines 14 and 15 show how to create a file with a specific file type in Merlin. The assemble-to-disk feature was described in Chapter 4. The TYP (file TYPE) directive on line 15 should immediately follow the DSK directive; it tells Merlin what the file type of the object file should be. For a SYS file, this should be $FF. Other ProDOS file types are listed in Table 13-1. Remember that when assembling to disk, you no longer have to save the object file manually at the main menu of the Merlin assembler. Of course, you will still have to save the source file after you’ve typed it in.

The program itself starts by clearing the screen, printing a message, and then waiting for a keypress. I used the BIT instruction (see Chapter 9) more for variety than necessity.

On line 30 is the JSR to MLI (defined with an Equate at the beginning as $BF00). The DFB $65 on line 31 is a Merlin pseudo-op that assembles as only a single byte, in this case the $65 for the Quit command. DFB (DeFine Byte) is a data storage pseudo-op, similar to HEX, and is used whenever you want to put a single byte, whose value may be defined by a label, in your program. Line 32 uses the DA pseudo-op, first mentioned in Chapter 10, that stores two bytes in address form (low-order byte first). In this case the address is a pointer to the parameter list (PARMTBL at $2021). These bytes are followed by a BCS to an error routine, which, if everything is assembled correctly, will never be taken. The BRK instruction on line 34 is likewise unused since a no-error Quit will never actually return. In an assembly listing, it is best to use the instruction BRK $00 (although just BRK with no operand is legal). This is because the monitor always lists a BRK plus the byte which follows. If you use just BRK, you’ll have trouble listing your programs with the monitor. The purpose of the BRK is to halt the program if something is wrong. In this case, the only probable cause is if you mistyped the quit instruction, and ProDOS returned an error code for some other command.

The structure of all ProDOS parameter lists, such as PARMTBL on line 36, follow a general pattern. The first byte in the table indicates the number of parameters in the list, not the number of bytes. Since Quit, or any other
ProDOS command, already knows how many parameters it requires, this first byte is mainly for internal error checking by the MLI handler.

The byte on line 37 is a single-byte code for the type of quit call being made. For the standard, no-frills ProDOS 8 quit call, this should be zero. This will return you to the program launcher (if one is used) directly. There is also another type of quit, called the ProDOS 8 Enhanced Quit, that can be used if you boot ProDOS 16 first. In our example of the disk where PRODOS is ProDOS 8, this is not an option. The next five bytes are not used for a standard ProDOS 8 quit, so are set to zero.

It is essential that you use precisely the correct number of bytes, and appropriate values, in the complete MLI call. If you use a DA on line 31 or a DFB on line 32, or if you use the incorrect structure in the parameter list, very strange and definitely unpredictable things will happen. Ninety percent of all ProDOS programming errors stem from incorrect MLI calls. Consider yourself warned.

APW users. If you're using APW, Program 13-2 is the corresponding listing for P8.SYSTEM. After the ASML assembly (see Chapter 5), first type MAKEBIN P8.SYS. This will convert the output EXE file into a binary file that loads at $2000. Then type FILETYPE P8.SYSTEM SYS to change the filetype to $FF (SYS). The file can then be launched from a program selector as described earlier.

The main differences between the APW version and the Merlin listing are as follows: First, to make sure the high bit is set on the print statements, the APW has the assembler directive MSB ON. APW also assumes that everything starts off in 16-bit mode. Since this is a ProDOS 8 file, we must include the directives LONGA OFF and LONGI OFF to tell the assembler that the microprocessor will be in the 8-bit mode. If this step is omitted, LDY #$00, for example, would be assembled as A0 00 00 (three bytes) instead of A0 00 (two).

Finally, APW doesn’t have the DA or DFB pseudo-ops, but instead uses the generic DC instruction, which is qualified by the leading characters in front of the operand. Leading characters 11 tell it to assemble a single byte; 12 specifies two bytes. Be very careful when constructing your MLI calls—it is very easy to use the wrong byte length and mess up the entire call.

The APW’s assembly is two bytes longer than Merlin’s. This is because APW assembled the BRK instructions as two-byte instructions, whereas Merlin assembled them as just one byte. Because a BRK stops program execution no matter what follows it, the use of one or two bytes is pretty much a programmer preference. You can force a two-byte BRK instruction in Merlin by including the byte you want used as the second byte, as in

BRK $00

251
Chapter 13

Program 13-1. Simple P8 System File

```
*********** SIMPLE P8 SYSTEM FILE ***********
+ SIMPLE P8 SYSTEM FILE +
+ MERLIN ASSEMBLER +
*********** ***********

=BF00 6 MLI EQU $BF00
=FDFO 7 COUT EQU $FDFO
=FC58 8 HOME EQU $FC58
=C000 9 KYBD EQU $C000
=C010 10 STROBE EQU $C010

ORG $2000
DSK P8.SYSTEM
TYP $FF ; SYSTEM FILE TYPE

002000: 20 58 FC START JSR HOME ; CLEAR SCREEN

002003: A0 00 PRINT LDY #$00 ; INIT Y-REG
002005: B9 2B 20 LOOP LDA MSSG,Y ; GET CHAR TO PRINT
002008: F0 06 =2010 BEQ GETKEY
00200A: 20 F0 FD JSR COUT ; PRINT IT
00200D: C8 INY ; NEXT CHAR
00200E: D0 F5 =2005 BNE LOOP ; WRAPAROUND PROTECT

002010: 2C 00 C0 GETKEY BIT KYBD ; KEYPRESS?
002013: 10 FB =2010 BPL GETKEY ; NOPE
002015: 2C 10 C0 BIT STROBE ; CLEAR KEYPRESS

002018: 20 00 BF QUIT JSR MLI ; DO QUIT CALL
00201B: 65 DFB $65 ; QUIT CODE
00201C: 22 20 DA PARMTBL ; ADDRESS OF PARM TABLE
00201E: B0 09 =2029 BCS ERROR ; NEVER TAKEN
002020: 00 00 34 BRK $00 ; SHOULD NEVER GET HERE . . .

002022: 04 36 PARMTBL DFB 4 ; NUMBER OF PARMS
002023: 00 DFB 0 ; QUIT TYPE: 0 = STD QUIT
002024: 00 00 38 DA $0000 ; NOT NEEDED FOR STD QUIT
002026: 00 DFB 0 ; NOT USED AT PRESENT
002027: 00 00 40 DA $0000 ; NOT USED AT PRESENT

002029: 00 00 42 ERROR BRK $00 ; WE'LL NEVER GET HERE?

00202B: D0 CC C5 C1 44 MSSG ASC "PLEASE PRESS A KEY ->",00
00202F: D3 C5 A0 D0
002033: D2 C5 D3 D3
002037: A0 C1 A0 CB
00203B: C5 D9 A0 AD
```

252
Program 13-2. Simple P8 System File for APW

0001 0000 **********************************************
0002 0000 *
0003 0000 SIMPLe P8 SYSTEM FILE *
0004 0000 *
0005 0000 APW ASSEMBLER *
0006 0000 **********************************************

KEEP PS SYSTEM
MSB ON
LONGA OFF
LONGI OFF
ORG $2000

MAIN START

MLI EQU $BF00
COUT EQU $FDF0
HOME EQU $FC58
KYBD EQU $C000
STROBE EQU $C010

ENTRY JSR HOME

PRINT LDY #$00 ; INIT Y-REG
LOOP LDA MESSG,Y ; GET CHAR TO PRINT
GETKEY JSR COUT ; PRINT IT

ENTRY JSR HOME

GETKEY BIT KYBD ; KEYPRESS?
BPL GETKEY ; NOPE

ENTRY JSR HOME

QUIT JSR MLI ; DO QUIT CALL
DC 11'$65' ; QUIT CODE
PARMTBL DC 12'PARMTBL' ; ADDRESS OF PARM TABLE
BCS ERROR ; NEVER TAKEN
BRK $00 ; SHOULD NEVER GET HERE . . .

PARMTBL DC J1'4' ; NUMBER OF PARMS
DC J1'$00' ; QUIT TYPE: 0 = STD QUIT
DC J12'$0000' ; NOT NEEDED FOR STD QUIT
Chapter 13

The Enhanced ProDOS 8 Quit

If ProDOS 8 has been started up by the normal ProDOS-16 boot process (described in greater detail in the next chapter), there is another quit option available, called the ProDOS 8 Enhanced Quit. In this command, you can specify the pathname of the program you wish to quit to. In this way, your program itself becomes a program launcher. The specified program can be either a ProDOS 8 or ProDOS 16 system file (SYS or S16).

The only changes that need to be made to our original program to demonstrate this are to change the quit type code from $00 to $EE, and to change the two bytes following to a pointer to the pathname for the program we want to run next. In the next example, program 13-3, we'll assume the you have the file P8.SYSTEM, which you assembled earlier, on the disk in the same directory as P8.LAUNCHER.

When you start up this program from a program selector, it will first prompt you for a keypress and then will quit by running P8.SYSTEM. Then, when P8.SYSTEM does its quit call, control will return back to the program selector that launched P8.LAUNCHER.

Notice that lines 46–48 define a string with a leading length byte. ProDOS uses a standard protocol that expects every string it deals with to begin with a length byte. This is sometimes also called a PASCAL-format string, from an obvious heritage.

P8.LAUNCHER should give you some ideas as to how you could create your own ProDOS menu program that presented the user with a list of programs to run. By changing the pointer on line 38, or by rewriting the string itself, you can create the pathname for any file you wish.

Program 13-3. ProDOS 8 Launcher Demo

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>* PRODOS 8 'LAUNCHER' DEMO *</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>* MERLIN ASSEMBLER *</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>***********************</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>***********************</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>*************************</td>
</tr>
<tr>
<td>6</td>
<td>MLI</td>
<td>EQU $BF00</td>
</tr>
<tr>
<td>7</td>
<td>COUT</td>
<td>EQU $FDF0</td>
</tr>
</tbody>
</table>
End Merlin-16 assembly, 89 bytes, Errors: 0
Chapter 13

**ProDOS File Types**

Table 13-1 is a list of file types. It shows some values in use under ProDOS 8 and ProDOS 16. New file types can be defined by Apple at any time, so the list is subject to additions.

Table 13-1. ProDOS 8 Filetypes

<table>
<thead>
<tr>
<th>File Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00</td>
<td>BAD</td>
<td>Uncategorized file</td>
</tr>
<tr>
<td>$01</td>
<td>TXT</td>
<td>ASCII text file</td>
</tr>
<tr>
<td>$04</td>
<td>BIN</td>
<td>General binary file</td>
</tr>
<tr>
<td>$08</td>
<td>FOT</td>
<td>Graphics screen file</td>
</tr>
<tr>
<td>$0F</td>
<td>DIR</td>
<td>Directory file</td>
</tr>
<tr>
<td>$19</td>
<td>ADB</td>
<td>AppleWorks Data Base file</td>
</tr>
<tr>
<td>$1A</td>
<td>AWP</td>
<td>AppleWorks Word Processor file</td>
</tr>
<tr>
<td>$1B</td>
<td>ASP</td>
<td>AppleWorks Spread Sheet file</td>
</tr>
<tr>
<td>$1C-$AF</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>$B0</td>
<td>SRC</td>
<td>APW source file</td>
</tr>
<tr>
<td>$B1</td>
<td>OBJ</td>
<td>APW object file</td>
</tr>
<tr>
<td>$B2</td>
<td>LIB</td>
<td>APW library file</td>
</tr>
<tr>
<td>$B3</td>
<td>S16</td>
<td>ProDOS 16 application program file</td>
</tr>
<tr>
<td>$B4</td>
<td>RTL</td>
<td>APW runtime library file</td>
</tr>
<tr>
<td>$B5</td>
<td>EXE</td>
<td>ProDOS 16 shell application file</td>
</tr>
<tr>
<td>$B6</td>
<td></td>
<td>ProDOS 16 permanent initialization file</td>
</tr>
<tr>
<td>$B7</td>
<td></td>
<td>ProDOS 16 temporary initialization file</td>
</tr>
<tr>
<td>$B8</td>
<td>NDA</td>
<td>New Desk Accessory</td>
</tr>
<tr>
<td>$B9</td>
<td>CDA</td>
<td>Classic Desk Accessory</td>
</tr>
<tr>
<td>$BA</td>
<td></td>
<td>Tool set file</td>
</tr>
<tr>
<td>$BB-$BE</td>
<td></td>
<td>Reserved for ProDOS 16 load files</td>
</tr>
<tr>
<td>$BF</td>
<td></td>
<td>ProDOS 16 document file</td>
</tr>
<tr>
<td>$C0-$EE</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>$EF</td>
<td>PAS</td>
<td>Pascal area on a partitioned disk</td>
</tr>
<tr>
<td>$F0</td>
<td>CMD</td>
<td>ProDOS 8 CI added command file</td>
</tr>
<tr>
<td>$F1-$F8</td>
<td></td>
<td>ProDOS 8 user-defined files 1-8</td>
</tr>
<tr>
<td>$F9</td>
<td></td>
<td>ProDOS 8 reserved</td>
</tr>
<tr>
<td>$FA</td>
<td>INT</td>
<td>Integer BASIC program file</td>
</tr>
<tr>
<td>$FB</td>
<td>IVR</td>
<td>Integer BASIC variable file</td>
</tr>
<tr>
<td>$FC</td>
<td>BAS</td>
<td>Applesoft BASIC program file</td>
</tr>
<tr>
<td>$FD</td>
<td>VAR</td>
<td>Applesoft BASIC variables file</td>
</tr>
<tr>
<td>$FE</td>
<td>REL</td>
<td>Relocatable code file (Merlin)</td>
</tr>
<tr>
<td>$FF</td>
<td>SYS</td>
<td>ProDOS 8 system program file</td>
</tr>
</tbody>
</table>
Other ProDOS 8 MLI Commands

There are a total of 26 ProDOS 8 MLI commands. These are shown in Table 13-2.

Table 13-2. ProDOS 8 MLI Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alloc_Interrupt</td>
<td>$40</td>
<td>Place a pointer to an interrupt-handling routine into the system-interrupt vector table.</td>
</tr>
<tr>
<td>Dealloc_Interrupt</td>
<td>$41</td>
<td>Remove pointer from system-interrupt table.</td>
</tr>
<tr>
<td>Quit</td>
<td>$65</td>
<td>Quit current program back to another system program.</td>
</tr>
<tr>
<td>Read_Block</td>
<td>$80</td>
<td>Read a data block (512 bytes) from the disk.</td>
</tr>
<tr>
<td>Write_Block</td>
<td>$81</td>
<td>Write a data block to disk.</td>
</tr>
<tr>
<td>Get_Time</td>
<td>$82</td>
<td>Read current time using ProDOS built-in routine.</td>
</tr>
<tr>
<td>Create</td>
<td>$C0</td>
<td>Create a new file or directory.</td>
</tr>
<tr>
<td>Destroy</td>
<td>$C1</td>
<td>Remove name from directory.</td>
</tr>
<tr>
<td>Rename</td>
<td>$C2</td>
<td>Change name of file.</td>
</tr>
<tr>
<td>Set_File_Info</td>
<td>$C3</td>
<td>Set file's type and all other associated information (dates and so forth).</td>
</tr>
<tr>
<td>Get_File_Info</td>
<td>$C4</td>
<td>Read directory information entry for a file.</td>
</tr>
<tr>
<td>On_Line</td>
<td>$C5</td>
<td>Get slot, drive and volume name of one or all active volumes.</td>
</tr>
<tr>
<td>Set_Prefix</td>
<td>$C6</td>
<td>Set pathname to be used as prefix.</td>
</tr>
<tr>
<td>Get_Prefix</td>
<td>$C7</td>
<td>Get current prefix.</td>
</tr>
<tr>
<td>Open</td>
<td>$C8</td>
<td>Prepare a file to be read from or written to.</td>
</tr>
<tr>
<td>Newline</td>
<td>$C9</td>
<td>Specify character that terminates a file read, such as a carriage return.</td>
</tr>
<tr>
<td>Read</td>
<td>$CA</td>
<td>Read any number of bytes into memory from a file.</td>
</tr>
<tr>
<td>Write</td>
<td>$CB</td>
<td>Write any number of bytes from memory into a file.</td>
</tr>
<tr>
<td>Close</td>
<td>$CC</td>
<td>Finish file access. Update file directory entry if necessary.</td>
</tr>
<tr>
<td>Flush</td>
<td>$CD</td>
<td>Like a close, but doesn't release file buffers.</td>
</tr>
<tr>
<td>Set_Mark</td>
<td>$CE</td>
<td>Change current byte position in file.</td>
</tr>
<tr>
<td>Get_Mark</td>
<td>$CF</td>
<td>Get current byte position in file.</td>
</tr>
<tr>
<td>Set_EOF</td>
<td>$D0</td>
<td>Set length of file.</td>
</tr>
<tr>
<td>Get_EOF</td>
<td>$D1</td>
<td>Get length of file.</td>
</tr>
<tr>
<td>Set_Buf</td>
<td>$D2</td>
<td>Assign new location of input/output buffer for file.</td>
</tr>
<tr>
<td>Get_Buf</td>
<td>$D3</td>
<td>Get current location of input/output buffer of an open file.</td>
</tr>
</tbody>
</table>

An assembly language program uses ProDOS by using a combination of the appropriate commands to accomplish a given task. Program 13-4 uses the ProDOS 8 MLI system. It displays the contents of a text file on the screen. It's very simplistic. It offers no way to catalog a disk or to determine the names of the volumes online. It's error handling is minimal at best. However, it does present a working example of a program that opens a file, reads and displays the data, and then closes the file and exits with the standard Quit call.
It also shows how, in the course of writing an actual application, other issues become as important as the program itself. Such issues include error handling, the user interface, and concerns about the system state when your program starts up. Anyone that has written a commercial program can tell you that the user interface and error handling can take as much or more time and program code as the primary functions of the program itself.

In addition, this program introduces quite a number of new concepts in programming style, assembler pseudo-ops, and more.

After you've typed in the program, be sure to check the checksum value generated by the assembly on line 173. The byte in your program should match the listing, CHK = $89. The program is longer than any presented so far, and verifying the checksum will help avoid program bugs caused by typographical errors.

The program operates in general by first asking for the name of the file to be dumped. If the name includes a slash (/) as the first character of the name, it will use the input as the complete pathname, and use the volume indicated. If the name doesn't include a slash, it will append in the given name to the current prefix, and it will try to open that file. If the file is not found, or there is any error in opening and reading the file, the ProDOS MLI error code will be printed, and you can try again. A Monitor routine, PRBYTE ($FDDA = "PRint BYTE"), is used to print the error code as a hex number. This routine prints the contents of the Accumulator when called.

As the file is displayed, you can start and stop the text scrolling by pressing Control-S. You can exit the program by typing QUIT for the filename.

When you run the program, it will make some difference how you actually start up the program. If you BRUN the file from BASIC without having ever specifically set the prefix from BASIC, there will be no default prefix when FDUMP.SYS runs. Hence you'll have to include the volume name for the file you want to examine. This is because BASIC.SYSTEM does not specifically set the internal ProDOS pathname when it runs. On the other hand, if you run FDUMP.SYS from a program selector like the Program Launcher or DeskTop, these programs will set the internal ProDOS prefix when they launch the system file, and the prefix will be set to whatever volume and directory where the file itself is located.

A Closer Look

Now let's take a closer look at Program 13-4. The first thing to notice is a new initialization routine that you'll be seeing a lot of in the remainder of the listings in this book. When you call a routine from BASIC, if something goes wrong, and the program BRKs in the Monitor somewhere, you can always
press RESET or Control-C to get back to BASIC and try it again.

With a ProDOS system file, things are a little different. Now there’s no way to cleanly exit back to the program launcher without executing the ProDOS Quit command. The SETQUIT routine at the beginning of this program sets up some insurance in the way of the Control-Y vector, which is set to jump to our Quit code. If your program should break in the Monitor, you may be able to recover control by pressing Control-Y while in the Monitor.

In fact, once you get this program working, you may want to try deliberately placing a BRK instruction in the listing—perhaps around line 66—and then try pressing Control-Y in the Monitor to verify that the technique works. This will come in very handy as we move into ProDOS 16 and the tools where a BRK is even more likely, and when avoiding having to reboot the entire machine will be very helpful.

Once the Control-Y vector is set up, the very next thing the program does is check a memory location, $C01F = RD80COL (ReaD 80-COLumn status), which tells it whether the 80-column display is active. When the 80-column display is active, bit 7 of RD80COL will be set (BMI will work).

The reason this is required is because it’s possible to run FDUMP.SYS from a program selector that is running in 80 columns, and to have the program start up in 40 columns. This is a function of the program selector and the Apple IIgs, rather than FDUMP.SYS itself. The problem is, if a SYSTEM program is run from an 80-column display, and comes up in 40 columns, the screen width byte, $21 (WNDWDTH = WiNDow WiDTH) will still be set to 80 columns. This in turn means that text will not be printed correctly on the screen.

To prevent all of this, the program first checks to see if we’re in 40 columns (80 columns not active). If we are, the program stores the correct width, 40, in WNDWDTH.

Line 36 clears the screen and prints a prompt message. You’ll notice that the PROMPT section uses something new, a local label. Local labels are temporary labels you can use in a source listing for the destination of branches and loops so you don’t have to keep thinking of new names.

In a small program, branching back to LOOP is fine. As the program gets larger, you can probably use the labels LOOP2 and LOOP3, but after awhile it’s rather pointless. Readable labels exist only to add meaning to an entry point. If the meaning is obvious, you may want to consider using a local label. A local label is designated by a colon (:) followed by a number between 1 and 9. Local labels are remembered by the assembler only between real (global) labels. For example, if there is a label like CHAR on line 41, the assembler won’t know where to assign the BNE :1 on line 43, because the label CHAR
would be between the branch and the target local label. Restrictions and syntax for local labels vary by assembler, so you should read your assembler manual to find out all the particulars.

In the PROMPT print loop, you might also notice that we use a BNE at the bottom of the loop on line 43. As the Y register is incremented, line 40 is already testing for the end of the string, marked by a zero. Logically, we could have used a BRA or JMP on line 43. However, in the interest of possible debugging, a BNE is used instead of BRA or JMP so that if a zero was somehow left out of the string text (MSSG1), the loop would terminate when Y wrapped around to $00. If BRA or JMP were used, the loop would go forever with an omitted zero, and the program would seem to hang up.

After the prompt, line 45 uses a Monitor routine, GETLN2, which will do the equivalent of an INPUT command for you. It even supports a cursor and editing with the arrow keys. Believe me, this is not something that would be fun to have to write yourself. A JSR to the Monitor is much easier.

GETLN2 returns when the user presses Return, and the name entered is in the input buffer, $200 to $2FF. The X register contains the length.

Now for the next tricky part. Later on, we're going to tell ProDOS where the pathname typed in is located. Right now, it starts at location $200. Remember that ProDOS expects every string to begin with a length byte. What we need to do is to rewrite the pathname in the input buffer with a length byte at the beginning. Lines 46–53 illustrate a very direct solution to this. Remember that when indexing a string of bytes, the value for the length is usually one unit too large for accessing the last byte of the string.

For example, suppose starting at $200, you have the characters A at $200, B at $201, and C at $202. The string has a length of 3. Assume we'll use indexed addressing of the form LDA $200,X to access each character. You can see that if X = 3, we will be accessing byte $203 ($200 + 3), which is actually the fourth, and non-existent character, of the string. Normally, in a loop that just scans a string in the input buffer, the length, as such, is used as an upper limit to tell you when to stop, by going one byte too far.

Back to our routine. We can use the fact that $200,X will start at one byte past the end of the current string as a method to move the entire string to the right one byte. Line 48 reads LDA INBUF-1,X. This is a neat trick for accessing one byte previous to INBUF,X. The first time through the loop, LDA INBUF-1,X will pick up the last character of the string, and STA INBUF,X will move it to right one byte.

This will continue until the entire name has been shifted. At that point, location $200 is now empty; there we can store the length, which has been
saved on the stack. Look this over carefully until you’re confident you understand exactly how it works. It’s not enough to memorize a hundred or so assembly language commands—you must also start to learn how to combine, manipulate, and use them to accomplish your own programming goals.

The next step is to see if the user typed QUIT. A quick check is made on line 55 to see if the input string was four characters long. If it wasn’t, there’s no reason to do an exact check. It’s true that machine language is so fast that this test is more aesthetic than necessary, but its purpose is to illustrate further techniques of programming.

CHK2 on lines 58–64 actually tests the input string to see if it’s QUIT. There are a few new tricks here as well. First, look at the definition of MSSG1 on lines 160–163. Although a simple message like this could have been defined with one ASC instruction, breaking it up like this lets us assign a label to specify the word QUIT. Otherwise, the characters would have to be stored elsewhere a second time for the testing loop at CHK2. The LABEL-1,X addressing mode is again used. Now that the string has been moved to the right one character in the input buffer, the characters are now found at $201$ to $201 + X$.

This is nice, because we can now test for X reaching zero as it is decremented. If it isn’t immediately obvious why this is an advantage, consider the more traditional loop to scan the input buffer. In this case, we’ll pretend we want to convert an input string to entirely uppercase:

```
START JSR GETLN2 ; GET INPUT STRING
         ; DATA @ $200+, LEN IN X-REG.
DEX    ; CORRECT X FOR INDEXED ADDRESS
LOOP   LDA $200,X ; GET A CHARACTER
       ORA #$DF ; CONVERT TO UPPER CASE
       STA $200,X ; PUT IT BACK
       DEX ; X = X -1
       CPX #$FF ; WAIT FOR WRAPAROUND
       BNE LOOP ; STILL IN THE LOOP
DONE   RTS ; THAT’S ALL!
```

Notice how the X register must first be decremented to make the indexed addressing work out. Then, the end-of-loop test must look for a wraparound from $00$ to $FF$. You can’t do a BNE test, because then you’d leave out the last pass of the loop for $X = 0$ (first character of the buffer). You can’t use a BMI (another way to test for $FF$ by looking for the high bit set), because if the starting length of the string is greater than $7F$ (127 characters), the high bit will set when you start, and you’ll never loop back.

All this is avoided in the CHK2 routine, because the string has been moved up, to $201 + X$. This makes it easy to pick up each character and then use a BNE loop test. The only problem now is that the data for QUIT at WORD
Chapter 13

runs from WORD to WORD+3. No problem. By addressing it as CMP WORD-1,X, the addresses will match up properly, and the check routine will work.

Now for the actual ProDOS part of the program. OPEN on line 66 calls the ProDOS Open command. The three-byte MLI data block contains $C8 (the Open command), and a pointer to PARMTBL2, where the specifics of what file to open and where to store the data are kept. Look ahead to lines 147–150 to examine this table.

Line 147 holds the value 3, for the number of parameters for the Open command. Line 148 holds the pointer to $200 where the pathname of the file to open is stored. Line 149 tells ProDOS where a 1024-byte working buffer has been assigned. It will use this to read in each block from the disk. Line 150 reserves room for a reference number byte that ProDOS will use to make sure everybody is talking about the same file.

After the call (line 70), if the Carry is set, indicating a ProDOS error has occurred, a message will be printed that includes the ProDOS MLI error code, and the program will jump back after a keypress to ask for a new pathname.

Assuming there is no error in the Open command (Table 13-3 lists the error codes), lines 69 through 74 then read in 255 bytes at a time. Because the parameter table for both a Read and Close (which will be used shortly) are so similar, the same table can be used for both calls. The only consideration that must be made is to customize the beginning number-of-parameters value at the beginning of the table for each call. Lines 69,70 do this by storing a 4, which is the number of parameters for a Read table, at the beginning of PARMTBL3.

In looking at PARMTBL3 (lines 142–146), notice that a different buffer area is used for the data to actually be read from the file. DOSBUF is a 1024-byte buffer that ProDOS uses to manage the file it’s reading. The information in that buffer is not directly accessed by an application, but rather is requested using the Read command. The Read command itself must specify a separate buffer (BUFFER in this case). The buffer must be at least as large as the number of bytes specified to be read by the Read command parameter table. Our BUFFER is $100 bytes long.

The ProDOS buffer, DOSBUF, must also begin at a page boundary (even multiple of $100 such as $2000, $2100, and $2200), so we use another Merlin pseudo-op, DS (Defined Storage). This pseudo-op is used whenever you want to set aside a large block of bytes within your program without having to use a lot of instruction, such as HEX and DA. Merlin allows a special form of DS (DS followed by a Z) which pads the object file with empty bytes until the next page boundary. At that point ($2200 in our program), BUFFER is defined. BUFFER itself didn’t have to be on a page boundary, but its length of $100 makes it compatible with sandwiching between the DSZ instruction and
DOSBUF. Notice also that DOSBUF is just a label without an associated DS (or anything for that matter). Normally, all the empty bytes specified by a DS instruction are also saved to disk when the object file is saved. Since DOSBUF is at the end, and we don’t need to save the empty bytes as part of the file, line 164 accomplishes what we need: It assigns DOSBUF an address.

If there is no error on the Read command, lines 76–83 print out the characters read. At some point, an error ($4C = End of File) will be generated when the end of the file is reached. This error is specifically tested for, and a branch to CLOSE is done. Lines 90–96 rewrite the first byte of PARMTBL3 to correspond to the Close command parameter list, and then execute the Close command.

You might think that the Read command would generate an error (End-of-File error) when it reads the last few bytes of the file and reaches the end of file marker, since it will be rare when the length of the file will be an exact multiple of the number of bytes you’re requesting for each read.

Fortunately, Read is a little more sophisticated than that. Part of the Read parameter table is NUMREAD, which returns the actual number of bytes read from the file. When Read reaches the end of a file, it returns the number of bytes successfully read in the NUMREAD position, and it does not generate an End-of-File error. It is only the next or any successive attempts to read the file that will generate an error. Thus, you don’t have to worry about any special case handling to print the last few remaining characters in a file after an End-of-File error has occurred.

Finally, a PRESS A KEY prompt is printed, and the program goes back to the beginning.

It is important that you try to understand each part of this program. All ProDOS programming uses MLI calls and techniques like those used in this example. It may seem complicated when you’re reading the explanation, but it will begin to make sense as you read and reread the source listing. Also, remember this: Assembly language programs of any substance usually involve very lengthy listings. Remember—you’re trying to build skyscrapers out of very little building blocks, and it takes a lot of blocks to make a single wall, let alone the entire building. The trick is to stand back and look at the main labels for each routine and to try to get the big picture of what’s going on.

**Error Codes**

ProDOS MLI error codes are not the same as Applesoft DOS error codes, so Table 13-3 may be helpful in testing the program.


Chapter 13

Table 13-3

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$01</td>
<td>Invalid MLI command number was used.</td>
</tr>
<tr>
<td>$04</td>
<td>Incorrect number of parameters in PARMTBL.</td>
</tr>
<tr>
<td>$25</td>
<td>ProDOS Interrupt Table is full (not relevant to this example).</td>
</tr>
<tr>
<td>$27</td>
<td>Disk I/O error, such as open door, or bad disk.</td>
</tr>
<tr>
<td>$28</td>
<td>No device connected. You removed the drive while nobody was looking.</td>
</tr>
<tr>
<td>$2E</td>
<td>A disk with an open file was removed from the drive.</td>
</tr>
<tr>
<td>$40</td>
<td>Invalid pathname syntax (illegal characters).</td>
</tr>
<tr>
<td>$42</td>
<td>No buffers available. Too many files open (more than eight).</td>
</tr>
<tr>
<td>$43</td>
<td>File not open. Wrong reference number, or you tried to read a file without opening it.</td>
</tr>
<tr>
<td>$44</td>
<td>Subdirectory not found. Wrong name used.</td>
</tr>
<tr>
<td>$45</td>
<td>Volume not found. Wrong name used.</td>
</tr>
<tr>
<td>$46</td>
<td>File not found. Wrong name used.</td>
</tr>
<tr>
<td>$47</td>
<td>Duplicate filename. You’ve tried to create a new file with the name that is already in use (not relevant to this example).</td>
</tr>
<tr>
<td>$48</td>
<td>Disk is full.</td>
</tr>
<tr>
<td>$4A</td>
<td>File itself is not in a ProDOS format.</td>
</tr>
<tr>
<td>$4B</td>
<td>File type mismatch. Since the example doesn’t check for a specific file type, this isn’t likely.</td>
</tr>
<tr>
<td>$4C</td>
<td>End of data. No more data in file. EOF.</td>
</tr>
<tr>
<td>$4D</td>
<td>Range error. Occurs when Set-Mark is used for a position past the end of the file.</td>
</tr>
<tr>
<td>$4E</td>
<td>File locked. The file access bit in the directory information won’t let you in.</td>
</tr>
<tr>
<td>$50</td>
<td>File busy. Somebody else is already talking to that file (file open).</td>
</tr>
<tr>
<td>$51</td>
<td>Directory count is messed up and is different than the actual number of files in the directory.</td>
</tr>
<tr>
<td>$52</td>
<td>Disk is not a ProDOS format.</td>
</tr>
<tr>
<td>$53</td>
<td>Some parameter is out of range.</td>
</tr>
<tr>
<td>$55</td>
<td>Eight files on eight separate drives are open, and somebody wants still more.</td>
</tr>
<tr>
<td>$56</td>
<td>No buffers available. You’re trying to assign a buffer to a place in memory that’s already being used.</td>
</tr>
<tr>
<td>$57</td>
<td>Duplicate volumes. There are two disks online that have the same name.</td>
</tr>
<tr>
<td>$5A</td>
<td>The disk bitmap says there’s a free block somewhere past the actual size of the disk itself. The volume bitmap has been damaged.</td>
</tr>
</tbody>
</table>
Program 13-4. P8 File Dump Demo

1  **************************************************************
2  *  P8 FILE DUMP DEMO PROGRAM  *
3  *  MERLIN ASSEMBLER  *
4  **************************************************************
5
6  ORG $2000
7
8  DSK FDUMP.SYS
9  TYP $FF ; SYSTEM FILE TYPE
10
11  MLI EQU $BF00 ; STD. PRODOS 8 ENTRY
12  COUT EQU $FDED
13  HOME EQU $FC58
14  RDKEY EQU $FD6F ; MONITOR READ KEY ROUTINE
15  GETLN2 EQU $FD6F ; MONITOR INPUT ROUTINE W/O PROMPT
16  INBUF EQU $200 ; INPUT BUFFER
17  PRBYTE EQU $FD6F ; PRINT ACC. AS HEX NUMBER
18  RD80COL EQU $C01F ; BIT 7 = 1 = 80 COLS. “ON”
19  WNDWDTH EQU $21 ; TEXT WINDOW WIDTH
20  CH80 EQU $57B ; 80-COL HORIZ. CURSOR POSN
21
22
23  SETQUIT LDA #$4C ; JMP INSTRUCTION
24  STA $3F8 ; CTRL-Y VECTOR
25  LDA #$<QUIT ; LOW BYTE OF QUIT ADDR.
26  STA $3F9 ; LOW BYTE OF CTRL-Y VECTOR
27  STA $3FA ; HIGH BYTE OF CTRL-Y VECTOR
28
29
30  BEGIN BIT RD80COL ; 80 COLS ACTIVE?
31  LDA #$40 ; WINDOW WIDTH
32  STA WNDWDTH ; SET WIDTH, JUST IN CASE
33  STA CH80 ; SET 80-COL CURSOR H = 0
34
35
36  CLEAR JSR HOME ; CLEAR SCREEN
37
38  PROMPT LDY #$00 ; INIT Y-REG
39  LDA MESSG1,Y ; PRINT PROMPT MESS.
40  BEQ GETPATH
41  JSR COUT ; PRINT JT
42  INY ; NEXT CHAR
43  BNE :1 ; WRAPAROUND PROTECT
44
45  GETPATH JSR GETLN2 ; GET PATHNAME FROM USER
46
47  FIX PHX ; SAVE LENGTH OF INPUT STRING
48  LDA INBUF-1,X ; GET LAST CHAR
49  STA INBUF,X ; MOVE OVER ONE BYTE
50  DEX ; X = X - 1

265
Chapter 13

002036: D0 F7 =202F 51 BNE :1 ; NEXT CHARACTER
002038: FA 52 PLX ; RETRIEVE LENGTH
002039: 8E 00 02 53 STX INBUF ; PUT AT BEG. OF STRING
00203C: E0 04 55 CHK1 CPX #$04 ; 4 = LEN “QUIT”
00203D: BD 00 02 58 CHK2 LDA INBUF,X ; LAST CHAR OF INPUT
00203E: 29 DF 59 AND #$DF ; CONVERT TO UPPERCASE IF NEEDED
002040: DA 62 DEX
002041: 4C BO 20 64 JMP QUIT ; STR$ = “QUIT”
002046: 20 00 BF 66 OPEN JSR MLI ; OPEN COMMAND
00204F: 90 09 =2061 70 BCC OPEN2 ; NO ERROR
002051: 20 88 20 72 JSR ERROR ; PRODOS ERROR MESSAGE
002053: 70 20 88 73 JSR RDKEY ; WAIT FOR A KEYPRESS
002055: 4C OF 20 74 JMP BEGIN ; TRY AGAIN IF ERROR
00205C: AD E4 20 76 OPEN2 LDA PARMTBL2 + 5 ; GET REFERENCE NUMBER
00205E: 89 00 22 77 LDA BUFFER,Y ; INIT Y-REG
00205F: 09 80 93 ORA #$80 ; SET HIGH BIT
002060: 9E 20 ED FD 94 JSR COUT
002061: CC EB 20 95 CPY NUMREAD ; PRINT CHARS READ IN.
002062: F0 15 =208D 97 BCC :1 ; GET ANOTHER LINE OF TEXT
002063: 9A 00 01 100 CLOSE LDA #$01 ; REWRITE PARMTBL3
002064: 8D E5 20 101 STA PARMTBL3 ; # OF PARMS = 1
002066: 20 00 BF 102 JSR MLI

266
002095: CC 103  DFB $CC  ; CLOSE COMMAND
002096: E5 20 104  DA PARMTBL3  ; SAME TABLE AS 'READ'
002098: 90 03 =209D 105  BCC DONE  ; NO ERRORS
00209A: 20 B8 20 106  JSR ERROR  ; PRODOS ERROR MSSG

00209D: A0 00 108  DONE  LDY #$00  ; INIT Y-REG
00209F: B9 3D 21 109 :1  LDA MMSG3,Y  ; GET CHAR TO PRINT
0020A2: 20 ED FD 111  JSR COUT  ; PRINT IT
0020A4: 20 BS 20 112  INY  ; NEXT CHAR
0020A6: D0 F5 =209F 113  BNE :1  ; WRAPAROUND PROTECT

0020B0: 20 00 BF 118  QUIT  JSR MLI  ; DO QUIT CALL
0020B3: 65 119  DFB $65  ; QUIT CALL COMMAND VALUE
0020B4: D8 20 120  DA PARMTBL  ; ADDRESS OF PARM TABLE
0020B6: 00 00 121  BRK $00  ; SHOULD NEVER GET HERE ...

0020B8: 48 123  ERROR  PHA  ; SAVE ERROR CODE
0020BB: A0 00 124  LDY #$00  ; INIT Y-REG
0020BE: F0 06 =20C6 125 :1  LDA MMSG2,Y  ; GET CHAR TO PRINT
0020C0: 20 ED FD 127  JSR COUT  ; PRINT IT
0020C2: C8 128  INY  ; NEXT CHAR
0020C4: D0 F5 =20BB 129  BNE :1  ; WRAPAROUND PROTECT

0020C6: 68 131  PARMTBL DFB 3  ; NUMBER OF PARMS FOR OPEN = 3
0020C7: 20 DA FD 132  JSR PRBYTE  ; PRINT IT
0020CA: A0 00 133  LDY #$00  ; INIT Y-REG
0020CC: B9 23 21 134 :1  LDA MMSG2A,Y  ; GET CHAR TO PRINT
0020CE: F0 06 =20D7 135  BEQ ERDONE  ; END OF MMSG
0020D1: 20 ED FD 136  JSR COUT  ; PRINT IT
0020D3: C8 137  INY  ; NEXT CHAR
0020D5: D0 F5 =20CC 138  BNE :1  ; WRAPAROUND PROTECT

0020D7: 06 139  ERDONE  RTS

0020D8: 04 141  PARMTBL DFB 4  ; NUMBER OF PARMS
0020DA: 00 142  DFB 0  ; QUIT TYPE (0 = STD. QUIT)
0020DC: 00 143  DA $0000  ; NOT NEEDED FOR STD. QUIT
0020DD: 00 00 144  DFB 0  ; NOT USED AT PRESENT
0020DF: 03 145  DA $0000  ; NOT USED AT PRESENT

0020E0: 00 02 147  PARMTBL DFB 3  ; NUMBER OF PARMS FOR OPEN = 3
0020E2: 00 23 148  DA INBUF  ; POINTER TO PATHNAME
0020E4: 00 150  REFNUM DFB 0  ; PRODOS FILE REFERENCE NUMBER

0020E5: 00 152  PARMTBL DFB 0  ; NUMBER OF PARMS FOR READ/CLOSE
0020E6: 00 153  DFB 0  ; REFERENCE NUMBER
0020E7: 00 22 154  DA BUFFER  ; POINTER TO DATA BUFFER
0020E9: FF 00 155  DA 255  ; 255 CHARACTERS TO READ
Chapter 13

0020EB: 00 00 156 NUMREAD DA 0 ; NUMBER OF CHARACTERS READ.
157
159
0020ED: D0 CC C5 C1 160 MSSG1 ASC "PLEASE ENTER PATHNAME: ",8D
002105: A8 CF D2 A0 161 ASC "(OR "
00210A: D1 D5 C9 D4 162 WORD ASC "QUIT"
00210E: A7 A9 A0 8D 163 ASC ") ",8D,00
164
002113: 8D 165 MSSG2 HEX 8D ; PRINT RETURN FIRST
002114: D0 D2 CF C4 166 ASC "PRODOS ERROR ",00
002123: 8D 167 MSSG2A HEX 8D ; ANOTHER CARRIAGE RETURN
002124: D0 D2 C5 D3 168 ASC "PRESS A KEY TO TRY AGAIN",00
169
00213D: 8D 170 MSSG3 HEX 8D ; PRINT RETURN FIRST...
00213E: 80 D2 C5 D3 171 ASC "PRESS A KEY FOR NEXT FILE",00
172
002158: EC 173 CHKSUM CHK ; CHECKSUM FOR VERIFICATION
174
002159: 00 00 00 00 175 DS ; SKIP TO NEXT PAGE BOUNDARY
002200: 00 00 00 00 176 BUFFER DS $100 ; DATA BUFFER FOR US
177
178 DOSBUF ; 1024 BYTES FOR PRODOS BUFFER
179
180 ; NOT IN PROGRAM SO AS TO NOT
181 ; TAKE UP DISK SPACE ...

End Merlin-16 assembly, 768 bytes, errors: 0
Chapter 14

ProDOS 16
Chapter 14

ProDOS 16

Like ProDOS 8, ProDOS 16 is a set of disk access routines designed to be called from machine language programs. Unlike ProDOS 8, there is no BASIC.SYSTEM for Applesoft. In fact, in ProDOS 16, the assumption is that Applesoft BASIC and the Monitor no longer exist. You’re not entirely on your own, however. It’s in the ProDOS 16 environment that the new Apple IIGS tools like super hi-res graphics, the Event Manager, the Memory Manager, and other tools become available. On the one hand, you lose many of the points of reference you’re familiar with. On the other, you enter the real world of the Apple IIGS, where memory seems unbounded, and the hundreds of built-in Applesoft BASIC and Monitor routines are replaced by literally thousands of Apple IIGS Toolbox commands. Future chapters will explore those tools in detail; for now let’s look at ProDOS 16.

Starting Up ProDOS 16

On a disk set up to boot ProDOS 16, such as the Apple IIGS System Disk, the boot process starts the same as it did for ProDOS 8—by reading and executing the code stored in blocks 0 and 1 on the disk. This code is the same on any ProDOS disk, regardless of whether it is set up for ProDOS 8 or 16. As before, this code begins by running the file ProDOS on the disk. Here’s where things under ProDOS 16 change considerably.

On a ProDOS 16 boot disk, the file PRODOS is now just an intermediate program itself, one that is not actually either version of ProDOS. Instead, it’s an initialization file whose job it is to determine the correct operating system (ProDOS 8 or 16) for whatever the startup application on the disk is. When the file PRODOS first runs, the first thing it does is copy part of itself and call PQUIT (ProDOS Quit) to a part of memory outside the first 64K, where it will remain permanently. PQUIT has two specific functions: First, it loads the appropriate operating system for whatever application is about to be run. Second, it contains a program selector of sorts, which actually starts up the application named by a path name passed to it.

After PQUIT is installed, ProDOS 16 and the System Loader are loaded.
using the file P16 in the SYSTEM subdirectory. The System Loader is a separate unit from ProDOS 16, even though they share the same file (P16) on the disk. In ProDOS 8, every application is a .SYSTEM file and is loaded into memory starting at location $2000 in bank zero. Under ProDOS 16, all of bank zero becomes very valuable, and applications are loaded almost anywhere else in the many other banks of available memory.

Because the actual final location at which a program will run is now unknown to the programmer, simple object files cannot be used. Instead, a new type of assembler-created file, called relocatable, must be used. A relocatable file contains not only the object file as you see it assembled, but also additional information such as all the internal reference JMPs, JSRs, and LDAs. Creating a file like this with all the additional information means the actual program bytes can be rewritten when the file is loaded into memory. This loading and rewriting is done by the System Loader (not ProDOS 16 itself).

Once ProDOS 16 and the System Loader have been installed, all the appropriate Apple IIgs supporting files in the various subdirectories of the SYSTEM folder are loaded. For example, TOOL.SETUP in the SYSTEM.SETUP folder contains a number of fixes to the Apple IIgs internal ROM routines. No system is ever perfect, and the designers of the Apple IIgs knew errors in the ROM were bound to be discovered after the machine was in production.

The solution was to design almost everything in ROM in such a way that any ROM routine could be amended, or even replaced, by a substitute routine loaded into RAM when the system was booted. Much of this was accomplished with vectors to the various ROM routines. You'll recall from Chapter 12 that vectors are pointers in RAM that direct control to a certain routine. By changing the vectors that correspond to a given ROM routine to a loaded RAM routine, the substitution is made.

TOOL.SETUP is not the only file loaded during the boot process. Classic Desk Accessories that may be on the disk are also loaded. Basically, any file with the file types $B6 (for example TOOL.SETUP, a permanent initialization file), $B7 (a temporary initialization file), $B8 (Classic Desk Accessories such as SDUMP), or $B9 (New Desk Accessories like CLOCK) are loaded during this startup process.

Now that everything is in place, the system looks for a $B3 type (application) file named START. Usually, this is a program selector like the Apple Program Launcher, but it can be any application you wish to write. If START is found, it is loaded and run.

If START is not found, the system then searches the main directory for the first file that is either a ProDOS 8 SYS ($FF) file whose name ends in .SYSTEM, or a ProDOS 16 application (type $B3 = S16) whose name ends in
.SYS16. Depending on which type of file is found first, the appropriate operating system is selected (P8 is loaded if necessary), and the program is run.

Ultimately, it is the PQUIT routine that selects the proper operating system and application pathname.

The Simplest ProDOS 16 Program: Quit
As with ProDOS 8, ProDOS 16 applications must end with a quit command, not an RTS (or RTL). ProDOS 16 calls are made in a fashion similar to the MLI call for ProDOS 8. For ProDOS 16, a JSL is done to location $E1/00A8; this is followed by a six-byte data block. The first two bytes define the command value, or *call number*. The next four bytes are a long-address pointer to a ProDOS 16 parameter block. Notice that in the world of ProDOS 16, everything assumes we are running in banks that can be anywhere in memory.

In general, every Apple II GS ProDOS 16 application must meet the following requirements:

1. Have the filetype $B3 (S16). There are specialized applications that may have other file types.
2. Be created in the relocatable file format called the Object Module Format (OMF). This will be described in greater detail shortly.
3. Do a proper ProDOS 16 quit call. (No RTS, RTL or forced reset.)
4. Obtain any memory used externally to the program and its stack and direct page from the Memory Manager.

Item 4 may seem new to you. With so much memory on the Apple II GS, it may seem like you should be able to use whatever you want. But answer these questions: How will you know how much memory the user has installed in his machine? How will you know where your program is actually running? How will you share the available memory with other programs, such as desk accessories, that may have been loaded during the boot process?

The Memory Manager
To make things as easy as possible for everyone that uses the Apple II GS environment, the system includes a tool called the *Memory Manager*. The Memory Manager keeps track of all the things mentioned above so you can concentrate on your application. Basically, whenever you need some memory for a file buffer, a picture, or a data block, you just say, "Hey, get me 10K of memory." and the Memory Manager not only finds the memory for you, but also protects it from everybody else in the system. The bottom line is: Use the memory manager; don’t blaze your own trails.

For your first ProDOS 16 program, let’s write a ProDOS 16 equivalent of
the P8.SYSTEM program (Program 13-1) that was presented in Chapter 13. The main differences will be:

1. ProDOS 16 does an automatic screen clear when it starts up a program, so the JSR HOME won’t be needed.
2. You won’t have Applesoft BASIC or the Monitor to depend on, for three reasons: Our program is now being started up in the full 16-bit mode (Accumulator and index registers); we’ll undoubtedly be in a different bank than the Monitor/Applesoft BASIC routines; and we now have our own stack and direct page (automatically determined, allocated and assigned by the System Loader and Memory Manager when our program was loaded). You’re on your own.
3. The quit command for ProDOS 16 is slightly different from that for ProDOS 8.

Simple System File
With all that in mind, take a look at Program 14-1, P16.SYSTEM. The APW version of this program is Program 14-3. (Before assembling this program, you may want to read the section on linking, below.)

Because the start-assembly default of Merlin is for the 8-bit mode, we must begin Program 14-1 with a new Merlin directive, MX. This is equivalent to APW’s LONGA and LONGI directives, and it tells the assembler what the starting condition of the m and x bits are assumed to be. To indicate the 16-bit mode, MX %00 is used. You may change Merlin’s startup default on an assembly to be 16-bit, thus eliminating the need for this instruction. But keeping it in the listing doesn’t hurt anything and is actually a good idea if you want to avoid errors should you ever change the startup default.

Lines 8 and 9 show how to create a relocatable file output with Merlin. The REL directive on line 8 tells Merlin to create a ProDOS REL type file ($FE) on the disk during the assembly. This will be used by the Merlin linker to create the OMF file needed for the final application. (This is more or less equivalent to APW’s .ROOT output file).

Because the program is running in an indeterminant memory bank, the first thing we need to do is set the data bank register equal to the program bank register (the bank we’re running in). The PHK, PLB instructions will do the trick here.

Next, the prompt message is printed to the screen. Because we can no longer use COUT, the message is printed by storing the characters directly on the screen, as was done when you first started all this in Chapter 3. Line 22 uses the STAL (STA Long) instruction to put the bytes on the text screen. Remember, the data bank register setting at this point means a STA $400,X would store a byte (actually, two) in the bank you’re currently running in, not $00/$0400,
where the text screen is. If the data bank were set to $00, the STA instruction would work, but not the LDA MSSG,X. One or the other must use a long addressing mode instruction.

Also because the Accumulator is in the 16-bit (two byte) mode, characters are loaded and stored two at a time. This means the number of characters in the string at MSSG (line 44) must be an even number. Also, because the Accumulator is in the 16-bit mode, two INX instructions are used to increment to the next pair of bytes. To add insult to injury, the complete 80-column display is made up by interleaving the address range from $400 to $7FF for both banks 0 and 1. Because we're only writing to bank 0, the characters only appear at every other position on the screen.

The keyboard is read in a similar manner, and is thus a little different from the ProDOS 8 version of this program. Because the LDA KYBD instruction actually loads two bytes, one each from $C000 and $C001, it is necessary to do an AND #$00FF to zero-out the high-order byte of the Accumulator.

The CMP #$0080 then checks to see if a key was pressed. (By the way, for those of you who are really deep thinkers, yes, it's true that the typed instructions AND #$FF, CMP #$80 would have been equivalent to AND #$00FF, CMP #$0080.) Leading zeros don't do any more for the assembler than they do for any other number—$000000012 is still $12. They were used in this case to make it clear in the source listing that two bytes were involved in the operation. Perhaps this isn't a bad idea for your own programs.

Now for the quit command itself: $0029 is the call number for a ProDOS quit. This follows the JSL to the ProDOS 16 common entry point, $E1/00A8. Following the call number is the 4-byte pointer to the parameter block. ADRL (Address Long) is the Merlin pseudo-op for a 4-byte address pointer.

The parameter block for the ProDOS 16 quit command consists of just six bytes. The first four are a pointer to an optional pathname for the next application to be run. This is similar to the ProDOS 8 Enhanced Quit Call. In our program, this is set to zero, which tells ProDOS to just do a standard quit back to the previous program.

The next two bytes define a flag that uses the upper two bits (bits 14 and 15), to flag certain exit conditions. The options are as follows:

If bit 15 is set, it means that you would like program control to return to the program (the one doing the quit at that point), instead of going back to the starting program selector. If this bit is clear, as in our P16.SYSTEM program, or there is no pathname specified (pointer = $0), then a direct quit back to the previous program is done.

If bit 15 is set, and there is a pathname for the program to quit to, our
Chapter 14

program itself now acts like a program selector, wherein it can launch a specified application. Control will return to it when that application is finished. This is where bit 14 comes in. When you think about it, the PQUIT routine (the thing really managing all this) is going to have to rerun your program when the other one quits. The obvious way to do this is to reload it from disk. However, this is not only slow (relatively speaking), but it also requires the proper disk be in the drive.

Since all that memory is lying around, why not just keep a copy of our program in memory in a dormant state, and resurrect it when the time comes? That is just what bit 14 is for. You can tell the system which you prefer. If bit 14 is set (flag = $CO00), your program will be restarted from memory. On the other hand, if you want to force a restart from disk, bit 14 can be left clear (flag = $8000).

Program 14-1. Simple P16 System File

```
;**********************************************
* SIMPLP16 SYSTEM FILE
* MERLIN ASSEMBLER
;**********************************************

MX %00 ; FULL 16 BIT MODE
REL ; RELOCATABLE OUTPUT
DSK P16.SYSTEM.L

=E100A8 PRODOS EQU $E100A8 ; PRODOS 16 ENTRY POINT
=C000 KYBD EQU $C000
=C010 STROBE EQU $C010
=0400 SCREEN EQU $00400 ; LINE 1 ON SCREEN

008000: 4B ENTRY PHK ; GET PROGRAM BANK
008001: AB PLB ; SET DATA BANK

008002: A2 00 00 19 PRINT LDX #$00 ; INIT X-REG
008005: BD 34 80 20 LOOP LDA MSSG,X ; GET CHAR TO PRINT
008008: F0 08 ≈8012 21 BEQ GETKEY ; END OF MSSG.
00800A: 9F 00 04 00 22 STAL SCREEN,X ; "PRINT" IT
00800E: E8 23 INX ; NEXT TWO CHARs
00800F: E8 24 INX ; X = X + 2
008010: D0 F3 ≈8005 25 BNE LOOP ; WRAPAROUND PROTECT

008012: AD 00 C0 27 GETKEY LDA KYBD ; CHECK KEYBOARD
008015: 29 FF 00 28 AND #$00FF ; CLEAR HI BYTE
008018: C9 80 00 29 CMP #$0080 ; KEYPRESS?
00801B: 90 F5 ≈8012 30 BCC GETKEY ; NOPE
00801D: 2C 10 C0 31 BIT STROBE ; CLEAR KEYPRESS
```
ProDOS 16

The Launcher

To show how all this works, Program 14-2 is a ProDOS 16 version of the P8.LAUNCHER (Program 13-3), called P16.LAUNCHER. The APW version of the launcher is Program 14-4.

This program starts off the same way as P16.SYSTEM. The main difference is that now four inputs are allowed: the number keys 0, 1, 2, and 3.

Three is the most complex example (flag = $C000). This will launch the previous program, P16.SYSTEM, from disk, and will store itself in memory. When P16.SYSTEM quites, P16.LAUNCHER will resume instantly from memory. In principle, there can be many multiple levels of different programs, or program modules, each running a successive module with control ultimately returning to the master program.

Option 2 (flag = $0) simply starts P16.SYSTEM without leaving itself in the return list. Thus, when P16.SYSTEM quits, control goes back to the previous program selector.

Option 1 is equivalent to the direct quit done by P16.SYSTEM.

Option 0 demonstrates what happens either when there is no program left in the return list to go to, or when the file specified is not found. This activates the Apple IIgs ProDOS interactive restart menu that gives you the option of rebooting, running the file named START, or specifying a startup program of your own.

By the way, you may have been wondering why we didn’t just bracket...
the 8-bit operations in the program P16.SYSTEM in a pair of SEP and REP instructions like this:

```
SEP $30 ; 8-BIT MODE
PRINT LDX #$00 ; INIT X-REG
LOOP LDA $1 ; GET CHAR TO PRINT
BEQ GETKEY ; END OF MSG.
STAL SCREEN,X ; "PRINT" IT
INX ; NEXT TWO CHARS
INX ; X = X + 2
BNE LOOP ; WRAPAROUND PROTECT
GETKEY LDA KYBD ; CHECK KEYBOARD
AND #$00FF ; CLEAR HI BYTE
CMP #$0080 ; KEYPRESS?
BCC GETKEY ; NOPE
BIT STROBE ; CLEAR KEYPRESS
REP $30 ; BACK TO 16-BIT MODE
```

There are two reasons. First, to show you what alternative you may have when going to the 16-bit mode may be impractical, and second, because doing this as P16.LAUNCHER is a little impractical.

Here's why. The SEP on line 22 or so of P16.LAUNCHER would work OK, but where can we put the REP $30? At the end of the print loop, around line 30, would work, but now the GETKEY routine is still in the 16-bit mode. You could move it to somewhere near line 36, but now you're comparing an Accumulator loaded in the 8-bit mode with 2-byte ASCII codes.

If you try to move the REP $30 past the compares, you'll have to put four of them in, one for each entry point to the various quit commands.

Program 14-2. ProDOS 16 Launcher Demo

```
1  ****************************
2  * PRODOS 16 'LAUNCHER' DEMO  *
3  * LAUNCHES 2ND SYSTEM FILE, *
4  * STAYS DORMANT, THEN REVIVED *
5  * WHEN 2ND QUITS.            *
6  *                             *
7  * MERLIN ASSEMBLER            *
8  ****************************
9
10  MX %00                     ; FULL 16-BIT MODE
11  REL
12  DSK P16.LAUNCH.L           ; RELOCATABLE OUTPUT
13
14
```

278
ProDOS 16

=E100A8 15 PRODOS EQU $E100A8 ; PRODOS 16 ENTRY POINT
=C000 16 KYBD EQU $C000
=C010 17 STROBE EQU $C010
=0400 18 SCREEN EQU $400 ; LINE 1 ON SCREEN

008000: 4B 20 ENTRY PHK ; PUSH CODE BANK
008001: AB 21 PLB ; PULL DATA BANK

008002: A2 00 00 23 PRINT LDX #$00 ; INIT X-REG
008005: BD 85 80 24 LOOP LDA MSGX ; GET CHAR TO PRINT
008008: F0 08 =8012 25 BEQ GETKEY ; END OF MSG.
00800A: 9F 00 04 00 26 STAL SCREEN,X ; "PRINT" IT

00800E: E8 27 INX ; NEXT TWO CHARS
00800F: E8 28 INX ; X = X + 2

008010: D0 F3 =8005 29 BNE LOOP ; WRAPAROUND PROTECT

008012: AD 00 C0 31 GETKEY LDA KYBD ; KEYPRESS?
008015: 29 FF 00 32 AND #$00FF ; MASK UPPER BYTE
008018: C9 80 00 33 CMP #$80 ; HI BIT SET?
008018: 90 FS =8012 34 BCC GETKEY ; NOPE

00801D: 2C 10 35 BIT STROBE ; CLEAR KEYPRESS

008020: C9 B0 00 37 CHK CMP #"O" ; QUIT TO ROM ROUTINE?
008023: F0 13 =8038 38 BEQ QUIT0 ; YES
008025: C9 B1 00 39 CMP #"1" ; REAL QUIT?
008028: F0 1B =8045 40 BEQ QUIT1 ; YES
00802A: C9 B2 00 41 CMP #"2" ; LAUNCH 2ND, DON'T RETURN
00802D: F0 23 =8052 42 BEQ QUIT2 ; LAUNCH 2ND, RETURN
00802F: C9 B3 00 43 CMP #"3" ; QUIT CALL
008032: F0 29 =865F 44 BEQ QUIT3 ; QUIT CALL

008034: 5C 12 80 00 46 TRYAGN JML GETKEY ; TRY AGAIN
008038: 22 A8 00 E1 48 QUIT0 JSL PRODOS ; DO QUIT CALL
00803C: 29 00 49 DA $29 ; QUIT CODE
00803E: 6C 80 00 00 50 ADRL PARM0 ; ADDRESS OF PARM TABLE
008042: B0 40 =8084 51 BCS ERROR ; NEVER TAKEN
008044: 00 52 BRK ; WE'LL NEVER GET HERE?

008045: 22 A8 00 E1 54 QUIT1 JSL PRODOS ; DO QUIT CALL
008049: 29 00 55 DA $29 ; QUIT CODE
00804B: 72 80 00 00 56 ADRL PARM1 ; ADDRESS OF PARM TABLE
00804F: B0 33 =8084 57 BCS ERROR ; NEVER TAKEN
008051: 00 58 BRK ; WE'LL NEVER GET HERE?

008052: 22 A8 00 E1 60 QUIT2 JSL PRODOS ; DO QUIT CALL
008056: 29 00 61 DA $29 ; QUIT CODE
008058: 78 80 00 00 62 ADRL PARM2 ; ADDRESS OF PARM TABLE
00805C: B0 26 =8084 63 BCS ERROR ; NEVER TAKEN
00805E: 00 64 BRK ; WE'LL NEVER GET HERE??

00805F: 22 A8 00 E1 66 QUIT3 JSL PRODOS ; DO QUIT CALL
008063: 29 00 67 DA $29 ; QUIT CODE
Chapter 14

Merlin File Assembly and Linking Instructions

Linking is the process of combining several different program modules, that have already been assembled, into one or more final output object files. Remember that a relocatable file contained information about the JSRs and JMPs. It can also contain information about calls to routines in itself that other modules might want to reference.

For example, suppose there are three program modules, A, B, and C. Furthermore, let’s suppose that B and C have the entry points READ and
WRITE, respectively. Program A will represent the main program, and B and C are modules that read and write files to the disk.

Program A will want to call the routines READ and WRITE in B and C, but remember these are being assembled separately. So, while the programmer is working on module A, he just tells the assembler that some other module will eventually have the external label READ in it. During the assembly, then, the assembler knows not to worry about the fact that READ hasn’t been given a specific address yet. You may know from assembling files already that normally an assembler will generate an error if a label is used that doesn’t have an address defined for it. The EXT (EXTERNAL) is used to assign external labels.

While working on B, the programmer tells the assembler that READ, which is a label in module B, is an entry point that other programs may want to use. This is done with another assembler directive, ENT (ENTRY point). Module C will have WRITE defined in a similar way.

Having assembled each of the three programs individually into its own relocatable files (type REL = $FE), this is where the linker comes in. The linker uses a list, usually a text file, of all the modules to be combined into one or more final output files. In the case of the Apple II GS and Merlin 16, this will be an Object Module Format (OMF) file. The list may also contain specific commands for the Linker, telling it to save the output file, to re-assemble a file, or which file type should be used for the output file.

Such a list might look like this:

```
TYPE $B3
LINK A
LINK B
LINK C
SAVE PROGRAM
```

During the linking process, the assembler will reconcile the calls to the READ and WRITE entry points in modules B and C with the JSRs (or whatever) to those labels from module A. This is the purpose of linking a file.

Also notice, however, that nothing insists that more than one input file be used. If all you want to do is to create an OMF file from a REL file, a link list with one file in it will do the trick.

To assemble, link, and run P16.SYSTEM, you must first load and activate the Merlin 16 linker. This is done from the main menu by BRUNning the file LINKER.GS on the Merlin 16 disk. You can type -LINKER.GS as a disk command to do this. The linker need only be installed once during a session.
Chapter 14

From then on, any program can be assembled and linked by following these steps:

1. Enter and assemble the listing, exactly as shown. Save the source file. This will give you two files on the disk, P16.SYSTEM.S and P16.SYSTEM.L (a REL file). Notice the special suffix .L given the REL file to differentiate it from the source file and the final OMF file. The suffix was part of the name in the DSK instruction and is not added automatically by Merlion.

2. After assembling and saving the source file, type in NEW to clear the editor workspace. Type this in:

   TYP $B3 ; $16 FILE
   LINK P16.SYSTEM.L ; CONVERT REL TO OMF
   SAVE P16.SYSTEM ; SAVE OUTPUT FILE

3. Save this new file on the disk under the name P16.SYS.CMD. This is the linker command list. It cannot be directly assembled. Do not type ASM to execute it.

4. To do the final link, type NEW again to clear the workspace, and then type:

   LINK "P16.SYS.CMD"

   and press Return.

5. When the link is complete, you'll be returned to the main menu. Your final $16 file has already been saved on the disk. Return now to your program selector and try running P16.SYSTEM. It should work as described next.

   If there is only one file to be linked, Merlion 16 also has a shortcut link command that will link the last source file saved, and will create an OMF file, with the type $16. To do the quick link, type LINK "=" instead of an actual filename while at the command prompt (;) in the Editor. The final object file generated by the link will be saved under the name P16.SYSTEM, assuming you use the DSK command DSK P16.SYSTEM.L as the REL type output file in the source listing itself.

   This general procedure should be used to assemble, link and run all further ProDOS 16 Merlion programs.

APW Linking

Linking a file in APW will seem fairly easy, because you've already done it. Because all files in APW must be linked, the linking process is part of the ASML command (Assemble and Link). For ProDOS 8, presumably you've been reverse-converting the files after the ASML back to a standard object file with the MAKEBIN and FILETYPE commands.
Normally, the output of the ASML command is an OMF file. After assembling with ASML, just type in FILETYPE S16. You can then quit APW and test the program by starting it from a program selector. Although APW will let you launch a program directly from within its command mode, it’s better to use the program selector to avoid unexpected interactions between APW and your programs.

For your reference, Program 14-3 and Program 14-4 are the APW versions of the two programs. Since APW assumes a starting long mode for the Accumulator and index registers, no LONGA ON, LONGI ON instructions are needed (although, like Merlin, it doesn’t hurt, and it makes the starting conditions for the assembly clear).

Program 14-3. APW Simple P16 System File

```
0001 0000  **********************************************************************
0002 0000  * SIMPLE P16 SYSTEM FILE *
0003 0000  * APW ASSEMBLER *
0004 0000  **********************************************************************
0005 0000
0006 0000  KEEP P16.SYSTEM
0007 0000  MSB ON
0008 0000
0009 0000  MAIN START
0010 0000
0011 0000  PRODOS EQU $E100A8 ; PRODOS 16 ENTRY POINT
0012 0000  KYBD EQU $C000
0013 0000  STROBE EQU $C010
0014 0000  SCREEN EQU $000400 ; LINE 1 ON SCREEN
0015 0000
0016 0000  4B ENTRY PHK ; GET PROGRAM BANK
0017 0001  AB PLB ; SET DATA BANK
0018 0000
0019 0002  A2 00 00 PRINT LDX #$00 ; INIT X-REG
0020 0005  BD 36 00 LOOP LDA MSSG,X ; GET CHAR TO PRINT
0021 0008  F0 08 BEQ CREATE ; END OF MESS.
0022 000A  9F 00 04 00 STA >SCREEN,X ; "PRINT" IT
0023 000E  E8 INX ; NEXT TWO CHAR'S
0024 000F  E8 INX ; X = X + 2
0025 0010  D0 F3 BNE LOOP ; WRAPAROUND PROTECT
0026 0012
0027 0012  AD 00 C0 GETKEY LDA KYBD ; CHECK KEYBOARD
0028 0015  29 FF 00 AND #$00FF ; CLEAR HI BYTE
0029 0018  C9 80 00 CMP #$0080 ; KEYPRESS?
0030 001B  90 F5 BCC GETKEY ; NOPE
0031 001D  2C 10 C0 BIT STROBE ; CLEAR KEYPRESS
0032 0020
0033 0020  22 A8 00 E1 QUIT JSL PRODOS ; DO QUIT CALL
0034 0024  29 00 DC 12'29' ; QUIT CODE
0035 0026  2E 00 00 00 DC 'PARAMTAB' ; ADDRESS OF PARM TABLE
0036 002A  B0 08 BCS ERROR ; NEVER TAKEN
```

283
Program 14-4. APW ProDOS 16 Launcher Demo

0001 0000 ****************************
0002 0000 * PRODOS 16 'LAUNCHER' DEMO *
0003 0000 * LAUNCHES 2ND SYSTEM FILE, *
0004 0000 * STAYS DORMANT, THEN REVIVED *
0005 0000 * WHEN 2ND QUITS. *
0006 0000 *
0007 0000 * APW ASSEMBLER *
0008 0000 ****************************
0009 0000
0010 0000 KEEP   P16.LAUNCH
0011 0000 MSB    ON
0012 0000 MAIN   START
0013 0000
0014 0000
0015 0000 PRODOS EQU $E100A8 ; PRODOS 16 ENTRY POINT
0016 0000 KYBD EQU $C000
0017 0000 STROBE EQU $C010
0018 0000 SCREEN EQU $400 ; LINE 1 ON SCREEN
0019 0000
0020 0000 4B ENTRY PHK ; PUSH CODE BANK
0021 0001 AB PLB ; PULL DATA BANK
0022 0002
0023 0002 A2 00 00 PRINT LDX #$00 ; INIT X-REG
0024 0005 BD 8A 00 LOOP LDA MSSI X ; GET CHAR TO PRINT
0025 0008 F0 08 BEQ GETKEY ; END OF MSSG.
0026 000A 9F 00 04 00 STA >SCREEN,X ; "PRINT" IT
0027 000E E8 INX ; NEXT TWO CHAR S
0028 000F E8 INX ; X = X + 2
0029 0010 D0 F3 BNE LOOP ; WRAPAROUND PROTECT
002A 0012
002B 0012 AD 00 C0 GETKEY LDA KYBD ; KEYPRESS?
002C 0015 29 FF 00 AND #$80FF ; MASK UPPER BYTE
002D 0018 C9 80 00 CMP #$80 ; HI BIT SET?
002E 001B 90 F5 BCC GETKEY ; NOPE
002F 001F

284
0035 001D 2C 10 C0  BIT  STROBE  ; CLEAR KEYPRESS  
0036 0020  
0037 0020  C9 B0 00  CHK  CMP  #'0'  ; QUIT TO ROM ROUTINE?  
0038 0023  F0 13  BEQ  QUIT0  ; YES  
0039 0025  C9 B1 00  CMP  #'1'  ; REAL QUIT?  
0040 0028  F0 1C  BEQ  QUIT1  ; YES  
0041 002A  C9 B2 00  CMP  #'2'  ; LAUNCH 2ND, DON'T RETURN  
0042 002D  F0 25  BEQ  QUIT2  ; LAUNCH 2ND, RETURN  
0043 002F  C9 B3 00  CMP  #'3'  
0044 0032  F0 2E  BEQ  QUIT3  
0045 0034  
0046 0034  5C 12 00 00  TRYAGN  JML  GETKEY  ; TRY AGAIN  
0047 0038  
0048 0038  22 A8 00 E1 QUIT0  JSL  PRODOS  ; DO QUIT CALL  
0049 003C  29 00  DC 12'$29'  ; QUIT CODE  
0050 003E  70 00 00 00  DC 14'PARM0'  ; ADDRESS OF PARM TABLE  
0051 0042  B0 44  BCS  ERROR  ; NEVER TAKEN  
0052 0044  00 00  BRK  ; WE'LL NEVER GET HERE??  
0053 0046  
0054 0046  22 A8 00 E1 QUIT1  JSL  PRODOS  ; DO QUIT CALL  
0055 004A  29 00  DC 12'$29'  ; QUIT CODE  
0056 004C  76 00 00 00  DC 14'PARM1'  ; ADDRESS OF PARM TABLE  
0057 0050  B0 36  BCS  ERROR  ; NEVER TAKEN  
0058 0052  00 00  BRK  ; WE'LL NEVER GET HERE??  
0059 0054  
0060 0054  22 A8 00 E1 QUIT2  JSL  PRODOS  ; DO QUIT CALL  
0061 0058  29 00  DC 12'$29'  ; QUIT CODE  
0062 005A  7C 00 00 00  DC 14'PARM2'  ; ADDRESS OF PARM TABLE  
0063 005E  B0 28  BCS  ERROR  ; NEVER TAKEN  
0064 0060  00 00  BRK  ; WE'LL NEVER GET HERE??  
0065 0062  
0066 0062  22 A8 00 E1 QUIT3  JSL  PRODOS  ; DO QUIT CALL  
0067 0066  29 00  DC 12'$29'  ; QUIT CODE  
0068 0068  82 00 00 00  DC 14'PARM3'  ; ADDRESS OF PARM TABLE  
0069 006C  B0 1A  BCS  ERROR  ; NEVER TAKEN  
0070 006E  00 00  BRK  ; WE'LL NEVER GET HERE??  
0071 0070  
0072 0070  A2 00 00 00 PARM0  DC 14'NAME0'  ; BAD PATH TO GEN ERROR  
0073 0074  00 00  FLAG0  DC 12'$00'  ; ABSOLUTE QUIT  
0074 0076  
0075 0076  00 00 00 00 PARM1  DC 14'$00'  ; NO PATHNAME  
0076 007A  00 00  FLAG1  DC 12'$00'  ; ABSOLUTE QUIT  
0077 007C  
0078 007C  A4 00 00 00 PARM2  DC 14'NAME1'  ; PTR TO PATHNAME  
0079 0080  00 00  FLAG2  DC 12'$00'  ; BITS 15,14 = 0: DON'T RESTART  
0080 0082  
0081 0082  A4 00 00 00 PARM3  DC 14'NAME1'  ; PTR TO PATHNAME  
0082 0086  00 C0  FLAG3  DC 12'$0000'  ; BITS 15,14 = 1: RESTART LATER  
0083 0088  
0084 0088  00 00  ERROR  BRK  ; WE'LL NEVER GET HERE?  
0085 008A  
0086 008A  D0 D2 C5 D3  MSSG  DC  C'PRESS 0, 1, 2, OR 3 ->'; EVEN NUMBER OF CHARACTERS'
Chapter 14

0087 00A0 00 00       DC II'0,0'                ; TWO ZEROS
0088 00A2
0089 00A2             * 0 - QUIT TO ROM RESTART
0090 00A2             * 1 - QUIT TO PREVIOUS PROGRAM
0091 00A2             * 2 - LAUNCH "P16.SYSTEM"
0092 00A2             * 3 - LAUNCH "P16.SYSTEM" AND RETURN WHEN DONE
0093 00A2
0094 00A2 01          NAMEO  DC II'1'                ; LEN OF ZERO
0095 00A3 D8          DC C'X'                ; WON'T FIND THIS!
0096 00A4
0097 00A4 0A          NAME1  DC II'NAMEND-NAMEN-1'    ; LEN OF PATHNAME
0098 00A5 D0 B1 B6 AE  DC C'P16.SYSTEM'        ; 2ND TEST SYS FILE
0099 00AF             NAMEND ANOP
0100 00AF
0101 00AF             END

101 source lines
0 macros expanded
0 lines generated
Chapter 15

A Look at Memory Use on the Apple II GS
Chapter 15

A Look at Memory Use on the Apple IIIGS

First of all, congratulations for having worked so hard to reach this point in the book. Going from only a simple understanding of BASIC to being able to write programs under both ProDOS 8 and ProDOS 16 is no trivial accomplishment.

Reflect for a moment on what you’ve learned so far: over 75 assembly language instructions; how to input and print text to the screen; how to manipulate the hi-res screen; Boolean math; the intricacies of the direct page and the stack, and exotic addressing schemes; how to extend Applesoft BASIC with your own machine language routines; how to pass variables back and forth between Applesoft BASIC and machine language; how to use the Monitor to examine memory, write a program, and debug it; how to use an assembler; and most of all, general techniques in programming that go beyond just knowing a command or two.

What all that gets you right now is the foundation to embark on writing your own programs for the Apple IIIGS that would be impossible on any other machine. You see, the Apple IIIGS, with its thousands of built-in routines, can act like an amplifier of your existing talent and let you write programs today that would be a super-human accomplishment for programmers just a few years ago.

Before we start with the actual Apple IIIGS Toolset, let's review the overall memory use of the Apple IIIGS by the different operating systems and applications.

Now, with what you’ve learned in the previous chapters, the explanation of total memory usage on the Apple IIIGS will seem easy to understand because you’ve already learned a lot of the most important details while you were learning about other major concepts. There will be new discoveries, like additional banks of memory, but most of this new information will be for your own background as a soon-to-be expert on the Apple IIIGS. After all, you can’t claim to be an expert if you’ve never heard of bank-switching or the alternate 4K bank of memory, now can you?
Chapter 15

The Apple II: Past Lives

Although it's not essential to know anything about previous Apple computers (like the Apple II+, IIe and IIc) to program on the Apple IIGS, a few facts about the past will help you understand why things are the way they are today.

The original Apple II and II+ machines were 64K computers, whose memory was roughly equivalent to just bank 0 of the Apple IIGS today. Because the first bank of memory on the GS is still used in very much the same way, and because Applesoft BASIC has not fundamentally changed since the Apple II+, let's look at bank 0 first (see Figure 15-1).

Figure 15-1. Zero Page: $00 to $FF

<table>
<thead>
<tr>
<th>$FF</th>
<th>$FE</th>
<th>$FD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$03</td>
<td>$02</td>
<td>$01</td>
</tr>
<tr>
<td>$00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first 256 ($100) bytes are page zero. This is the default direct page on the Apple IIGS, and was the only possible direct page on all Apples before the Apple IIGS that used the 6502 or 65C02 microprocessors. Because of the importance of indirect and indexed addressing, this is the most important memory use in the computer.

In Applesoft BASIC, for example, every important pointer—such as to the size of the program in memory, which line is currently being executed and the character in the line which is being looked at—is stored in page zero. Even if you had no reference books whatsoever on the internal workings of the computer, quite a large amount of information could be deduced just by looking at the contents of page zero and seeing how the memory values changed as different programs were loaded and run.

This is exactly what many people did in the late 1970s, when the Apple computer was first produced, and much more technical information was to be found published by average owners of the computer who had taken the time to do a little investigation than by Apple Computer Company itself.
A Look at Memory Use on the Apple IIGS

This tradition continues today, but the real message is that you don’t always have to depend on someone else to give you the information you desire. A little work and clear deductive thought can teach you a lot about the computer.

On the Apple IIGS, the direct page can be relocated by a program by using the Direct Page Register. This is a valuable enhancement to the original Apple II design because now programs don’t have to fight over the same part of memory. In the Applesoft BASIC environment, you’ve got Applesoft BASIC, the Monitor, ProDOS and your routine all trying to use the same 256 bytes of memory. Since the first three contenders are preprogrammed, this means you’ve got to be very careful choosing those zero-page bytes (we used $06–09) to not get in somebody else’s way.

With the Apple IIGS, even in routines called from Applesoft BASIC, you can re-assign the direct page for your routine’s use, as long as you’re sure to restore it to page zero before you return to Applesoft BASIC, call the ProDOS 8 MLI, or call an internal Applesoft BASIC or Monitor routine (like COUT).

By the way, another completely acceptable approach to using page zero is to just save the contents of the bytes you want to use in your routine, and then restore them when you’re done. The ProDOS MLI instructions, for example, take this approach and save the contents of $40–$4E, and then restore them when the command is finished. The main thing to be careful not to do is save the contents, use the bytes for yourself, and then call some other routine that expects the original values to be there when instead you’ve changed them.

For example, locations $28, 29 are used by COUT as the base address for the current text screen line being printed by the Monitor or Applesoft BASIC. If you saved the contents of these bytes, used $28,29 in your routine, and then restored them, no one would be any the wiser. However, if your routine called COUT in the middle of all this, things would really go crazy because COUT would expect the original values to still be there.

The Stack: $100 to $1FF

| $100–$1FF  | The Stack |
| $00–$FF   | Zero (Direct) Page |

The next page of memory, $100 to $1FF, is the default stack area. In the Applesoft BASIC environment, the stack is limited to $100 bytes, and is filled from $1FF–downward with stored return addresses for JSRs and JSls, and values pushed on the stack by running programs.

Like the direct page, this can also be changed by a routine, by changing
Chapter 15

the Stack Pointer to any two-byte address in the first bank (bank 0) of memory. With the 65816 in the full 16-bit mode, the stack can be of any size, and it is up to the programmer to see that data in it does not collide with anything stored below it.

The Input Buffer: $200 to $2FF

<table>
<thead>
<tr>
<th>$200–$2FF</th>
<th>The Input Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$100–$1FF</td>
<td>The Stack</td>
</tr>
<tr>
<td>$00–$FF</td>
<td>Zero (Direct) Page</td>
</tr>
</tbody>
</table>

The area from $200 to $2FF is used by Applesoft BASIC and the Monitor to store characters as they are input from the keyboard. Certain ProDOS 8 programs also use the second half of the input buffer ($280 to $2FF) to pass pathnames between successive programs. When ProDOS is reading or writing a file from disk, a dedicated buffer defined elsewhere in memory is used, so the input buffer from $200 to $2FF is not used, although you may see the term input buffer used in reference to other data areas.

Because the input buffer is so heavily used on a temporary basis, it makes an excellent choice for a temporary block of memory for your own programs, especially for string operations. Just don’t plan on leaving anything there after you leave your routine and expect to find it there when you come back.

Page Three

Figure 15-2 shows page three. The area from $300 to $3CF is considered an open area for user-defined machine language routines. This is the area used for most of the short demonstration programs so far in this book.

Starting at $3D0 are a number of ProDOS and system vectors that should not be changed unless you know exactly what you are doing. The existence and use of some of the vectors depend on whether you are in ProDOS 8 or 16, or are using Applesoft BASIC. Vectors like this are usually self-evident. For example, a ProDOS 16 program doesn’t have to worry about the Applesoft BASIC ampersand ( &) vector at $3F5–$3F7.

Looking through the list, here’s a brief discussion of each group of page-three bytes. A JMP instruction is held by $3D0–3D2 for the BASIC.SYSTEM warm-start entry. This is equivalent to being in the Monitor and typing Control-C. The difference is that this will reconnect ProDOS if an I/O handler has gotten things confused. The same vector is contained in $3D3–$3D5. Once upon a time, in a disk operating system called DOS 3.3, they were different.
A Look at Memory Use on the Apple IIgs

Figure 15-2. Page Three: $300 to $3FF

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3FE-$3FF</td>
<td>Address for IRQ handler (interrupts).</td>
</tr>
<tr>
<td>$3FB-$3FD</td>
<td>JMP vector for non-maskable interrupts.</td>
</tr>
<tr>
<td>$3F8-$3FA</td>
<td>JMP vector Control-Y Monitor command.</td>
</tr>
<tr>
<td>$3F5-$3F7</td>
<td>JMP vector for BASIC &amp; commands.</td>
</tr>
<tr>
<td>$3F4</td>
<td>Power-up byte. Must be EOR of contents of $3F3 with #$A5.</td>
</tr>
<tr>
<td>$3F2-$3F3</td>
<td>Address for a RESET restart.</td>
</tr>
<tr>
<td>$3F0-$3F1</td>
<td>Address of BRK handler.</td>
</tr>
<tr>
<td>$3EF</td>
<td>(Unused)</td>
</tr>
<tr>
<td>$3ED-$3EE</td>
<td>Address to go to when XFER ($C314) is called.</td>
</tr>
<tr>
<td>$3D6-$3EC</td>
<td>Reserved for use by a ProDOS 8 SYSTEM.</td>
</tr>
<tr>
<td>$3D3-$3D5</td>
<td>BASIC.SYSTEM warm-start entry vector.</td>
</tr>
<tr>
<td>$3D0-$3D2</td>
<td>BASIC.SYSTEM warm-start entry vector.</td>
</tr>
<tr>
<td>$300-$3CF</td>
<td>Free Area</td>
</tr>
<tr>
<td>$200-2FF</td>
<td>The Input Buffer</td>
</tr>
<tr>
<td>$100-1FF</td>
<td>The Stack</td>
</tr>
<tr>
<td>$00-$FF</td>
<td>Zero (Direct) Page.</td>
</tr>
</tbody>
</table>
Locations $3D6$ to $3EC$ are reserved for use by a ProDOS 8 system program. You can use the bytes here for a vector to routines in your own program. You might wonder why you would want to do this. Suppose you’re debugging a large program that has a tendency to crash. Certain entry points that could be used to restart the program after a crash may be within the body of the program itself. Instead of trying to remember that $21A7$ is the current assembly’s warm-start (restart without clearing variables), you could put a permanent warm-start vector of your own at $3D6$-$3D8$ (for example, JMP $21A7$). That way, each reassembly will put the correct reentry vector at $3D6$, and you only have to remember one address.

Locations $3ED$ and $3EE$ hold an address for something called the XFER (for transfer) routine. The XFER routine is derived from use on Apple Ile and Iic machines that used the 65C02 microprocessor, and could not do long-address JMPs or JSRs to other banks of memory. In the case of the Ile and Iic, there is only one other bank of memory (ignoring expansion RAM cards for the moment), bank 1, also called auxiliary memory or AUXMEM. XFER is a routine designed to transfer program control from one bank to another.

For example, if you had a program running in bank 0 at $300$, and wanted to jump to $300$ in bank 1, you could store the bytes $00,03$ in locations $3ED,3EE$, and do a JMP $C314$ (the XFER routine). The $C314$ address is a clue that the routine is on a peripheral card in slot 3, and it’s true. The 80-column display uses bank 1 for every-other character on the screen, and there are a number of built-in routines associated with the 80-column firmware and the extra memory in bank 1. The 80-column display routines appear as though they are on a peripheral card in slot 3. XFER is one of those routines.

Locations $3F0$ and $3F1$ hold the address that the computer will jump to after it has encountered a BRK instruction and has taken care of its own business with the event. The existence of this vector makes debuggers and programs that step through and trace assembly language programs possible.

Suppose we wrote a program that could keep track of a simulated program counter—where in memory a program was executing. That program could place a BRK instruction after the current instruction; then JMP to it. When the BRK was encountered after the instruction, the vector at $3F0$, $3F1$ would point back to our program, which would restore the byte where the BRK was and advance our own program counter to this now-current instruction. Then a new BRK would be written after this instruction, and the process would repeat. In this way, we would get control back after each and every program step in the target program. This is how a step-and-trace (or debugger) is written for the Apple computer.

The next three bytes, $3F2$–$3F4$ are the address of where to jump to after
A Look at Memory Use on the Apple II GS

a RESET, followed by a special checksum byte. When RESET is pressed, a number of things are reinitialized that don’t include reconnecting BASIC.SYSTEM under ProDOS 8. If things were not fixed up, typing CATALOG after a RESET would just give a SYNTAX ERROR. Likewise, for a SYSTEM program of your own, it’s very good practice to trap RESET, so that the user isn’t dumped into the Monitor if RESET is pressed.

The checksum byte is used by the Monitor to make sure that the vector that’s there is intended. When you first turn on the machine, some value has to be there. One of the ways the computer knows it has just been turned on is by doing an exclusive-OR (EOR) of the contents of location $3F3 with the constant value $A5. It then compares the result to a checksum byte stored in location $3F4. If they don’t match, the computer assumes that it has just been turned on, and a total reboot is done. Needless to say, putting a zero at $3F4 is pretty much equivalent to saying you want reboot on RESET—not a friendly thing to do (but not unknown in the world of commercial software).

In the interest of completeness for this book, Program 15-1 is a short listing of a short ProDOS 8 system program (in fact, a variation on P8.SYSTEM) that traps RESET while it’s waiting to do a quit.

The program should be pretty much self-explanatory. There are no new principles introduced other than the idea of saving the existing RESET vector at $3F2–$3F4 and then restoring it before the program quits. Try out the program and notice how RESET is now controlled by the system program itself.

Program 15-1. ProDOS Reset Demo

```assembly
6 MLI EQU $BF00
7 COUT EQU $FD00
8 HOME EQU $FC58
9 KYBD EQU $C000
10 STROBE EQU $C010
11 RESET EQU $3F2 ; RESET VECTOR
12 CHK EQU $3F4 ; CHECKSUM BYTE
13
14 ORG $2000
15
16 DSK P8.SYSTEM.RESET
17 TYP $FF ; SYSTEM FILE TYPE
18
002000: 20 58 FC 19 START JSR HOME ; CLEAR SCREEN
19
002003: AD F2 03 21 SAVE LDA RESET ; GET OLD VECTOR
```

295
Chapter 15

002006: 8D 90 20 22 STA OLDRESET ; SAVE IT
002009: AD F3 03 23 LDA RESET+1
00200C: 8D 91 20 24 STA OLDRESET+1
00200F: AD F4 03 25 LDA CHK
002012: 8D 92 20 26 STA OLDRESET+2
002015: A9 72 28 SET LDA #$<HANDLER ; OUR RESET ROUTINE
002017: SD F2 03 29 STA RESET
00201A: A9 20 30 LOA #>HANDLER
00201C: SD F3 03 31 STA RESET+l ; ALL SET!
00201F: 49 32 EOR #$A5 ; CHECKSUM
002021: 32 EOR #$AS ; CHECKSUM
002024: B9 5C 20 36 LOOP LDA MSG,Y ; GET CHAR TO PRINT
002029: F0 06 =2031 37 BEQ GETKEY
00202B: 20 F0 FD 38 JSR COUT ; PRINT IT
00202D: C8 39 INY ; NEXT CHAR
00202F: D0 F5 =2026 40 BNE LOOP ; WRAPAROUND PROTECT
002031: 2C 00 C0 42 GETKEY BIT KYBD ; KEYPRESS?
002034: 10 FB =2031 43 BPL GETKEY ; NOPE
002036: 2C 10 C0 44 BIT STROBE ; CLEAR KEYPRESS
002039: AD 90 20 46 RESTORE LDA OLDRESET ; GET ORIG. VECTOR
00203C: 8D F2 03 47 STA RESET
00203F: AD 91 20 48 LDA OLDRESET+1
002042: 8D F3 03 49 STA RESET+1
002045: AD 92 20 50 LDA OLDRESET+2
002048: 8D F4 03 51 STA CHK
00204B: 20 00 BF 53 QUIT JSR MLI ; DO QUIT CALL
00204E: 65 54 DFB $65 ; QUIT CODE
002051: 54 20 55 DA PARMTBL ; ADDRESS OF PARM TABLE
002054: B0 08 =205B 56 BCS ERROR ; NEVER TAKEN
002057: 00 57 BRK ; SHOULD NEVER GET HERE . . .
00205A: 04 59 PARMTBL DFB 4 ; NUMBER OF PARMS
00205C: 00 60 DFB 0 ; QUIT TYPE: 0 = STD QUIT
00205E: 00 61 DA $0000 ; NOT NEEDED FOR STD QUIT
002060: 00 62 DFB 0 ; NOT USED AT PRESENT
002062: 00 63 DA $0000 ; NOT USED AT PRESENT
002064: 64 00 65 ERROR BRK ; WE'LL NEVER GET HERE?
002066: 00 66 PARMTBL DFB 4 ; NUMBER OF PARMS
002068: D0 CC C5 C1 67 MSSG ASC "PLEASE PRESS A KEY =>“,00
00206A: D3 C5 A0 D0 D2 C5 D3 D3
00206C: A0 C1 A0 CB C5 D9 A0 AD
00206E: BE 00
002072: 20 58 FC 69 HANDLER JSR HOME ; OUR RESET HANDLER

296
A Look at Memory Use on the Apple II GS

Bytes $3F5-$3F7 hold a JMP instruction for where to go when an Apple­
soft BASIC program encounters the ampersand ( & ) character. This is a handy
way of adding new commands to BASIC, and was discussed in Chapter 12.

Locations $3F8-$3FA hold a JMP instruction indicating where to go
when Control-Y is pressed in the Monitor. Merlin, the Monitor itself, and the
Classic Desk Accessory called Mangler all use this vector. It’s a convenient way
to re-enter any program from the Monitor.

Bytes $3FB-$3FD create the vector for what are called nonmaskable inter­
rupts. Interrupts are a signal that can occur at any time to tell the computer to
stop what it is doing and to execute a program somewhere else in the com­
puter. Usually, control soon returns to the program that was interrupted. Inter­
rupts allow the computer to seem as though it’s doing more than one thing at
once, and they are an essential part of the Apple II GS environment. There are
different kinds of interrupts. The BRK instruction is, in a way, a kind of inter­
rupt. It tells the microprocessor to stop executing the program it’s in, and to
jump through the vector at $3F0, 3F1. This allows us to run a debugging utility
Chapter 15

at the same time as another program, namely the program being debugged.

A nonmaskable interrupt is an interrupt that can never be ignored (see maskable IRQs, next). This is usually used in dedicated microprocessor devices to detect an outside event. It is provided for in the Apple II GS, but rarely used.

There is another kind of interrupt, called a maskable IRQ (Interrupt Re­quest), that is vectored through location $3FE, 3FF. That type of interrupt can be ignored by the operating program by just using the instruction SEI (SEt In­terrupt disable). This tells the computer to ignore any maskable or discretionary interrupts. This type of interrupt is disabled when the disk is reading or writing to a drive, for example, because timing is very important at that point—an interruption could not be tolerated. Because interrupts are so important on the Apple II GS, it is not advisable to ever turn them off completely. Instead, your program should only turn them off briefly, when absolutely necessary, and then restore the interrupt status to its previous state as soon as possible.

The Text Display
The memory range from $400 to $7FF in bank 0 is used for the 40-column text display (see Figure 15-3). You have seen in previous chapters how a character can be printed to the screen by storing a byte directly in this part of memory.

In this range, 64 bytes in bank 0 are also used as screen holes by certain peripheral cards, as was discussed in a previous chapter.

Figure 15-3. The Text Display: $400 to $7FF

<table>
<thead>
<tr>
<th>$400–$7FF</th>
<th>Text Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>$300–$3FF</td>
<td>Free Space and Vectors</td>
</tr>
<tr>
<td>$200–$2FF</td>
<td>The Input Buffer</td>
</tr>
<tr>
<td>$100–$1FF</td>
<td>The Stack</td>
</tr>
<tr>
<td>$00–$FF</td>
<td>Zero (Direct) Page</td>
</tr>
</tbody>
</table>

Applesoft BASIC
For Applesoft BASIC, a program starts in memory at $800 and grows upward in memory (see Figure 15-4). This is complicated by the fact that two hi-res displays are located in the range of $2000 to $3FFF (hi-res page one), and $4000 to $5FFF (hi-res page two). There are utility routines that will split an Applesoft BASIC program around the hi-res pages, or you can just move the entire program up above the pages in memory.
Applesoft BASIC variables are stored beginning at the end of the BASIC program itself. String data is stored at the top of memory and works downward as new strings are defined.

Figure 15-4. Applesoft BASIC Programs: $800 to $95FF

Bank 0

$FFFF

$BF00–$BFFF

$BEFF

$9600

$95FF

$5FFF

$4000

$3FFF

$2000

$800

$7FF

$400

$3FF

$000
Chapter 15

**BASIC.SYSTEM: $9600 to $BEFF**

The next major memory boundary occurs at $9600, which is the default lower boundary of BASIC.SYSTEM under ProDOS 8. Although BASIC.SYSTEM varies this lower boundary as files are opened and closed, this gives you a general idea of how much memory is normally available to an Applesoft BASIC program. You’ll recall that BASIC.SYSTEM is used as a middleman between Applesoft BASIC and ProDOS 8. It is not required, nor is it in memory when a ProDOS system file is running.

**The ProDOS Global Page: $BF00 to $BFFF**

The area from $BF00 to $BFFF is called the ProDOS global page and is used to store information about the computer state under ProDOS that other programs may want to access. The MLI places all information that might be useful to a system program in this area of memory. You may want to consult the books specific to ProDOS 8 listed in Appendix D for more information on the ProDOS global page and other ProDOS 8 functions.

If you are running a ProDOS 8 system program, such as P8.SYSTEM, then BASIC.SYSTEM is not loaded into memory and your system file will be loaded starting at $2000. Obviously, if you intend to use the hi-res pages in such a program, you must relocate your program to some other nonconflicting part of memory. This can be done with a memory move routine in your own program, or using the 65816 memory move instructions. (See the 65816 instructions reference section in Appendix A).

**I/O ROM**

The next important area of memory is that from $C000 to $CFFF. This is typically assigned totally to hardware in the form of either softswitches ($C000 to $C0FF) or ROM-based firmware on the peripheral interface cards, such as printer cards. See Figure 15-5.

The memory from $C100 to $C7FF has been allocated to the seven expansion slots (or built-in ports) in $100-byte (one page) increments. The normal implementation is for the peripheral card to have a ROM program on it, which appears in the address space corresponding to its slot. For example, the firmware for the printer port, which on the Apple II GS appears as though it is in slot 1, appears in memory at $C100 to $C1FF. For the mouse interface, which is assigned to slot 4, the routines appear starting at $C400. When you type PR#4, for example, the computer is programmed to do a JMP to $C400. This is presumed to be an initialization routine in the ROM associated with the device assigned to slot 4.
In addition, the area from $C800 to $CFFF (2K) is a shared address space for all slots. When a device is turned on, it maps its own ROM into the complete space in this 2K area. Obviously, if two cards are active at the same time, and try to each use their own ROMs in this space, a conflict will occur. Therefore, there is a protocol for cards recognizing requests from other cards to switch in a new ROM assignment.

Figure 15-5. I/O ROM Space: $C000 to $CFFF
Chapter 15

Bank-Switched RAM and ROM

The area from $D000 to $FFFF gets really interesting. In Applesoft BASIC, the space starting at $D000 and going up to $F7FF is assigned to the Applesoft BASIC ROM routines. And $F800 to $FFFF is the area for Monitor routines. However, there is another 16K of RAM also assigned to this space. It is accessed by the use of a softswitch which switches out the Applesoft BASIC and Monitor ROMs, and reroutes addressing to the RAM memory. This area is usually occupied by ProDOS. When a ProDOS system file (other than BASIC.SYSTEM) is running, Applesoft BASIC is not even needed, so this part of memory stays set to RAM for most, if not all, of the time.

It's pointless trying to decide which memory, RAM or ROM, is the "real" memory from $D000 to $FFFF. In fact, all of the memory in the Apple II GS is scattered about in different physical locations. For some parts of memory, not even all the bits that make up a single byte are found in a single chip. Like the 2K of expansion ROM in the $C800-$CFFF range, this remapping is done all electronically. Memory is not actually moved, but rather, all addressing to that range is redirected to the proper physical memory (RAM or ROM).

It gets even stranger though. If you recheck that address range, $D000-$FFFF, you'll see that it corresponds to only 12K ($FFFF to $D000 = $3000 = 3 * 4096 = 12288 bytes). But 16K is allocated to this area. And it is. Still another softswitch is used to switch the extra 4K of RAM from $D000 to $DFFF with the other 4K in that space. Thus, using exactly the same addresses in the range of $D000 to $DFFF, it is possible to be looking at one of three distinctly different sets of data: Applesoft BASIC ROM, RAM, or bank-switched RAM. By the way, it's in this alternate RAM bank that ProDOS 8 stores the routine ultimately called by the ProDOS quit command (called the quit code). By rewriting this area, programs under ProDOS 8 can establish their own quit procedures or dead-end the return process altogether (not recommended).

The actual softswitch protocols for switching between the RAM and ROM banks are varied and somewhat involved. You may wish to consult Apple IIe and Apple II GS technical manuals for all the details on how to use bank-switched memory. Under ProDOS 16, with the long addressing modes of the 65816, bank-switching is required only for switching in and out the extra 4K (Applesoft BASIC being ignored). If it's any consolation, under ProDOS 16, it is unlikely you'll ever have to even concern yourself with this part of memory.

Two for the Price of One:
The Apple IIe/IIC and Bank 1

When the Apple IIe and (later) the IIc machines were introduced, they increased the amount of memory available with a elegantly simple solution: The first 64K of memory was just doubled with matching RAM, as shown in Figure 15-6.
A Look at Memory Use on the Apple IIgs

Figure 15-6. The First 64K

<table>
<thead>
<tr>
<th>Bank 0</th>
<th>Bank 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FFFF</td>
<td>Monitor ROM routines</td>
</tr>
<tr>
<td>$F800</td>
<td></td>
</tr>
<tr>
<td>$F7FF</td>
<td></td>
</tr>
<tr>
<td>$E000</td>
<td>Applesoft BASIC</td>
</tr>
<tr>
<td>$DFFF</td>
<td>ROM routines</td>
</tr>
<tr>
<td>$D000</td>
<td></td>
</tr>
<tr>
<td>$C000-$CFFF</td>
<td>I/O ROM and softswitches</td>
</tr>
<tr>
<td>$BF00-$BFFF</td>
<td>ProDOS 8 Global Page</td>
</tr>
<tr>
<td>$BEFF</td>
<td></td>
</tr>
<tr>
<td>$9600</td>
<td>BASIC.SYSTEM</td>
</tr>
<tr>
<td>$5FFF</td>
<td>Hi-Res Page Two</td>
</tr>
<tr>
<td>$3FFF</td>
<td>Hi-Res Page One</td>
</tr>
<tr>
<td>$2000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Applesoft BASIC Program</td>
</tr>
<tr>
<td>$800</td>
<td></td>
</tr>
<tr>
<td>$400-$7FF</td>
<td>Text Screen</td>
</tr>
<tr>
<td>$3FF</td>
<td></td>
</tr>
<tr>
<td>$000</td>
<td></td>
</tr>
</tbody>
</table>
The Apple IIe and IIc are called 128K machines because memory is set up in two parallel banks of memory, each addressed from $0000 to $FFFF. This is almost like having two computers side by side. Although each has its own zero page, stack, and other areas, these areas are not really very usable as such, and are more relevant in the IIe and IIc environment. It’s unlikely that you will ever have to be concerned with these functions on the Apple IIgs.

In bank 1 memory, the range from $400 to $7FF is used to augment the 40-column text display, thus creating the 80-column display (every-other character is from the alternate bank). You have seen in Chapter 10 how a character can be printed to the screen by storing a byte directly in this part of memory. Also, you saw how the identical address range in bank 1 was used to complete the 80-column display and how clearing this part of memory was required for a full 80-column screen clear. In the 80-column display, each byte from one bank is interleaved with each byte from the other. All even character-position bytes (0, 2, 4, and so on) are in bank 1 (AUXMEM), and all odd character-position bytes (1, 3, 5, and so on) are in bank 0.

Double hi-res graphics were actually discovered by accident on the Apple IIe and IIc after the implementation of the 80-column text display. It was discovered that a similar interleaving of bytes would double the graphics screen resolution of hi res from 280 pixels horizontally to 560 for double hi res.

The gap in the memory map for bank 1 is there to indicate that when you try to address, for example, $C300 in bank 1, you are actually addressing $C300 in bank 0. As mentioned earlier, it’s possible, through simple wiring, to make the contents of any RAM or ROM address appear at any one, or several, addresses. It’s a little like call forwarding: You can request access to an address; the computer will determine where that ultimately takes you.

The Apple IIgs: 256 for the Price of One
The Apple IIgs was designed to not only expand the capabilities of the earlier Apple IIe and IIc machines, but also to retain compatibility with software designed to run on those machines as well. The result is a system that retains the underlying structure of the IIe and IIc environment, yet one that extends it in a way that makes the transition to ProDOS 16 programming very easy.

Some people try to conceptualize what goes on in the Apple IIgs between ProDOS 8 and ProDOS 16 as operating modes that are exclusive of one another. While it’s true that Applesoft BASIC cannot run under ProDOS 16, many Apple IIgs tools can be called from within even an Applesoft BASIC program. You have also seen that although the e-bit’s name, emulation bit, implies a separate and distinct operating mode, in fact, it serves mainly to enable the 16-bit width selection of data handling. Therefore, try to avoid the temptation...
to restrict the IIGS environment into separate operating modes, and rather realize that it is a superset, not a separate set, of the earlier IIe and IIC computers.

The Apple IIGS has 256 ($00 to $FF) addressable (though not necessarily installed) banks of 64K memory. Banks $00 through $E1 are RAM. Banks $FE and $FF are ROM memory that include not only the Applesoft BASIC and Monitor routines, but also the Apple IIGS ROM toolset as well. The area from $F0 to $FD is reserved for expansion ROM, possibly in the form of a fast-boot ROM disk, or of permanently resident applications.

In addition, all memory except for banks $E0 and $E1 is what is called fast memory—all accessing to it can be done at the faster 3.1 MHz speed of the 65816, as opposed to the 1 MHz speed used to access banks $E0 and $E1.

The reason for this difference is that the video circuitry on the Apple relies on a certain timing and speed of operation to properly generate the video display. If this were arbitrarily sped up, your video monitor would no longer work. Remember that the video data areas in the Apple IIe and IIc are the text and hi-res pages in banks $00 and $01. At the same time, Apple knew people would expect that programs running in these banks would be faster because of the faster processor (clock speed) operation. Now here’s a dilemma: how to speed up the program in banks 0 and 1, and leave the video display areas at a slower rate.

The solution was to do something called shadowing, in which data written to the video areas ($400 to $7FF and $2000 to $5FFF) of banks 0 and 1 would be automatically reproduced electronically, in the same memory range in banks $E0 and $E1. This way, as your program writes bytes to the text screen in bank 0, the data automatically appears in bank $E0. The video circuitry looks only at bank $E0, and it interacts only at the slower speed. Hence, everything works out. On the Apple IIGS, the video display you see is the contents only of banks $E0 and $E1. In fact, if you turn shadowing off, it’s possible to write to the text or graphics pages in banks 0 and 1, and to have the results remain entirely invisible as the user continues to view the displays generated from banks $E0 and $E1.

Not every byte banks 0 and 1 are automatically shadowed into banks $E0 and $E1. The areas actually shadowed are controlled by a status byte at $C035, called the shadow register. Bits set and cleared in this byte determine which parts of memory will be automatically copied into banks $E0 and $E1 as they are written to in banks 0 and 1.

The control bits are as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Text page one</td>
</tr>
<tr>
<td>1</td>
<td>Hi-res page one</td>
</tr>
<tr>
<td>2</td>
<td>Hi-res page two</td>
</tr>
<tr>
<td>3</td>
<td>Auxmem for both hi-res pages (double hi-res)</td>
</tr>
</tbody>
</table>
Chapter 15

Bit 4  Super hi-res area ($2000–$9FFF) in bank 1
Bit 5  Reserved; always write as zero
Bit 6  Exchange 4K language card RAM with $C000 space
       (Write zero if you know what’s good for you!)
Bit 7  Reserved: always write as zero

If you’re wondering where the Applesoft BASIC and Monitor ROM routines are, the answer is in bank $FF. If you’ve gone to the Monitor and typed 00/FC58L to list the HOME routine, you probably didn’t find much. That is because under ProDOS 8, FF/D000 to FF/FFFF is remapped into bank 0 when the Applesoft BASIC and Monitor ROMS are being selected by a program there. However, in the normal ProDOS 16 environment, you won’t find the Applesoft BASIC routines in bank 0, but rather in bank $FF.

If you compare the size of the 4K bank-switched expansion RAM to the $C000 to $CFFF space, you’ll notice that they are each 4K in size. In the Apple IIGS, it’s possible to flip this expansion RAM down into the $C000 to $CFFF space in every bank of RAM, including bank 0. However, this is very tricky because you no longer have access to any of the softswitches or system status bytes that are so important to system control. Normally, the $C000 to $CFFF I/O area is mapped into each bank so that a program running in any bank can directly access the I/O space.

The Apple IIGS also has a new display mode, called super hi-res graphics, that uses the memory area in bank $E1 from $2000 to $9FFF (32K). It is possible to turn on shadowing for this display also, so that any writing to bank 1 in this memory range will automatically be copied into bank $E1 so that it will be in the video display.

Banks $E0 and $E1 are the main parts of memory used by the Apple IIGS internal system software and the system tools for such things as the super hi-res display, data storage and work areas for AppleTalk, the Miscellaneous Tool set, the Memory Manager, Text Tools, and more.

In general, it’s best not to put any code directly here, but rather to use the Memory Manager, which will find and allocate memory as it’s available throughout the entire computer.

For those who are curious, Figure 15-7 shows some of the memory use of banks $E0 and $E1.
A Look at Memory Use on the Apple IIgs

Figure 15-7. Banks $E0 and $E1

<table>
<thead>
<tr>
<th>Bank $E0</th>
<th>Bank $E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FFFF</td>
<td>AppleTalk routines</td>
</tr>
<tr>
<td></td>
<td>and buffers</td>
</tr>
<tr>
<td>$E000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I/O ROM and softswitches</td>
</tr>
<tr>
<td>$DFFF</td>
<td></td>
</tr>
<tr>
<td>$D000</td>
<td></td>
</tr>
<tr>
<td>$C000-$CFFF</td>
<td></td>
</tr>
<tr>
<td>$BFFF</td>
<td></td>
</tr>
<tr>
<td>$A000</td>
<td></td>
</tr>
<tr>
<td>$9FFF</td>
<td></td>
</tr>
<tr>
<td>$6000</td>
<td></td>
</tr>
<tr>
<td>$5FFF</td>
<td></td>
</tr>
<tr>
<td>$4000</td>
<td></td>
</tr>
<tr>
<td>$3FFF</td>
<td></td>
</tr>
<tr>
<td>$2000</td>
<td></td>
</tr>
<tr>
<td>$1FFF</td>
<td></td>
</tr>
<tr>
<td>$1E00</td>
<td></td>
</tr>
<tr>
<td>$1DDF</td>
<td></td>
</tr>
<tr>
<td>$1DD8</td>
<td></td>
</tr>
<tr>
<td>$1DD7</td>
<td></td>
</tr>
<tr>
<td>$1DD0</td>
<td></td>
</tr>
<tr>
<td>$1DCF</td>
<td></td>
</tr>
<tr>
<td>$1DB0</td>
<td></td>
</tr>
<tr>
<td>$DAF</td>
<td></td>
</tr>
<tr>
<td>$19B8</td>
<td></td>
</tr>
<tr>
<td>$19B7</td>
<td></td>
</tr>
<tr>
<td>$15FE</td>
<td></td>
</tr>
<tr>
<td>$15FD</td>
<td></td>
</tr>
<tr>
<td>$15CD</td>
<td></td>
</tr>
<tr>
<td>$15CC</td>
<td></td>
</tr>
<tr>
<td>$15C1</td>
<td></td>
</tr>
<tr>
<td>$15C0</td>
<td></td>
</tr>
<tr>
<td>$15AA</td>
<td></td>
</tr>
</tbody>
</table>

| Free Memory      | Super Hi-Res Display   |
|                  |                        |
|                  | DHR Interleave Page Two|
|                  |                        |
|                  | DHR Interleave Page One|
|                  |                        |
|                  | Serial Input Buffer    |
|                  |                        |
|                  | Misc. Tools Buffer     |
|                  |                        |
|                  | Sound Variables Buffer |
|                  |                        |
|                  | Reserved               |
|                  |                        |
|                  | Memory Mgr. Buffer     |
|                  |                        |
|                  | Reserved               |
|                  |                        |
|                  | Serial Port Variables  |
|                  |                        |
|                  | Text Tools Data        |

307
Chapter 15

<table>
<thead>
<tr>
<th>$15A9</th>
<th>Serial Port Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>$158A</td>
<td>ADB Address &amp; attribute list</td>
</tr>
<tr>
<td>$1589</td>
<td>SmartPort usage</td>
</tr>
<tr>
<td>$154A</td>
<td>AppleTalk Data Storage</td>
</tr>
<tr>
<td>$1549</td>
<td>Serial Port Storage</td>
</tr>
<tr>
<td>$14E2</td>
<td>ADB Storage</td>
</tr>
<tr>
<td>$14E1</td>
<td>Misc. Tools usage</td>
</tr>
<tr>
<td>$1000</td>
<td>Disk Transfer Buffer</td>
</tr>
<tr>
<td>$FFF</td>
<td>80-Column Interleave Text Screen Page Two</td>
</tr>
<tr>
<td>$FFB</td>
<td>80-Column Interleave Text Screen Page One</td>
</tr>
<tr>
<td>$FFA</td>
<td>Clock Buffer</td>
</tr>
<tr>
<td>$FD6</td>
<td>ADB Interrupt Queue</td>
</tr>
<tr>
<td>$FD5</td>
<td>Tool Locator Variables</td>
</tr>
<tr>
<td>$FD0</td>
<td>Battery RAM Buffer</td>
</tr>
<tr>
<td>$FCF</td>
<td>Mouse Clamp Data</td>
</tr>
<tr>
<td>$C00</td>
<td>User access vectors-Monitor Entry Points</td>
</tr>
<tr>
<td>$BFF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Text Screen Page Two

<table>
<thead>
<tr>
<th>Text Screen Page One</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desk</td>
</tr>
<tr>
<td>Accessory Buffer</td>
</tr>
<tr>
<td>$3EF</td>
</tr>
<tr>
<td>$3C0</td>
</tr>
<tr>
<td>$3BF</td>
</tr>
<tr>
<td>$300</td>
</tr>
<tr>
<td>$2FF</td>
</tr>
<tr>
<td>$2C0</td>
</tr>
<tr>
<td>$2BF</td>
</tr>
<tr>
<td>$2B8</td>
</tr>
<tr>
<td>$2B7</td>
</tr>
<tr>
<td>$000</td>
</tr>
</tbody>
</table>
Chapter 16

The Apple IIGS Toolbox
Chapter 16

The Apple II GS Toolbox

For an assembly language programmer, one of the nicest things about the Apple II GS is the enormous set of built-in (and loadable) routines that can be called by an application. This simplifies the programming process in that you do not have to write as many of the supporting routines to clear the screen, to use graphics, or even to catalog disks. All of these functions are available in the Apple II GS Toolbox.

Banks $FE and $FF contain a number of the tools used most often. Others, including tools you create yourself, can be loaded in from disk.

The routines available are organized by tool sets, which are a group of related Toolbox commands. Each Toolbox command is called through a common vector, much the same way that ProDOS 8 and 16 commands were called. The main difference for the Toolbox commands is that information for each command is passed back and forth between the application and the command using the stack. The entry point for the Toolbox routines is at $E1/0000, and is called in the following fashion:

1. Push zeros (or anything) on the stack to make room for any results that may be returned by the routine. Nothing need be pushed on the stack if there are no results to be returned.
2. Load the X register with a two-byte value that identifies the tool set to use ($00 to $FF), and the command value ($00 to $FF). Thus, to call tool set number 5, and use command $27, you would use the instruction:
   
   LDX #$2705 ; TOOL $05, COMMAND = $27

3. Do a JSL $E10000 to call the Toolbox dispatcher.
4. Pull any returned results off the stack. Not required if there are no results returned.
5. Check the carry flag for an error. The carry will be set if there was a problem executing the Toolbox command, and the Accumulator will hold the error code. If there was no error, the carry will be clear, and the Accumulator will be set to zero.
### Table 16-1. The First 28 Tools

<table>
<thead>
<tr>
<th>Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong> Tool Locator</td>
</tr>
<tr>
<td><strong>2</strong> Memory Manager</td>
</tr>
<tr>
<td><strong>3</strong> Miscellaneous Tools</td>
</tr>
<tr>
<td><strong>4</strong> QuickDraw II</td>
</tr>
<tr>
<td><strong>5</strong> Desk Manager</td>
</tr>
<tr>
<td><strong>6</strong> Event Manager</td>
</tr>
<tr>
<td><strong>7</strong> Scheduler</td>
</tr>
<tr>
<td><strong>8</strong> Sound Tools</td>
</tr>
<tr>
<td><strong>9</strong> Apple DeskTop Bus</td>
</tr>
<tr>
<td><strong>11</strong> Integer Math Tools</td>
</tr>
<tr>
<td><strong>12</strong> Text Tools</td>
</tr>
<tr>
<td><strong>13</strong> GS System Tools</td>
</tr>
<tr>
<td><strong>14</strong> Window Manager</td>
</tr>
<tr>
<td><strong>15</strong> Menu Manager</td>
</tr>
<tr>
<td><strong>16</strong> Control Manager</td>
</tr>
<tr>
<td><strong>17</strong> Loader</td>
</tr>
<tr>
<td><strong>18</strong> QuickDraw Auxiliary</td>
</tr>
<tr>
<td><strong>19</strong> Print Manager</td>
</tr>
<tr>
<td><strong>20</strong> Line Edit</td>
</tr>
<tr>
<td><strong>21</strong> Dialog Manager</td>
</tr>
<tr>
<td><strong>22</strong> Scrap Manager</td>
</tr>
<tr>
<td><strong>23</strong> Standard File Tools</td>
</tr>
<tr>
<td><strong>24</strong> Disk Utilities</td>
</tr>
<tr>
<td><strong>25</strong> Note Synthesizer</td>
</tr>
<tr>
<td><strong>26</strong> Note Sequencer</td>
</tr>
<tr>
<td><strong>27</strong> Font Manager</td>
</tr>
<tr>
<td><strong>28</strong> List Manager</td>
</tr>
</tbody>
</table>
There are, at this writing, 28 tool sets for the Apple IIGS. Programmers can also define their own tool sets, and more tools are sure to be released by Apple as time goes on. Table 16-1 shows the first 28 tools.

Tools 1 through 13 are ROM-based, that is, they're stored in the upper two banks of ROM memory in the Apple IIGS, banks $FE and $FF. All other tools must be loaded into memory using the Tool Locator LoadTool command.

Although it would be impossible in this book to cover every single tool in detail, we can get a good understanding of the general principles involved by exploring a half-dozen or so of the most often-used tools.

Let's start with the most important, the Tool Locator.

**The Tool Locator**

The Tool Locator tool set is the set of commands used to load RAM tools from disk, and to manage all the other tools currently in the system. The main entry point, $E10000 is itself part of the Tool Locator.

Generally speaking, each tool must be started up by an application when the application first runs, and then later shut down before the application does its quit command. This is so that memory used by a particular tool set may by freed for use by other tools or applications once it is no longer needed.

In practice, there are a number of tools, like the Tool Locator, that are always on because they're used by the Apple IIGS system itself. The computer itself could not function without the Tool Locator, so even if you try to shut it down, nothing actually happens. As a result, many programs do not specifically start up or shut down the Tool Locator, or tools like the Text Tools, as they probably should.

The *probably* is because, at the present, it doesn't make any difference one way or the other. However, if Apple ever changes these apparently safe tools and requires some sort of initialization, programs which don't follow the proper procedures may not work.

Table 16-2 lists some of the commands in the Tool Locator tool set.

The first six tool-set functions are a constant for every tool. Tool call $04xx, for example, will always return the version number of the tool set called, where xx represents its tool number. For some tools, the answer may be obvious, such as whether the Tool Locator is active (tool call $0601). The Tool Locator has to be active or the call to find out whether it was or not wouldn't work. Despite this, there is value in establishing a standard for tools whose status may not be active, and these six calls are guaranteed to be supported in every IIGS tool set.

Tool calls $0901 through $0D01 deal with internal work areas and pointers to particular tool calls. It's unlikely you'll ever need to use any of these.
### Table 16-2. Selected Commands in the Tool Locator Tool Set

<table>
<thead>
<tr>
<th>Command Value</th>
<th>Command Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0101</td>
<td>BootInit</td>
<td>Initialized by system on boot. Not used by application.</td>
</tr>
<tr>
<td>$0201</td>
<td>TLStartUp</td>
<td>Starts up Tool Locator.</td>
</tr>
<tr>
<td>$0301</td>
<td>TLShutDown</td>
<td>Shuts down Tool Locator.</td>
</tr>
<tr>
<td>$0401</td>
<td>TLVersion</td>
<td>Returns version number of Tool Locator.</td>
</tr>
<tr>
<td>$0501</td>
<td>TLReset</td>
<td>Re-initializes Tool Locator and all other ROM-based tool sets on a system reset.</td>
</tr>
<tr>
<td>$0601</td>
<td>TLEntry</td>
<td>Indicates whether Tool Locator is active.</td>
</tr>
<tr>
<td>$0701</td>
<td>GetTSPtr</td>
<td>Get Function Pointer Table of a specified tool set.</td>
</tr>
<tr>
<td>$0801</td>
<td>SetTSPtr</td>
<td>Set Function Pointer Table of a specified tool set.</td>
</tr>
<tr>
<td>$0901</td>
<td>GetFuncPtr</td>
<td>Get pointer to a specified function in a tool set.</td>
</tr>
<tr>
<td>$0A01</td>
<td>GetWAP</td>
<td>Get pointer to work area for a tool set.</td>
</tr>
<tr>
<td>$0B01</td>
<td>SetWAP</td>
<td>Set pointer to work area for a tool set.</td>
</tr>
<tr>
<td>$0C01</td>
<td>LoadTools</td>
<td>Load a group of RAM-based tools from disk.</td>
</tr>
<tr>
<td>$0D01</td>
<td>LoadOneTool</td>
<td>Load a single RAM-based tool from disk.</td>
</tr>
<tr>
<td>$0E01</td>
<td>UnloadOneTool</td>
<td>Release memory for a single RAM-based tool.</td>
</tr>
<tr>
<td>$0F01</td>
<td>TLMountVolume</td>
<td>DeskTop (graphics) message usually for asking user to insert system disk. May be used for other messages.</td>
</tr>
<tr>
<td>$1001</td>
<td>TLTextMountVolume</td>
<td>Text screen message usually for asking user to insert system disk. May be used for other messages.</td>
</tr>
<tr>
<td>$1101</td>
<td>SaveTextState*</td>
<td>Saves current text screen and switches display to text screen. Usually for TLTextMountVolume, but could be used for other reasons.</td>
</tr>
<tr>
<td>$1201</td>
<td>RestoreTextState*</td>
<td>Restores saved state of text screen.</td>
</tr>
<tr>
<td>$1301</td>
<td>MessageCenter*</td>
<td>Allows common point to pass text data between different applications.</td>
</tr>
</tbody>
</table>

* Available on ProDOS 16 versions 1.2 or later.

LoadTools ($0E01) and LoadOneTool ($0F01) are used to load RAM-based tools from disk. These tools are always assumed to be in the TOOLS subdirectory of the SYSTEM folder on the boot disk.

UnloadOneTool ($1001) is used to free memory used by a RAM-loaded tool so that other applications aren’t hindered by memory used by previous applications and their tools.

TLMountVolume ($1101) and TLTextMountVolume ($1201) are routines to put a message on the screen when one of the LoadTools commands can’t find the system disk. One routine is for the super hi-res display, and the other is for the 40-column text screen. Note that QuickDraw and the Event Manager must have been started up already for the graphics-based TLMountVolume command to work.
If you were using the text screen, then TLTextMountVolume would destroy whatever was on the screen, so two other commands, SaveTextScreen ($1301), and RestoreTextScreen ($1401) have been provided to save and restore the screen state when you used TLTextMountVolume.

The last routine MessageCenter ($1501), is used to pass brief text data between applications. For example, suppose you had written two applications that changed back and forth between each other. MessageCenter would provide a way to communicate things like the active data disk prefix, the user's name, or whatever else you wanted, while avoiding the necessity of using a disk text file or the Scrap Manager to temporarily store the information.

The Tool Locator is actually pretty boring as far as suggesting interesting demo programs, but this is a good time to show how to call a tool from both Applesoft BASIC or ProDOS 8 and ProDOS 16 applications.

Program 16-1 is another variation on the P8.SYSTEM program that shows how to use the TLTextMountVolume command to print any message on the text screen.

Program 16-1. ProDOS 8 Tool Locator Demo

1  ********************************************************************************
2  *       PRODOS 8 TOOL LOCATOR DEMO       *
3  *               MERLIN ASSEMBLER            *
4  ********************************************************************************
5
6  =BF00  6  MLI EQU $BF00
7  =FDFO  7  COUT EQU $FDFO
8  =FC58  8  HOME EQU $FC58
9  =C000  9  KYBD EQU $C000
10 =C010 10  STROBE EQU $C010
11
12
13
14 =DSK 14  P8.TOOL.DEMO
15  TYP $FF ; SYSTEM FILE TYPE
16
17
18
19 002000:  A9 4C 18  SETQUIT LDA #$4C ; JMP INSTRUCTION
20 002002:  8D F8 03 19  STA $3F8 ; CTRL Y VECTOR
21 002005:  A9 5C 20  LDA <QUIT ; LOW BYTE OF QUIT ADDR.
22 002007:  8D F9 03 21  STA $3F9 ; LOW BYTE OF CTRL-Y VECTOR
23 00200A:  A9 20 22  LDA >QUIT
24 00200C:  8D FA 03 23  STA $3FA ; HIGH BYTE OF CTRL-Y VECTOR
25 00200F:  18 24  MODE16 CLC
26 002010:  FB 26  XCE ; ENABLE 16 BIT SELECT
27 002011:  C2 30 27  REP $30 ; FULL 16 BIT MODE
28
29 002013:  A2 01 02 29  STARTUP LDX #$0201 ; TLStartUp
Chapter 16

002016: 22 00 00 E1 30 JSL $E10000 ; START UP TOOL LOCATOR
00201A: 90 02 201E 31 BCC BOX ; TOOL ERROR (NOT LIKELY)
00201C: 00 00 32 BRK $00
00201E: F4 00 00 34 BOX PEA $0000 ; PUSH SPACE FOR RESULT
002021: F4 00 00 36 PEA "LINE1" ; HIGH WORD OF LINE1
002024: F4 00 00 37 PEA LINE1 ; LOW WORD OF LINE1
002027: F4 00 00 38 PEA "LINE2" ; TOOL ERROR (NOT LIKELY)
00202A: F4 00 00 39 PEA LINE2
00202D: F4 00 00 40 PEA "BUTTON1"
002030: F4 00 00 41 PEA BUTTON1
002033: F4 00 00 42 PEA "BUTTON2"
002036: F4 A9 20 43 PEA BUTTON2
002039: A2 01 12 44 LDX #$1201 ; TLTextMountVolume
00203C: 22 00 00 E1 46 JSL $E10000 ; DO TOOLBOX CMD
002040: 90 02 2044 47 BCC TEST ; HIGH WORD OF LINE1
002042: 00 00 48 BRK $00
002044: 68 50 TEST PLA ; GET BUTTON VALUE
002045: C9 01 00 51 CMP #1 ; BUTTON 1 = QUIT
002048: F0 05 204F 52 BEQ SHUTDOWN ; BUTTON 1 = QUIT
00204A: EE 98 20 54 INC LINE2+20 ; INCREMENT COUNTER
00204D: 80 CO 200F 55 BRA MODE16 ; TRY AGAIN ...
00204F: A2 01 03 57 SHUTDOWN LDX #$0301 ; TLShutDown
002052: 22 00 00 E1 58 JSL $E10000 ; TOOL LOCATOR SHUTDOWN
002055: 90 02 205A 59 BCC MODE8
002058: 00 00 60 BRK $00
00205A: 38 62 MODE8 SEC ; BACK TO 8 BITS ...
00205B: FB 63 XCE
00205C: 20 00 BF 64 QUIT JSR MLI ; DO QUIT CALL
00205F: F0 65 66 DFB $65 ; QUIT CODE
002060: 66 20 67 DA PARM_TBL ; ADDRESS OF PARM TABLE
002062: B0 09 206D 68 BCS ERROR ; NEVER TAKEN
002064: 00 00 69 BRK $00 ; SHOULD NEVER GET HERE ...
002066: 04 71 PARM_TBL DFB 4 ; NUMBER OF PARMS
002067: 00 72 DFB 0 ; QUIT TYPE: 0 = STD QUIT
002068: 00 00 73 DA $0000 ; NOT NEEDED FOR STD QUIT
00206A: 00 74 DFB 0 ; NOT USED AT PRESENT
00206B: 00 00 75 DA $0000 ; NOT Used AT PRESENT
00206D: 00 00 76 ERROR BRK $00 ; WE'LL NEVER GET HERE?
00206F: 14 D0 D2 CF 79 LINE1 STR "PRODOS 8 TEST SYSTEM"
002073: C4 CF D3 A0 00
002077: B8 A0 D4 C5

316
Apple IIGS tools can be called from ProDOS 8 (or even from Applesoft BASIC). The only requirement is that the accumulator and registers both be set to the 16-bit mode before any actual tool calls are done.

Lines 25–27 of Program 16-1 switch to the 16-bit mode. We could’ve delayed the transition, but since a tool call is the first thing the program does, the mode switch must be at the beginning. The Tool Locator tool set is then started up on lines 29 and 32.

In general, every tool used by a program should be started at the beginning of the program and shut down before the program quits. At the present time, not every tool absolutely requires this, but it is a good practice because it insures that your program will be compatible with future versions of the tools.

TLTextMountVolume has the following parameter requirements.

First, each of these must be pushed onto the stack:

1. A two-byte word to make room for the result (which key is pressed).
2. A 4-byte, long-address pointer to the string that will be used as the title for the message box.
3. A 4-byte, long-address pointer to the string that will be used as the secondary line of text in the message box.
4. A 4-byte, long-address pointer to the string that will be used as the text for button 1 in the message box.
5. A 4-byte, long-address pointer to the string that will be used as the text for button 2 in the message box.

When each of these has been pushed onto the stack, the X register will load with the immediate value #$1201 (tool call $12, tool set number $01). This tool sets the display mode to the 40-column text display and draws a dialog box with two lines of text and two button indicators. The user may press the Return key for one choice and the Escape key for the other. Normally, Return accepts the request, or message, while Escape indicates the user wants to cancel.

If you’ve ever launched a program with the boot-up system disk not in the drive, you have probably seen this dialog box used in its intended fashion, that is, to prompt the user to insert a desired disk volume.

The tool call returns with a value on the stack equal to either 1 for button 1 pressed or 2 for button 2. This value can then be pulled off the stack. Figure 16-1 and 16-2 are diagrammatical ways of showing the input and output parameters for a tool call.

Figure 16-1. Input and Output Parameters for Tool Calls: Stack Before Call

<table>
<thead>
<tr>
<th>Previous Contents</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Space for Result</td>
<td>Word: Allow space for result.</td>
<td></td>
</tr>
<tr>
<td>Line 1 Pointer</td>
<td>Long: Pointer to the string appear at top of box.</td>
<td></td>
</tr>
<tr>
<td>Line 2 Pointer</td>
<td>Long: Pointer to the string appear below line 2.</td>
<td></td>
</tr>
<tr>
<td>Button1Pointer</td>
<td>Long: Pointer to the string appear for Button1.</td>
<td></td>
</tr>
<tr>
<td>Button2Pointer</td>
<td>Long: Pointer to the string appear for Button 2.</td>
<td>←SP: Stack pointer after setup.</td>
</tr>
</tbody>
</table>

318
The Apple IIgs Toolbox

Figure 16-2. Input and Output Parameters for Tool Calls: The Stack After Call

<table>
<thead>
<tr>
<th>Previous Contents</th>
<th>Word: Specifies which button was chosen.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Which Button</td>
<td>←SP: Stack Pointer after return from routine.</td>
</tr>
</tbody>
</table>

Lines 34–43 push the needed values on the stack for the call. Remember PEA (Push Effective Absolute address) pushes the value of the operand on the stack (2 bytes). Line 34 pushes a zero on the stack to reserve space for the result returned by the Toolbox routine. Next, line 36 pushes the high word, or bank value, of the address of the string to be used as the title of the dialog box. Notice the use of the caret symbol (^), a Merlin 16 addressing protocol, to signify the high-order portion of the address assigned to the label LINE1. Line 37 then pushes the low-order word for the address on the stack. No special symbol is required to differentiate the low-order word. Because data may be located anywhere in memory, all Toolbox commands generally require four-byte pointers to data.

When the stack is prepared, line 45 loads the X register with the proper command value ($1201), and line 46 does the actual call to the tool function dispatcher at $E10000.

Assuming no error occurs, line 50 pulls the value for the choice off the stack and stores it for a comparison to be made shortly.

**Error Codes**

Then, the carry is checked to see if an error has occurred. For simplicity’s sake, a BRK is used as our error handler. By examining the Accumulator in such a case, you can see what error occurred. Naturally, a more friendly error-handler is recommended for any program of your own.

There are a few particular error codes you should know about.

Error Code = $xx01 This tool set is not available. 
(You haven’t loaded the tool.)

= $xx02 This tool command is not available. 
(You used the wrong command number.)

In addition, each tool set and routine has a variety of other individual errors that may be returned, depending on the actual call made. For example,
Chapter 16

here are some of the possible errors when using the Tool Locator:

Error Code = $0110  Version error (a tool with the minimum version number could not be found).
  = $0100  Couldn't find system startup (boot) volume.
  = $xxxx  Any System Loader/ProDOS error is returned unchanged.

In general, error codes return the tool set that generated the error in the high-order byte, and the actual error code in the low-order byte. Not every tool call has a potential error. Some, like TLStartUp, will never generate an error.

You'll also notice that the check for the error has been done after any results are pulled off the stack. If you do the test first, and then branch to an error-handling routine without removing data from the stack, any subsequent RTS or RTL instructions will be executed incorrectly. You also need to be careful about forgetting to reset the microprocessor back to the 8-bit mode before doing the quit call in ProDOS 8. A common mistake is to branch past the mode switch to an error routine that then tries to return to ProDOS 8 or Applesoft BASIC in the 16-bit mode, with disastrous results.

Finally, the Tool Locator is shut down before the final ProDOS quit call is done.

ProDOS 16 Tool Locator

In ProDOS 16, the only variation is that you need not specifically set and clear the 16-bit mode, since that is the system state when your program begins. Also, don't forget to set the data bank register equal to the program bank register when you start your program. See Program 16-2.

The only other major change is that the Merlin directive TR ON (for TRuncate ON) has been added on line 81. If you compare this listing to that for the ProDOS 8 Tool Demo program, you'll see that TR ON limits the bytes printed for data blocks like those created by the STR pseudo-op to a single line. The remaining bytes are, of course, assembled—just not listed.

Program 16-2. ProDOS 16 Tool Locator Demo

1 * PRODOS 16 TOOL LOCATOR DEMO *
2 * MERLIN ASSEMBLER *
3 ********************************************
4 5
5 6 MX  %00 ; FULL 16-BIT MODE
6 7 REL ; RELOCATABLE OUTPUT
7 8 DSK P16.TOOL.DEMO.L
8 9
9 =E100A8
10 PRODOS EQU $E100A8 ; PRODOS 16 ENTRY POINT
11

320
The Apple IIgs Toolbox

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Machine Code</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>008000: 4B</td>
<td>48 14</td>
<td>BEGIN PHK</td>
<td>GET PROGRAM BANK</td>
</tr>
<tr>
<td>008001: AB</td>
<td>AB</td>
<td>PLB</td>
<td>SET DATA BANK</td>
</tr>
<tr>
<td>008002: E2 30</td>
<td>00 00</td>
<td>SETRES SEP $30</td>
<td>8-BIT MODE</td>
</tr>
<tr>
<td>008004: A9 5C</td>
<td>E2 30</td>
<td>LDA #$5C</td>
<td>JML (JMP LONG)</td>
</tr>
<tr>
<td>008006: 8F F8</td>
<td>A9 SC</td>
<td>LDA #$SC</td>
<td>CTRL VECTOR</td>
</tr>
<tr>
<td>008008: C2 30</td>
<td>A9 SC</td>
<td>LDA #$SC</td>
<td>CTRL VECTOR</td>
</tr>
<tr>
<td>00800A: A9 6F</td>
<td>C2 30</td>
<td>LDA #$RESUME</td>
<td>16-BIT MODE</td>
</tr>
<tr>
<td>00800C: 8F F9</td>
<td>SF FS 03 00</td>
<td>STAL $3F8</td>
<td>$3F9,3FA</td>
</tr>
<tr>
<td>00800E: A9 00</td>
<td>C2 30</td>
<td>LDA #$RESUME</td>
<td>16-BIT MODE</td>
</tr>
<tr>
<td>008010: 8F FB</td>
<td>SF FB 03 00</td>
<td>STAL $3FB</td>
<td>$3FB,3FC</td>
</tr>
<tr>
<td>008012: A2 01 02</td>
<td>22 00 00 El</td>
<td>STARTUP LDX #$0201</td>
<td>TLStartUp</td>
</tr>
<tr>
<td>008014: 90 02</td>
<td>JSL $E10000</td>
<td>START UP TOOL LOCATOR</td>
<td></td>
</tr>
<tr>
<td>008016: 00 00</td>
<td>BCC BOX</td>
<td>NO ERROR</td>
<td></td>
</tr>
<tr>
<td>008018: 68 00 00</td>
<td>22 00 00 El</td>
<td>STARTUP LDX #$0201</td>
<td>TLStartUp</td>
</tr>
<tr>
<td>00801A: F4 00 00</td>
<td>SF $0000</td>
<td>PUSH SPACE FOR RESULT</td>
<td></td>
</tr>
<tr>
<td>00801C: F4 00 00</td>
<td>PEA LINE1</td>
<td>HIGH WORD OF LINE1</td>
<td></td>
</tr>
<tr>
<td>00801E: F4 80 80</td>
<td>PEA LINE1</td>
<td>LOW WORD OF LINE1</td>
<td></td>
</tr>
<tr>
<td>008020: F4 00 00</td>
<td>PEA LINE2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>008022: F4 96 80</td>
<td>PEA LINE2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>008024: F4 00 00</td>
<td>PEA BUTTON1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>008026: F4 96 80</td>
<td>PEA BUTTON1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>008028: F4 00 00</td>
<td>PEA BUTTON2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00802A: F4 96 80</td>
<td>PEA BUTTON2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00802C: A2 01 12</td>
<td>LDX #$1201</td>
<td>TLTextMountVolume</td>
<td></td>
</tr>
<tr>
<td>00802E: 90 02</td>
<td>JSL $E10000</td>
<td>DO TOOLBOX CMD</td>
<td></td>
</tr>
<tr>
<td>008030: 00 00</td>
<td>BCC TEST</td>
<td>NO ERROR</td>
<td></td>
</tr>
<tr>
<td>008032: 68 00 00</td>
<td>22 00 00 El</td>
<td>STARTUP LDX #$0201</td>
<td>TLStartUp</td>
</tr>
<tr>
<td>008034: 00 00</td>
<td>BRK $00</td>
<td>BRK IF THERE IS</td>
<td></td>
</tr>
<tr>
<td>008036: F4 00 00</td>
<td>BOX PEA $0000</td>
<td>PUSH SPACE FOR RESULT</td>
<td></td>
</tr>
<tr>
<td>008038: F4 00 00</td>
<td>PEA LINE1</td>
<td>HIGH WORD OF LINE1</td>
<td></td>
</tr>
<tr>
<td>00803A: F4 80 80</td>
<td>PEA LINE1</td>
<td>LOW WORD OF LINE1</td>
<td></td>
</tr>
<tr>
<td>00803C: F4 96 80</td>
<td>PEA LINE2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00803E: F4 00 00</td>
<td>PEA BUTTON1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>008040: F4 96 80</td>
<td>PEA BUTTON1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>008042: A2 01 12</td>
<td>LDX #$1201</td>
<td>TLTextMountVolume</td>
<td></td>
</tr>
<tr>
<td>008044: 90 02</td>
<td>JSL $E10000</td>
<td>DO TOOLBOX CMD</td>
<td></td>
</tr>
<tr>
<td>008046: 00 00</td>
<td>BCC TEST</td>
<td>NO ERROR</td>
<td></td>
</tr>
<tr>
<td>008048: 68 00 00</td>
<td>22 00 00 El</td>
<td>STARTUP LDX #$0201</td>
<td>TLStartUp</td>
</tr>
<tr>
<td>00804A: C9 01 00</td>
<td>CMP #1</td>
<td>BUTTON 1 = QUIT</td>
<td></td>
</tr>
<tr>
<td>00804C: F0 05</td>
<td>INC LINE2+20</td>
<td>INCREMENT COUNTER</td>
<td></td>
</tr>
<tr>
<td>00804E: F0 05</td>
<td>BRA BOX</td>
<td>TRY AGAIN ...</td>
<td></td>
</tr>
<tr>
<td>008050: EE AA</td>
<td>00 80</td>
<td>TLUshDown</td>
<td></td>
</tr>
<tr>
<td>008052: 80 CF</td>
<td>LDX #$0301</td>
<td>TLShutDown</td>
<td></td>
</tr>
<tr>
<td>008054: EE AA</td>
<td>00 80</td>
<td>SHUT DOWN TOOL LOCATOR</td>
<td></td>
</tr>
<tr>
<td>008056: A2 01 03</td>
<td>LDX #$0301</td>
<td>SHUT DOWN TOOL LOCATOR</td>
<td></td>
</tr>
<tr>
<td>008058: 80 90</td>
<td>JSL $E10000</td>
<td>NO ERROR</td>
<td></td>
</tr>
<tr>
<td>00805A: 00 00</td>
<td>BCC QUIT</td>
<td>ERROR</td>
<td></td>
</tr>
<tr>
<td>00805C: 22 00 00</td>
<td>JSL QUIT</td>
<td>QUIT CALL</td>
<td></td>
</tr>
<tr>
<td>00805E: 22 00 00</td>
<td>DA $29</td>
<td>QUIT CODE</td>
<td></td>
</tr>
<tr>
<td>008060: 78 80</td>
<td>ADCR PARMBL</td>
<td>ADDRESS OF PARM TABLE</td>
<td></td>
</tr>
<tr>
<td>008062: B0 11</td>
<td>BCS ERROR</td>
<td>NEVER TAKEN</td>
<td></td>
</tr>
<tr>
<td>008064: 00 00</td>
<td>BRK $00</td>
<td>SHOULD NEVER GET HERE...</td>
<td></td>
</tr>
</tbody>
</table>

321
Using Macros in a Source Listing

There are times in writing an assembly language source listing that you’ll find yourself typing the same sort of lines over and over again. You may even find yourself wishing there was some abbreviated way of telling the assembler to create an often-used block of code, perhaps even with certain customized aspects to it.

Let’s start with a simple example: Incrementing a two-byte pointer in the 8-bit mode takes several steps in assembly language:

```
INCR INC $06 ; INCREMENT LOW-ORDER BYTE
BNE DONE ; NO CARRY, SKIP NEXT STEP
INC $07 ; INCREMENT HIGH-ORDER BYTE
DONE NOP ; YOUR PROGRAM CONTINUES HERE
```

Assuming you have to use this routine often within a program, you can avoid retyping it by creating a macro definition. Macro is short for a macro-instruction, meaning a larger pseudo-instruction created out of several primary instructions. A macro can be defined in the assembler at the beginning of a source listing or in a separate file of definitions called a macro library. A macro library can be referenced once at the beginning of an assembly, and from then
on any macro defined in the library will automatically be recognized during the assembly.

A *Merlin* macro definition looks like this:

```
INCR MAC ; BEGIN MACRO DEFINITION
INC $06 ; INCREMENT LOW-ORDER BYTE
BNE DONE ; NO CARRY, SKIP NEXT STEP
INC $07 ; INCREMENT HIGH-ORDER BYTE
DONE NOP ; YOUR PROGRAM CONTINUES HERE
EOM ; END OF MACRO DEFINITION
BEGIN INCR ; NEW INSTRUCTION = MACRO 'INCR'
```

A *Merlin* macro definition begins with the assembler directive `MAC`. The label used on that line will be the name assigned to the defined macro. The lines which follow define each instruction that is to be part of the macro. To terminate a macro definition, use the directive `EOM (End Of Macro)`, or, alternatively, use the characters `<<`.

After the definition, the macro name can be used in the opcode field, just as though it were an assembly language instruction. When the file is assembled, all of the bytes associated with the macro definition will be created, just as though the lines of the macro had been typed at that spot in the listing. The macro definition itself at the beginning of the listing is not assembled as such. Program 16-3 shows the way the two-byte increment program would assemble.

Program 16-3. Two-Byte Increment Example

```
1 INCR MAC ; BEGIN MACRO DEFINITION
2 INC $06 ; INCREMENT LOW-ORDER BYTE
3 BNE DONE ; NO CARRY, SKIP NEXT STEP
4 INC $07 ; INCREMENT HIGH-ORDER BYTE
5 DONE NOP ; YOUR PROGRAM CONTINUES HERE
6 EOM ; END OF MACRO DEFINITION
7 BEGIN INCR ; NEW INSTRUCTION = MACRO 'INCR'
8
---End Merlin-16 assembly, 7 bytes, Errors: 0
```

Notice that no bytes of object code are generated during the macro definition itself. Object code is generated only when the macro is actually used in the program. In the listing, you can see line 8 repeated as each line of the
Chapter 16

Macro is played out. Each time INCR is used, locations $06$ and $07$ will be incremented.

Now, suppose that later in the listing you want to increment locations $08$ and $09$. Do you have to create a new macro definition? No, because you can rewrite the original definition with variables, like this:

```
INCR MAC ; BEGIN MACRO DEFINITION
INC ]1 ; INCREMENT LOW-ORDER BYTE
BNE DONE ; NO CARRY, SKIP NEXT STEP
INC ]1+1 ; INCREMENT HIGH-ORDER BYTE
DONE NOP ; YOUR PROGRAM CONTINUES HERE
EOM ; END OF MACRO DEFINITION

BEGIN INCR $06 ; INCREMENT LOCATIONS $06,07
BEGIN MACRO DEFINITION
INCREMENT WW-ORDER BYTE
NO CARRY, SKIP NEXT STEP
INCREMENT HIGH-ORDER BYTE
YOUR PROGRAM CONTINUES HERE
END OF MACRO DEFINITION
BEGIN INCR $08 ; INCREMENT LOCATIONS $08,09
```

The characters ]1 represent a variable that will be filled in with whatever value or label is in the operand field following the macro call. For example, Program 16-4 defines the macro INCR and the memory locations LOC1 as $06$ and LOC2 as $08$.

Program 16-4. Macro Example

```
="0006
=0008

1 =0006 2 EQU $06 ; $06,07
2 =0008 3 EQU $08 ; $08,09

BEGIN
INCR MAC ; BEGIN MACRO DEFINITION
BEGIN MACRO DEFINITION
INCREMENT WW-ORDER BYTE
NO CARRY, SKIP NEXT STEP
INCREMENT HIGH-ORDER BYTE
YOUR PROGRAM CONTINUES HERE
END OF MACRO DEFINITION
BEGIN INCR LOC1 ; BEGIN MACRO DEFINITION
BEGIN MACRO DEFINITION
INCREMENT LOC1
BEGIN
INC LOC1
INC LOC1
BNE DONE
INC LOC1+1
DONE NOP
EOM
END OF MACRO DEFINITION
BEGIN
INCR LOC2
BEGIN MACRO DEFINITION
INCREMENT LOC2
BEGIN
INC LOC2
INC LOC2
BNE DONE
INC LOC2+1
DONE NOP
EOM
END OF MACRO DEFINITION
```

--End Merlin-16 assembly, 14 bytes, Errors: 0

324
You can include up to 9 variables in a macro definition using the variables J1 through J9, separating each variable in the operand field with a comma. For example, here's a macro definition with three variables:

```assembly
SWAP MAC
LDA J1 ; GET 1ST BYTE
STA J3 ; SAVE IN TEMP LOCATION
LDA J2 ; GET 2ND VALUE
STA J1 ; PUT IN 1ST LOCATION
LDA J3 ; GET ORIG. 1ST VALUE
STA J2 ; MOVE TO POSITION 2
EOM ; END OF DEFINITION
```

This could then be used within the program like this:

```assembly
PROGRAM SWAP $06,$08,$0A
```

or like this:

```assembly
SWAP MEM1,MEM2,MEM3
```

Macro definitions are used extensively in Apple IIgs programming because the same programming procedures are used over and over again. For example, you've seen how every tool call begins with pushing data on the stack, followed by loading the X register with the command value and doing the JSL to $E10000. Apple has established a standard set of tool macros using predefined names for each tool. For example TLStartUp is defined as:

```assembly
TLStartUp LDX #$0201
JSL $E10000
```

In a program, you'd just see:

```assembly
BEGIN TLStartUp
```

The APW assembler precedes all the tool macro names with an underscore (as in _TLStartUp), but otherwise the names are the same.

In addition to not having to type as much, macros have the advantage of conserving space in the listing and making it easier to see the big picture. It's also easier to remember the name TLStartUp for a function than the code number #$0201.

Program listings that use macros like this assume that there is a predefined macro library on the disk with the source file, and that all the reference macros have been defined.

The disadvantage is that if a listing only shows the macro name, and not the expanded lines generated, you don't see the actual lines of code. And, if you don't have the macro definition used in a program on your assembler disk,
you won’t be able to assemble the program without writing the macro code yourself.

Because the source listings for Apple II GS programs that use the tools can get very large, it will be essential in the next few chapters that we make some use of macros to minimize the length of the presented listings. In addition, this will introduce you to the idea of using macros in your own programs on a regular basis. In the interest of keeping the listings short, and as clear as possible, only a half-dozen macro definitions will be used here.

Building a Macro Library

You can create this macro library yourself with either Merlin 16 or APW. We’ll define a standard tool call macro that includes our JSL $E10000 plus the test for an error and a BRK. Normally, you wouldn’t include a BRK as an error handler in a program anyone else was going to use, but it’s handy here to help debug your programs. Just be sure to include the Control-Y resume routine so you can get back to a program launcher if the program does break.

The first macro, ToolCall, will be defined as follows:

```
ToolCall MAC
DEFINE THE MACRO ToolCall
LDX #)1
GET THE CALL NUMBER
JSL $E10000
DO THE CALL
BCC CONT
CONTINUE WITH PROGRAM
BRK $00
BREAK ON ERROR
CONT
EOM
END OF MACRO DEFINITION
```

This macro, ToolCall, will simply load the X register with the value for the call number, and then JSL to $E10000. The BCC CONT instruction will then test for an error and cause the program to continue with whatever instruction follows the BRK if no error is detected. Remember that, in Merlin, an instruction isn’t required after a label, so this will achieve the desired result of making CONT equivalent to the address of the next instruction in the program the macro is used in. You’ll see the call numbers as part of each macro, and the official name of each call will be included in the comment field, like this:

```
ToolCall $0201 ; TLStartUp
```

Because many of the Apple II GS tools require information to be pushed on the stack prior to a call, it will also be useful to define some macros for this. For example, we can create a macro called PushWord that will push a two-byte word onto the stack. It would look like this in a program:

```
PushWord LABEL
```

326
However, now we have a new challenge. How will you distinguish between when you want to push the contents of the location LABEL on the stack, and when you want to push the address value of LABEL itself?

The answer is to use a conditional statement in a macro. A conditional statement is just like an IF-THEN test in BASIC. It will let you test certain aspects of the inputs to the macro and generate different expansions accordingly. For example, here's a macro definition for PushWord:

```
PushWord MAC ; DEFINE MACRO PushWord
IF #,),1  ; IS 1ST CHAR OF ]1 A "#"
PEA #)1 ; YES, PUSH VALUE ON STACK
ELSE ; OTHERWISE,
LDA ]1 ; LOAD CONTENTS
PHA ; AND STORE THAT
FIN ; END OF CONDITIONAL PORTION
EOM ; END OF MACRO DEFINITION
```

The macro definition directive IF tells the assembler to look at the first character of the input variable ]1. If it's the pound sign (#), signifying an immediate value, the value itself is pushed on the stack using the PEA instruction. The ELSE directive begins the block of instructions of what to do if the test fails, which in this case is to then load the Accumulator with the contents of ]1, and to push that onto the stack.

FIN defines the end of the conditional part of the macro and tells the assembler to assemble everything from that point on (or until the next IF). This is where any code to be found in both possible macro expansions would be entered (or it's before the IF at the beginning).

With this macro, you can now have two uses of PushWord:

```
PushWord #LABEL ; PUSH VALUE OF THE LABEL ON STACK
PushWord LABEL  ; PUSH CONTENTS OF LABEL ON THE STACK
```

The same technique can be used to create a PushLong macro to push four bytes, a long address, for example, on the stack:

```
PushLong MAC ; DEFINE MACRO PushLong
IF #,),1 ; IMMEDIATE VALUE?
PEA ]1 ; YES, PUSH HIGH WORD
PEA ]1 ; PUSH LOW WORD
ELSE ; OTHERWISE,
LDA ]1 +2 ; GET HIGH WORD CONTENTS
PHA ; PUSH ON STACK
LDA ]1 ; GET LOW WORD CONTENTS
PHA ; PUSH ON STACK
FIN ; END OF CONDITIONAL PORTION
EOM ; END OF MACRO DEFINITION
```
In a similar manner, it would be nice to be able to pull data off the stack. In this case, the data pulled off the stack will be stored in a specified memory location. Here's two macros, *PullWord* and *PullLong* that do just that:

```
MAC
DEFINE MACRO PullWord
PLA ; GET TWO BYTES OFF STACK
STA J1 ; STORE IN MEMORY
EOM ; END OF MACRO DEFINITION

MAC
DEFINE MACRO PullLong
PLA ; GET LOW WORD OFF STACK
STA J1 ; STORE IT
PLA ; GET HIGH WORD OFF STACK
STA J1+2 ; STORE IT
EOM ; END OF MACRO DEFINITION
```

Not every macro will be used in every program, but having these definitions in one library that we'll call UTIL.MACS will make it easy to use any of them in a given program.

Program 16-5 is an example of what a complete program using the Merlin Apple II GS disk-based macro library might look like using these macros.

When you assemble Program 16-5, be sure to verify that the checksum on line 82 agrees with your own assembly. This checksum is the same for both the macro and nonmacro versions of this program.

This listing appears much longer because the contents of the macro file itself are printed at the beginning, and because each use of a macro is expanded within the listing. By using the Merlin directives LST OFF and LST ON on either side of the USE UTIL.MACS instruction, the listing of the macro file itself can be suppressed. There is also another Merlin directive, EXP OFF (EXPand macros—OFF) that will keep each macro from being expanded during the assembly.

In the interest of conserving space for the listings in the following chapters, most programs will use the UTIL.MACS macro library, and will not be expanded in the listings. If you should forget what a particular macro does in looking at a listing, you can always refer back to this chapter. Of course, in your own assemblies, you may also just omit the LST OFF or EXP OFF directives when you want a complete listing of all the assembled bytes in an object file.
Program 16-5. ProDOS 16 Tool Locator Demo with Macros

**********************************************************************
* PRODOS 16 TOOL LOCATOR DEMO *
* --USING MACROS-- *
* MERLIN ASSEMBLER *
**********************************************************************

MX %00 ; FULL 16-BIT MODE
REL ; RELOCATABLE OUTPUT
DSK P16.TOOL.DEMO.L

=E100A8

PRODOS EQU $E100A8 ; PRODOS 16 ENTRY POINT

LST OFF ; DON'T LIST MACROS
USE UTIL.MACS ; USE MACRO LIBRARY
LST ON ; LISTING BACK "ON"
EXP OFF ; DON'T EXPAND MACROS

008000: 4B
008001: AB

BEGIN

PHK
; GET PROGRAM BANK
PLB
; SET DATA BANK

SETRES
SEP $30 ; 8-BIT MODE
LDA #$5C ; JML (JMP LONG)
STAL $3F8 ; CTRLY VECTOR
REP $30 ; 16-BIT MODE

LDA #$RESUME
STAL $3F9 ; $3F9,3FA

LDA #$RESUME
STAL $3FB ; $3FB,3FC

STARTUP ToolCall $0201 ; TLStartUp

TEST

PLA
; RETRIEVE CHOICE VALUE
CMP #1
; BUTTON 1 = QUIT
BEQ SHUTDOWN

INC LINE2+20 ; INCREMENT COUNTER
BRA BOX ; TRY AGAIN...
Chapter 16

51 SHUTDOWN ToolCall $0301 ; TLShutDown
52
008061: 22 A8 00 E1 53 QUIT JSL PRODOS ; DO QUIT CALL
008065: 29 00 54 DA $29 ; QUIT CODE
008067: 78 80 00 00 55 ADRL PARMBL ; ADDRESS OF PARM TABLE
00806B: B0 11 =807E 56 BCS ERROR ; NEVER TAKEN
00806D: 00 00 57 BRK $00 ; SHOULD NEVER GET HERE...

58
59 **********************************************
60
00806F : 48 61 RESUME PHK
008070: AB 62 PLB ; SET OUR DATA BANK
008071: 18 CLC
008072: FB XCE ; SET NATIVE MODE
008073 : C2 30 REP $30 ; 16-BIT MODE
008075: 4C 56 80 66 JMP SHUTDOWN ; TRY TO SHUTDOWN

67
68 **********************************************
69
008078: 00 00 00 00 70 PARMBL ADRL $0000 ; PTR TO PATHNAME
00807C: 00 00 71 FLAG DA $00 ; ABSOLUTE QUIT
72

00807E: 00 00 73 ERROR BRK $00 ; WE'LL NEVER GET HERE?
74
75 TR ON ; TRUNCATE BYTES PRINTED.
76

008080: 15 D0 D2 CF 77 LINE1 STR "PRODOS 16 TEST SYSTEM"
008096: 15 D0 F2 E5 78 LINE2 STR "Press a key. (Try #1)"
79

0080AC: 0E D2 E5 F4 80 BUTTON1 STR "Return to Quit"
0080BB: 10 C5 F3 E3 81 BUTTON2 STR "Esc to Try Again"
82

0080CC: 63 83 CHK ; CHECKSUM FOR LISTING

--End Merlin-16 assembly, 205 bytes, errors: 0

APW Macros

Defining macros with the APW assembler is similar in concept, although different in actual procedure.

In APW, the macro library must be defined as a separate file—it cannot be included as part of the main source file itself. A macro definition in such a file begins with the directive MACRO. This is followed on the next line by the name of the macro. In the example below, this is INCR (INCREMENT). The macro definition ends with the directive MEND (Macro END).
The Apple IIgs Toolbox

MACRO
INCR
INC $06
BNE DONE
INC $07
DONE ANOP
MEND

The macro is then used in a source program with the directive MCOPY (for Macro COPY). This places a copy of the macro library specified in the list of available macro libraries. A source program that uses a macro library would look like this:

```
MCOPY MYMACROS
GEN ON
MAIN START
PROGRAM INCR
END
```

The macro is invoked by using the macro name in the opcode field as though it were an assembly language instruction. The GEN ON directive is used if you want each macro instruction generated listed. Program 16-6 is the output of an assembly using the macro file and the source file above:

```
Program 16-6. Sample APW Program Using Macros

0001 0000
0002 0000
0003 0000
0004 0009
0005 0000
0006 0000
0007 0000
 0000 E6 06 + INC $06 ; INCREMENT $06
 0002 D0 02 + BNE DONE ; NO WRAPAROUND
 0004 E6 07 + INC $07
 0006 + DONE ANOP
0008 0006
0009 0006
```

9 source lines
1 macros expanded
4 lines generated
Chapter 16

It’s also possible to include variables in APW macros as well. A macro definition that uses a variable label looks like this:

```
MACRO ; DEFINE ‘INCR’
INCR &LOC
INC &LOC ; INCREMENT $06
BNE DONE ; NO WRAPAROUND
INC &LOC+1
DONE ANOP
MEND ; END OF DEFINITION
```

The requirements are that the use of a variable label be called out on the first line of the macro definition by following the name of the macro by the expected variable list to be used when the macro is called.

The program calling the macro then follows its use of the macro name by the values or labels it wishes substituted in the macro. To increment location $06 and $07, for example, the listing would look like this:

```
MCOPY MYMACROS2
GEN ON
MAIN START
PROGRAM INCR $06
END
```

Which would generate the assembly in Program 16-7.

Program 16-7. Increment Location

```
0001 0000
0002 0000
0003 0000
0004 0000
0005 0000
0006 0000
0007 0000
PROGRAM INCR $06
0000 E6 06 + INC $06 ; INCREMENT $06
0002 D0 02 + BNE DONE ; NO WRAPAROUND
0004 E6 07 + INC $06+1
0006 +DONE ANOP
0008 0006
0009 0006
END
```

9 source lines
1 macros expanded
4 lines generated
If a macro used several labels, the definition would look like this:

```
MACRO      ; DEFINE 'SWAP'
SWAP       &LOC1,&LOC2,&LOC3
LDA &LOC1  ; GET 1ST VALUE
STA &LOC3  ; STORE IN TEMP LOC
LDA &LOC2  ; GET 2ND VALUE
STA &LOC1  ; PUT IN 1ST POSITION
LDA &LOC3  ; GET ORIG. 1ST VALUE
STA &LOC2  ; PUT IN 2ND POSITION
DONE  ANOP
MEND       ; END OF DEFINITION
```

This would be called in a program like this:

```
MCOPY MYMACROS3
GEN ON
MAIN START
PROGRAM SWAP $06,$07,$08
END
```

Which would assemble as follows:

```
0001 0000
0002 0000 0005 0000 0006 0000 0007 0000
0003 0000 0004 0000 0007 0000
0000 A5 06 + 0002 85 08 + 0004 A5 07 + 0006 85 06 + 0008 A5 08 + 000A 85 07 + 000C +
0002 0000 0004 0000 0006 0000 0008 000C 0009 000C
0000 0005 0006 0007 0008 0009
0001 0000 0002 0000 0003 0000 0004 0000 0005 0000 0006 0000 0007 0000 0008 000C 0009 000C
0001 0000 0002 0000 0003 0000 0004 0000 0005 0000 0006 0000 0007 0000 0008 000C 0009 000C
0001 0000 0002 0000 0003 0000 0004 0000 0005 0000 0006 0000 0007 0000 0008 000C 0009 000C
0001 0000 0002 0000 0003 0000 0004 0000 0005 0000 0006 0000 0007 0000 0008 000C 0009 000C
0001 0000 0002 0000 0003 0000 0004 0000 0005 0000 0006 0000 0007 0000 0008 000C 0009 000C
0001 0000 0002 0000 0003 0000 0004 0000 0005 0000 0006 0000 0007 0000 0008 000C 0009 000C
0001 0000 0002 0000 0003 0000 0004 0000 0005 0000 0006 0000 0007 0000 0008 000C 0009 000C
0001 0000 0002 0000 0003 0000 0004 0000 0005 0000 0006 0000 0007 0000 0008 000C 0009 000C
0001 0000 0002 0000 0003 0000 0004 0000 0005 0000 0006 0000 0007 0000 0008 000C 0009 000C
0001 0000 0002 0000 0003 0000 0004 0000 0005 0000 0006 0000 0007 0000 0008 000C 0009 000C
0001 0000 0002 0000 0003 0000 0004 0000 0005 0000 0006 0000 0007 0000 0008 000C 0009 000C
0001 0000 0002 0000 0003 0000 0004 0000 0005 0000 0006 0000 0007 0000 0008 000C 0009 000C
```

9 source lines
1 macros expanded
8 lines generated
Chapter 16

The APW disk also contains an Apple IIGS resource macro library with macro definitions for all of the Apple IIGS tools. As with Merlin 16, these macros can be used to simplify a listing that uses the Apple IIGS tools.

In addition, the macro library M16.UTILITY contains the Push and Pull macros just discussed for Merlin. You will have to add the definition for ToolCall if you wish to use it as part of M16.UTILITY.
Chapter 17

The Memory Manager and Miscellaneous Tools
Chapter 17

The Memory Manager and Miscellaneous Tools

The multiple program operating environment of the Apple IIGS would not be possible without some supervising system to manage memory for the various applications, desk accessories, and the operating system itself. The Memory Manager tool set is a collection of routines for allocating, moving, and de-allocating blocks of memory that are used by different applications and the operating system. Even in the 64K Applesoft BASIC environment, you've seen how Applesoft BASIC, the Monitor, ProDOS and your own program all compete for the same memory in bank 0.

In the Apple IIGS, there are even more program entities such as desk accessories and the tool sets themselves—all in memory at the same time. For proper integration of all of this, it is essential that there be a central manager of all memory, to which each individual program goes to be assigned any memory it needs. This is the job of the Memory Manager.

The Memory Manager

As with most things, the Memory Manager is actually quite simple. When ProDOS 16 first loads a program, it asks the Memory Manager to find sufficient memory for the application from within any memory that is not already in use.

On boot up, most memory is available, and when the application is loaded, the Memory Manager marks the area occupied by the program as in use. The Memory Manager then creates two data structures to keep track of that memory. The first is an identification number, called the User ID. This ID number is associated with a particular application, and can then be associated with any additional blocks of memory allocated to that program. Then, when the program quits, all associated—but perhaps physically separate—blocks of memory for that application can be de-allocated and returned to the pool of available memory. The second data structure is called a handle, and is a pointer to the memory block while it's in use.

The ID number of a block of memory is made up of several components,
or fields. These are called the Main, Aux, and Type fields, and are assigned to specific bit positions in the ID number (a two-byte value):

```
<table>
<thead>
<tr>
<th>Type Field</th>
<th>Aux ID Field</th>
<th>Main ID Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0-$F</td>
<td>$0-$F</td>
<td>$00-$FF</td>
</tr>
</tbody>
</table>
```

The Main ID field is assigned by the ID Manager, and ranges from $00 to $FF for each particular Type. The value $00 is reserved for the system, and so Main ID values in this field start with $01.

The Type field is an indicator of whom the ID value belongs to. The possible values, in the range of $0 to $F have been assigned as shown in Table 17-1.

Table 17-1. Type Field Indicator

0  Memory Manager use (for example, in allocating RAM disk space)
1  Application
2  Control Program
3  ProDOS
4  Tool Sets
5  Desk Accessories
6  Runtime Libraries
7  System Loader
8  Firmware
9  Tool Locator
A  Setup File
B  Undefined
C  Undefined
D  Undefined
E  Undefined
F  Undefined

For example, the first ProDOS 16 application to own memory in the system might have an ID of $1001 (application, first ID).

When ProDOS 8 is run from ProDOS 16, much of the memory in banks 0 and 1 will have already been assigned, and the ProDOS 8 program will be considered part of that memory allocation. A ProDOS 8 System file (or even an Applesoft BASIC routine requesting memory), would probably get the ID value $3001 (ProDOS, first ID).

Note that by the time your application gets control when it starts up, in either ProDOS 8 or ProDOS 16, this ID value has already been assigned to the memory your program occupies. For ProDOS 16, this will be just the memory occupied by your program, plus an additional $400 bytes given as a default direct page and stack to every ProDOS 16 program, unless otherwise directed
during the assembly of the program.

For ProDOS 8, this memory is just generally assigned to the entire range of $800 to $BFFF in banks 0 and 1, and the shadowed graphics display areas in banks $E0 and $E1: $E0/2000 to $E0/5FFF for hi-res, and $E1/2000 to $E1/9FFF for double and super hi-res displays. (Yes, as a matter of fact, they did provide the option of an Applesoft BASIC or ProDOS 8 system file accessing the bank 1 super hi-res area and then shadowing this to bank $E1 to make the display visible. Try a POKE 49193,161 in Applesoft BASIC to enable the super hi-res display. You’ll have to carefully type POKE 49193,33 to restore things, since you won’t be able to see what you’re typing.)

The $00 to $7FF and $C000 to $FFFF parts of banks 0 and 1 are marked as permanently “in use” by the memory manager, regardless of whether you’re using ProDOS 8 or ProDOS 16, and need no special attention from your application.

**Loading ProDOS 8 Directly**

It is possible, but not advisable, to boot directly into ProDOS 8 on the Apple IIGS, as was described in Chapter 13. There are two big drawbacks to this approach. The first, and most important, is that none of the RAM patches to the built-in tool sets are loaded and activated, since the Tool.Setup file is not used in a simple ProDOS 8 boot. This means that virtually every ROM-based tool in the system is operating with known bugs, and is likely to perform less than perfectly. The whole purpose of the Tool.Setup file under ProDOS 16 is to make sure that every running Apple IIGS machine is functionally equivalent, regardless of its manufacture date, when started up with the current tool sets and setup files. If you’re not using any of the Apple IIGS tools, you can boot directly into ProDOS 8. However, if you intend to use any of the routines in these chapters discussing the tools, you should always boot into ProDOS 16 first, even if you intend to ultimately end up in ProDOS 8.

The second disadvantage of booting directly into ProDOS 8 is that the Memory Manager only marks minimal areas of memory, including the ramdisk allocation, $00/0000 to $00/7FF, $00/C000 to $00/FFFF, and $01/C000 to $01/FFFF as in use (along with a very few others); thus, you must allocate your own memory in banks 0 and 1 if you want to call the Memory Manager. As a matter of interest, in such cases you can first call the Miscellaneous Tools *GetNewID* to get an ID for your itself, then use *NewHandle* ($0902) to allocate the memory your program occupies at that moment, and then finally call *MMStartUp* ($0202) to get things going (*MMStartUp* will give you your same User ID back). This is really not the recommended approach, though, and it does nothing to alleviate the tool bug problem.

The real moral is, if you’re going to use the tools, boot ProDOS 16 first.
Chapter 17

Auxiliary IDs

While an application is running, it may request additional memory blocks. These are usually associated with the Main ID of the application itself, and the application may then assign auxiliary IDs to these blocks. This is the purpose of the Aux ID field in the ID value. This field is controlled entirely by the application and is used to further classify blocks of memory.

For example, suppose an application loads two documents, each with a text and a graphics portion, for a total of four distinct blocks of memory. Let’s suppose the application’s Main ID, issued to it by the System Loader when it was started up, is $1001. The application itself may then generate two new sub-IDs as follows:

```
LDA ID ; $1001
ORA #$0100 ; $1101
STA SID1 ; SAVE SUB-ID #1
LDA ID ; $1001
ORA #$0200 ; $1201
STA SID2 ; SAVE SUB-ID #2
```

The application can then assign sub-ID #1 to the two blocks of memory for the first document, and the second sub-ID to the memory used by the second document.

The advantage is that the application may then later selectively de-allocate all the memory (which for some applications may be dozens of memory blocks per document) in one tool call, using the sub-ID (for example, SID2 = $1201) for the particular document. The alternative would be to create a loop to run through each data block and de-allocate it individually. The sub-ID method provides a much more efficient approach.

These sub-IDs should be used for any additional memory your program requires for its own use, such as additional direct-page areas or data blocks.

It’s also possible, although rare, to spinoff completely separate programs from the one currently loaded. For example, suppose you had an application that wanted to install a classic desk accessory. You would want the accessory’s memory to remain marked as in use even after your start up application had departed the scene.

New IDs may be obtained from within an application using the Miscellaneous tool set, specifically with the call GetNewID (call number = $2003). This new ID is a completely new entity and would only be required if you wanted to allocate memory for some routine or data structure that was to remain in memory after your program quit and had its own memory de-allocated by the System Loader. In the normal course of events, you should never have to call GetNewID. The Miscellaneous tool set, as its name implies, is a collection of
miscellaneous tool commands that can be useful during program execution. These include not only the GetNewID routine, but also routines to read the mouse, determine the system time and others. The Miscellaneous tool set is described in greater detail below.

**Handles**

If a given ID number is not unique for a given memory block, that is, the same ID number may be assigned to several blocks of memory, how does the Memory Manager identify a single block? It identifies it with the second data structure created when memory is allocated, called a *handle*. A handle contains a pointer to the location of the block of memory, its size (length), and the ID number of the application to which that block of memory is assigned (the User ID), and some additional information. To reference a block of memory, most Apple IIIGS tool calls use the address of the handle to a memory block, not the address of the block itself.

This is because the Memory Manager, from time to time, moves blocks of memory, providing they are designated as movable, from their current location to a new one. This is usually done to try to make room for another memory request, either from an application or the System Loader trying to load a new application, but it could be related to any system function that requires more memory, including desk accessories.

For this reason, the Memory Manager and most other Apple IIIGS tools deal with handles as the indicator and descriptor of a particular block of memory. You can also think of a handle as a pointer to a pointer. The handles themselves are allocated in bank $E1, and are guaranteed not to move. Thus, whenever you want to access your block of memory, you give the handle address, and the system then looks there to determine where your block of memory is being kept at the moment.

Not all memory in the Apple IIIGS is managed in the same way by the Memory Manager. The Memory in the Apple IIIGS falls into three categories:

**Normal Memory.** Memory managed by the Memory Manager, this includes banks $02 through $DF and a little of banks $E0 and $E1, specifically $E0/6000 to $E0/BFFF and $E1/A000 to $E1/BFFF.

**Special Memory.** This is memory managed by the Memory Manager, but it has restrictions on it because it is used by programs designed for the Apple IIe and IIc. By keeping this memory restricted, the memory is more likely to be usable if the user wants to start up a ProDOS 8 or other 64K or 128K Apple II–type application. Special memory includes $00/800 to $00/BFFF, $01/800 to $01/BFFF, $E0/2000 to $E0/5FFF (the shadowed hi-res pages), and $E1/2000 to $E1/9FFF (the shadowed double and super hi-res display pages).
Chapter 17

**Unmanaged Memory.** The remaining memory, the area from $00 to $800 and $C000 to $FFFF (including the 4K expansion RAM area) in banks $00, $01, $E0 and $E1, and the $800 to $1FFF areas in banks $E0 and $E1, are called *reserved memory* and are not controlled by the Memory Manager. This area is always marked as in use by the Memory Manager.

**Requesting Memory**

Ordinarily when requesting memory, the Memory Manager will allocate a block in nonspecial, or normal, memory. If, however, you have a specific reason for using special memory, for example using the super hi-res page directly in bank $E1, then you can include that as part of your request to the Memory Manager.

When requesting memory, there are a number of variable *attributes* that you can require of the memory block. These are controlled by a 2-byte attribute word, where certain bits are used to flag a given attribute. These are shown in Table 17-2.

Table 17-2. Variable Attributes

<table>
<thead>
<tr>
<th>Bit</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Fixed Bank</td>
<td>1 = Block must be in a particular bank.</td>
</tr>
<tr>
<td>1</td>
<td>Fixed Address</td>
<td>1 = Block must start at a specific address.</td>
</tr>
<tr>
<td>2</td>
<td>Page Aligned</td>
<td>1 = Block must start at a page boundary ($100, $200, $300, and so on).</td>
</tr>
<tr>
<td>3</td>
<td>Special Memory</td>
<td>1 = Block may NOT use special memory areas</td>
</tr>
<tr>
<td>4</td>
<td>Bank Boundary</td>
<td>1 = Block cannot extend across a bank boundary (for example, from $02/F000 to $03/0100).</td>
</tr>
<tr>
<td>5-7</td>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>8, 9</td>
<td>Purge Level</td>
<td>(0-3) = Priority level for de-allocation (purging).</td>
</tr>
<tr>
<td>10-13</td>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Fixed Address</td>
<td>1 = Block cannot be moved in memory.</td>
</tr>
<tr>
<td>15</td>
<td>Locked</td>
<td>1 = Block is temporarily unmovable and unpurgeable.</td>
</tr>
</tbody>
</table>

When the System Loader loads your program, its memory is allocated as not page aligned; it cannot cross bank boundaries (or your LDA/STAs wouldn’t work); it cannot be in special memory, and it must be unmovable, locked, and with a purge level of 0 (cannot be de-allocated). A typical attribute byte for a ProDOS 16 application would be $C018 (%1100 0000 0001 1000). This is assigned automatically, more or less, but it is possible to change these attributes during the assembly and linking of an application. Ordinarily, you will not have to be concerned with controlling the attribute bytes for your application.

You’ll have to specify the attribute byte for any memory your application requests from the Memory Manager. For most instances, an attribute byte of
The Memory Manager and Miscellaneous Tools

$0014 (bank boundary limited, page aligned) or $0000 (no special requirements) is adequate.

When a ProDOS 16 application is started up by the System Loader, it is given two blocks of memory. The first block is for the program itself, the second is a $400-byte block used for that application’s private stack ($300 bytes) and direct-page areas ($100 bytes). However, many of the Apple II GS tools, QuickDraw, for example, require that you obtain more bank 0 direct-page space for their operation. This is obtained through the memory manager, and the address of that block (not its handle) is passed to the tool when starting it up.

In those cases, you will need a fixed and locked block specifically in bank 0. The attribute byte for this type of memory is $C001. You need not memorize these values, as they will be included in the examples that follow, and you can refer to this chapter or the sample listings in the future as you need them.

### Memory Allocation and Movement

When the Memory Manager allocates a block of memory, it first tries to obtain the block in available memory. If it can’t locate enough memory, it first compacts memory. Compacting is done by trying to move all the in-use blocks into one area, to free larger sections of available memory. Movable memory is moved to the top of memory, making available areas of free memory at the bottom. Any block that is immovable remains where it stands.

The only problem is that some blocks, such as applications themselves, have already been designated as immovable, and they may reside in the middle of memory forming barriers to relocation of other blocks. In this case, the Memory Manager tries to relocate movable blocks as high in memory as possible without going past a fixed block that it may encounter. The Memory Manager will never relocate a movable block past a fixed block.

After compacting, if enough memory to satisfy the request is still not available, the Memory Manager will try to purge any blocks marked as purgeable. A priority level of 0 to 3 has been established.

3 Most purgeable (assigned by System Loader)
2 Next-most purgeable
1 Least purgeable
0 Not purgeable

When purging, the Memory Manager looks for level 3 blocks to purge first. If this is not sufficient, it starts purging blocks at level 2, and so on. Blocks at level 0 are most likely the application itself, or other important data blocks, and so are not purged. If sufficient memory cannot be found, the Memory Manager will return an insufficient memory error.
Chapter 17

The reason for purge levels is to allow the application to set aside memory on a priority basis. For example, suppose your program has a help list that it can load from disk. Each time the user asks for help, the program has to reload the file from disk. You could get some memory from the Memory Manager and store your help list in memory, but what happens when the user’s document becomes very large? It would be a shame to limit the usefulness of the program because of memory used for the help list.

The answer is to load the help list when memory is available, but to mark it as purgeable. Then, if the application’s document starts growing large, the Memory Manager will automatically dispose of the help list to make room for the document. Obviously, in such schemes you’ve got to check to see if your data block is still in the computer whenever you want to print the help list. This is fairly simple though. Whenever the Memory Manager purges a data block, it sets the pointer within the handle to zero, but it doesn’t delete the handle itself. Thus, the handle can be checked prior to use to make sure the data is still there.

Normally, this is done by dereferencing the handle. Dereferencing is the process of looking into the handle to see what actual memory address it points to. The actual technique of dereferencing will be described in a later demonstration program.

It is also possible to check for a nil (equal to zero) handle by calling RestoreHandle ($0B02) each time you want to access a possibly purged handle. RestoreHandle tries to re-allocate a purged block to its original size (although the data contents have presumably already been lost). For our purposes, it returns an error if the data block still contains information (the block was not really purged before), or if there is insufficient memory to re-allocate the block. You can use this call to check to see if a handle has already been purged. If the error NotEmptyErr ($0203) is returned, you know the data is still there. On the other hand, if you get a MemoryErr ($0201) or no error at all, then you know your data is long gone.

To summarize the data structures then, each block of memory has a handle and a UserID associated with it. The UserID is specific to an application, and is obtained from MMStartUp when the application starts. The application may use the Aux ID field to create sub-IDs for any additional memory it requires. Purging memory releases the memory, but does not remove the handle with the UserID embedded in it from the Memory Manager’s list. Disposing of a handle both de-allocates the memory block and removes the handle with the UserID from the handle list of the Memory Manager. The UserID is at that point still active in the ID list (a separate function from memory handles).

When a program quits, it should first call DisposeAll to de-allocate any additional memory it has obtained, and then call MMShutDown to tell the
memory manager it is finished with its memory related operations. During the ProDOS quit, the system will then de-allocate the memory for the application itself and remove the application’s UserID (and any sub-IDs) from the ID list.

*DeleteID*, a Miscellaneous tool set call, is only required if the application has established other IDs (not sub-IDs) using *GetNewID*, which are separate from the main ID given the application when it first started up.

**The Memory Manager Tools**

The Memory Manager tool set contains a wide variety of routines for allocating, moving, and de-allocating memory, as well as routines for setting the purge levels and other attributes. Table 17-3 lists some of the routines (tool commands) available in the Memory Manager. Other commands may be added by Apple Computer at any time.

Table 17-3. Memory Manager Tools

<table>
<thead>
<tr>
<th>Command Value</th>
<th>Command Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0102</td>
<td>MMBootInit</td>
<td>Initialized by system on boot.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not used by application.</td>
</tr>
<tr>
<td>$0102</td>
<td>MMStartUp</td>
<td>Starts up Memory Manager.</td>
</tr>
<tr>
<td>$0302</td>
<td>MMShutDown</td>
<td>Tells Memory Manager you’re finished.</td>
</tr>
<tr>
<td>$0402</td>
<td>MMVersion</td>
<td>Returns version number of the Memory Manager.</td>
</tr>
<tr>
<td>$0502</td>
<td>MMReset</td>
<td>Reinitializes MM on reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Should not be called by an application.</td>
</tr>
<tr>
<td>$0602</td>
<td>MMStatus</td>
<td>Indicates whether MM is active.</td>
</tr>
<tr>
<td>$0902</td>
<td>NewHandle</td>
<td>Creates a new block of memory and returns handle.</td>
</tr>
<tr>
<td>$0A02</td>
<td>ReAllocHandle</td>
<td>Re-allocates a block that was purged.</td>
</tr>
<tr>
<td>$0B02</td>
<td>RestoreHandle</td>
<td>Re-allocates a handle that was purged.</td>
</tr>
<tr>
<td>$1002</td>
<td>DisposeHandle</td>
<td>Disposes of a specified memory block and its handle.</td>
</tr>
<tr>
<td>$1102</td>
<td>DisposeAll</td>
<td>Disposes of all memory and handles belonging to a specified UserID. The ID, however, remains active.</td>
</tr>
<tr>
<td>$1202</td>
<td>PurgeHandle</td>
<td>Purges a specified block.</td>
</tr>
<tr>
<td>$1302</td>
<td>PurgeAll</td>
<td>Purges all blocks for a specified ID.</td>
</tr>
</tbody>
</table>
Chapter 17

<table>
<thead>
<tr>
<th>Address</th>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1802</td>
<td>GetHandleSize</td>
<td>Returns the size of a specified block.</td>
</tr>
<tr>
<td>$1902</td>
<td>SetHandleSize</td>
<td>Changes the size of a specified block.</td>
</tr>
<tr>
<td>$1A02</td>
<td>FindHandle</td>
<td>Returns the handle of the block containing the specified memory address.</td>
</tr>
<tr>
<td>$1B02</td>
<td>FreeMem</td>
<td>Returns the total number of free bytes of memory.</td>
</tr>
<tr>
<td>$1C02</td>
<td>MaxBlock</td>
<td>Returns size of largest free block in memory.</td>
</tr>
<tr>
<td>$1D02</td>
<td>TotalMem</td>
<td>Returns total memory size of system.</td>
</tr>
<tr>
<td>$1E02</td>
<td>CheckHandle</td>
<td>Checks to see if a given handle exists in the handle list.</td>
</tr>
<tr>
<td>$1F02</td>
<td>CompactMem</td>
<td>Forces memory compaction.</td>
</tr>
<tr>
<td>$2002</td>
<td>HLock</td>
<td>Locks a handle (cannot be purged or moved).</td>
</tr>
<tr>
<td>$2102</td>
<td>HLockAll</td>
<td>Locks all handles for a given ID.</td>
</tr>
<tr>
<td>$2202</td>
<td>HUnLock</td>
<td>Unlocks a given handle.</td>
</tr>
<tr>
<td>$2302</td>
<td>HUnLockAll</td>
<td>Unlocks all handles for a given ID.</td>
</tr>
<tr>
<td>$2402</td>
<td>SetPurge</td>
<td>Sets purge level for a block.</td>
</tr>
<tr>
<td>$2502</td>
<td>SetPurgeAll</td>
<td>Sets purge level for all blocks for a given ID.</td>
</tr>
<tr>
<td>$2B02</td>
<td>BlockMove</td>
<td>Copies a range of memory from one address to another.</td>
</tr>
<tr>
<td>$2802</td>
<td>PtrToHand</td>
<td>Copies a range of memory from one address to a block specified with a handle.</td>
</tr>
<tr>
<td>$2902</td>
<td>HandToPtr</td>
<td>Copies a range of memory from a block specified with a handle to a given address.</td>
</tr>
<tr>
<td>$2A02</td>
<td>HandToHand</td>
<td>Copies a range of memory from a block specified with a handle to a block specified with another handle.</td>
</tr>
</tbody>
</table>

Table 17-4 lists some of the possible errors which may be encountered from Memory Manager routines.
The Memory Manager and Miscellaneous Tools

Table 17-4. Memory Manager Errors

<table>
<thead>
<tr>
<th>Code</th>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0201</td>
<td>MemoryErr</td>
<td>Unable to allocate block.</td>
</tr>
<tr>
<td>$0202</td>
<td>EmptyErr</td>
<td>Illegal operation on empty handle.</td>
</tr>
<tr>
<td>$0203</td>
<td>NotEmptyErr</td>
<td>Empty handle expected for that operation.</td>
</tr>
<tr>
<td>$0204</td>
<td>LockErr</td>
<td>Illegal operation on a locked or immovable block.</td>
</tr>
<tr>
<td>$0205</td>
<td>PurgeErr</td>
<td>Attempt to purge an unpurgeable block.</td>
</tr>
<tr>
<td>$0206</td>
<td>HandleErr</td>
<td>Invalid handle given.</td>
</tr>
</tbody>
</table>

Using the Memory Manager

Program 17-1 is a ProDOS 8 application that will load a number of hi-res pictures from disk, store each away in memory using the Memory Manager, and then successively move them back onto the hi-res screen to create a fast-action slide show.

Any regular hi-res pictures will work with this program; the only requirement is that they be named PICTURE.A, PICTURE.B, PICTURE.C, and so forth on the same disk as this sample program. If you don’t have any hi-res pictures, Program 17-2 is an Applesoft BASIC program that will create 18 hi-res pictures in something just vaguely like animation. You may want to configure your Apple IIGS RAM disk to 800K to hold all the pictures, or you can use a formatted, but otherwise blank, 3½-inch disk to hold the pictures and the slide show object file.

Without going into great detail, suffice it to say that the Applesoft BASIC program simulates the dropping of a box, which then bounces off to the right of the screen. Unfortunately, because of the size of disk space, the program is limited to about 20 frames. The result is a very spotty animation. Most importantly, the assembly language program demonstrates just how quickly the Memory Manager can move large blocks of memory. If you’ve got the disk space and the inclination, you can change the 20 in line 105 to a larger number and the value of ACC on line 35 to 3.

This is the longest source listing presented so far in this book. It is true that assembly language programs of any consequence tend to be very long in terms of lines of code. Even a simple program on the Apple IIGS that uses the various tools can easily run from 1000 to 4000 lines of source code. Fortunately, each line is only a few characters for each instruction, and the typing goes quickly. As with previous listings, a checksum is included at the end of the listing to help you be sure the lines have been entered correctly.

You’ll notice that the program is similar to, and in fact a derivative of,
the P8 File Dump Sample program in Chapter 13. You may find it easier to
start with that listing and to use the existing parameter blocks at the end and
error message routines within it as a starting point for this program.

Because many of the ProDOS file-handling principles have been dis­
cussed in earlier chapters, we’ll concentrate the discussion on just those parts of
the program that deal with the Memory Manager, or that are otherwise
unusual.

Program 17-1. Memory Manager Demo: Slide Show

```assembly
;********************************************************************************
* MEMORY MANAGER DEMO PROGRAM *
* PRODOS 8 SYSTEM FILE *
* MERLIN ASSEMBLER *
********************************************************************************

ORG $2000

* DSK MM.DEMO.P8

MLI       EQU $BF00    ; STD. PRODOS 8 ENTRY
COUT      EQU $FDED
HOME      EQU $FC58
RDKEY     EQU $FD0C    ; MONITOR READ KEY ROUTINE
PRBYTE    EQU $FD0C    ; PRINT ACC. AS HEX NUMBER
HGR2      EQU $FD0C    ; Applesoft BASIC 'HGR2' ROUTINE
SETTXT    EQU $FD0C    ; Applesoft BASIC 'TEXT' ROUTINE
KYBD      EQU $C000
STROBE    EQU $C010

SCREEN    EQU $4000    ; HI-RES PAGE TWO MEMORY
HNDPTR    EQU $0006   ; $06,07

; SETUP STARTING CONDITIONS

START
SETRES   LDA  #$4C    ; JMP INSTRUCTION
         STA  $3F8     ; CTRLY VECTOR
         LDA  #$<RESUME ; LOW BYTE OF SHUTDOWN ADDR.
         STA  $3F9     ; LOW BYTE OF CTRL-Y VECTOR
         LDA  #$>RESUME
         STA  $3FA     ; HIGH BYTE OF CTRL-Y VECTOR

SETUP   CLC
XCE
```

348
### The Memory Manager and Miscellaneous Tools

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>002014: C2 30</td>
<td>REP $30</td>
<td>SET FULL 16-BIT MODE</td>
</tr>
<tr>
<td>002015: TLSTART</td>
<td>ToolCall $0201</td>
<td>TLStartUp</td>
</tr>
<tr>
<td>002016: MMSTART</td>
<td>ToolCall $0202</td>
<td>MMStartUp</td>
</tr>
<tr>
<td>002018: 68 51</td>
<td>PLA</td>
<td>GET OUR ID</td>
</tr>
<tr>
<td>002019: SD 58 22 52 STA ID</td>
<td>KEEP IT ON HAND</td>
<td></td>
</tr>
<tr>
<td>00201C: 09 00 01 MAKEID</td>
<td>ORA #$0100</td>
<td>SET AUX ID = 1</td>
</tr>
<tr>
<td>00201E: 0D 5A 22</td>
<td>STA ID2</td>
<td>SAVE SUB-ID</td>
</tr>
<tr>
<td>002020: A9 DC 21</td>
<td>INIT</td>
<td>ADDRESS OF 1ST PICTURE HANDLE</td>
</tr>
<tr>
<td>002022: 85 06</td>
<td>STA HNDPTR</td>
<td>POINTER TO IT.</td>
</tr>
<tr>
<td>002024: 38</td>
<td>SEC</td>
<td>FREE 16-BIT OPERATIONS</td>
</tr>
<tr>
<td>002025: FB</td>
<td>XCE</td>
<td>BACK TO 8 BITS...</td>
</tr>
<tr>
<td>002027: 20 00</td>
<td>JSR MLI</td>
<td>OPEN COMMAND</td>
</tr>
<tr>
<td>002028: 0D 0A 0B</td>
<td>STA OPENTBL</td>
<td>OPEN CMD TABLE</td>
</tr>
<tr>
<td>00202A: 90 11</td>
<td>BCC OPEN2</td>
<td>NO ERROR</td>
</tr>
<tr>
<td>00202C: C9 46</td>
<td>CMP #$46</td>
<td>FILE NOT FOUND ERR</td>
</tr>
<tr>
<td>00202E: DA NAMEEND-1</td>
<td>LDX</td>
<td>FILE COUNTER BYTE</td>
</tr>
<tr>
<td>002030: EP</td>
<td>CPX #&quot;A&quot;</td>
<td>1ST FILE?</td>
</tr>
<tr>
<td>002032: FO 03</td>
<td>BEQ :1</td>
<td>YEP</td>
</tr>
<tr>
<td>002034: 4C DD 20</td>
<td>JMP SHOW</td>
<td>NOPE, WE GOT AT LEAST ONE!</td>
</tr>
<tr>
<td>002036: 4C 66 21</td>
<td>JMP ERROR</td>
<td>PRODOS ERROR</td>
</tr>
<tr>
<td>002038: AD A4 21</td>
<td>JSR MLI</td>
<td>OPEN COMMAND</td>
</tr>
<tr>
<td>00203A: 8D A6 21</td>
<td>STA REFNUM1</td>
<td>STORE REF NUMBER</td>
</tr>
<tr>
<td>00203C: A9 04</td>
<td>READ</td>
<td># OF PARMS FOR 'READ'</td>
</tr>
<tr>
<td>00203E: 8D A5 21</td>
<td>STA READTBL</td>
<td>MODIFY TABLE ENTRY</td>
</tr>
<tr>
<td>002040: CA</td>
<td>JSR MLI</td>
<td>READ COMMAND</td>
</tr>
<tr>
<td>002042: A5 21</td>
<td>DA READTBL</td>
<td>READ CMD TABLE</td>
</tr>
<tr>
<td>002044: 90 03</td>
<td>BCC CLOSE</td>
<td>NO ERRORS...</td>
</tr>
</tbody>
</table>
Chapter 17

002069: 4C 66 21 96 JMP ERROR ; PRODOS ERROR MSSG
96
97
98
99 * CLOSE THE FILE
100
101
102 CLOSE LDA #$01 ; REWRITE READTBL
103 STA READTBL ; # OF PARMS = 1
104 JSR ML1
105 DFB $CC ; CLOSE COMMAND
106 DA READTBL ; SAME TABLE AS 'READ'
107 BCC :1 ; NO ERROR
108 JMP ERROR ; PRODOS ERROR
109 :1 ; PROGRAM CONTINUES HERE...
110
111 * REQUEST A MEMORY BLOCK TO STORE PICTURE
112
113
114
115 CLC
116 XCE
117 REP $30 ; FULL 16-BIT MODE
118
119 GETHNDL PushLong #$0000 ; PUSH ROOM FOR RESULT
120 PushLong #$2000 ; SIZE OF BLOCK NEEDED
121 PushWord ID2 ; GET DATA ID AND PUSH IT
122 PushWord #$0000 ; ATTRIBUTE BYTE
123 ; UNLOCKED, NOT PURGEABLE
124 PushLong #$0000 ; ADDRESS (NONE NEEDED)
125
126 ToolCall $0902 ; NewHandle
127 ; GET BLOCK OF MEMORY
128
129 :1 PLA ; GET LOW WORD OF HANDLE
130 STA (HNDPTR) ; STORE IN LOW WORD POSN
131 LDY #$02
132 PLA ; GET HIGH WORD OF HANDLE
133 STA (HNDPTR),Y ; STORE IN HIGH WORD POSN
134
135
136 * MOVE FROM SCREEN TO GS MEMORY
137
138
139 STORE PushLong SCREEN ; SCREEN ADDRESS (DATA SOURCE)
140 LDY #$02
141 LDA (HNDPTR),Y ; GET HIGH WORD OF HANDLE
142 ; PUSH ON STACK (DESTINATION)
143 LDA (HNDPTR) ; GET LOW WORD OF HANDLE
144 PHA
145 PushLong #$2000 ; NUMBER OF BYTES TO BE COPIED
146
147 ToolCall $2802 ; PtrToHand

350
The Memory Manager and Miscellaneous Tools

; PTRTO-HANDLE MOVE COMMAND

; * ADVANCE HANDLE STORAGE TO NEXT GROUP

NXTHNDL CLC
LDA HNDPTR
ADC #$04 ; ADVANCE 4 BYTES
STA HNDPTR

; * INCREMENT SUFFIX TO PICTURE NAME . . .

NXTFILE SEC
SEC
XCE ; BACK TO 8 BITS . . .

; CYCLE THROUGH "SLIDE SHOW"

SHOW CLC
XCE

; FULL 16-BIT MODE

FIRST LDA #HNDLI ; ADDRESS OF 1ST PICTURE HANDLE
STA HNDPTR ; POINTER TO IT.

; MOVE PICTURE DATA ONTO SCREEN

XFER LDY #$02
LDA (HNDPTR),Y ; HIGH WORD OF HANDLE
PHA
LDA (HNDPTR) ; LOW WORD OF HANDLE
PHA
PushLong #$3000 ; DESTINATION
PushLong #$2000 ; NUMBER OF BYTES TO COPY

ToolCall $2902 ; HandToPtr
; HANDLE-TO-POINTER MOVE COMMAND

; CLEAR HIGH BYTE

CHKKEY LDA KYBD
AND #$00FF
CMP #$0000
BCC NEXT ; NO KEYPRESS

BIT STROBE ; CLEAR KEYBOARD

351
002114: 4C 2D 21  
002117: 18  
00211A: 69 04 00  
00211D: 85 06  
00211F: B2 06  
002121: D0 C3 =20E6  
002123: AO 02 00  
002126: Bl 06  
002128: FO 87 =20El  
00212A: 4C E6 20  
002136: 38  
002137: FB  
002138: 20 99 F3  
00213B: 20 58 FC  
00213E: 20 00 BF  
002141: 00 00  
002146: 48  
00214D: A0 00  
002152: B9 AD 21  
00215B: F0 06 =217A  
002160: 20 FD  

JMP SHUTDOWN ; KEYPRESS = TIME TO QUIT

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

* INCREMENT HNDPTR TO NEXT HANDLE

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

002118: A5 06  
00211A: 69 04 00  
00211D: 85 06  
00211F: B2 06  
002121: D0 C3 =20E6  
002123: AO 02 00  
002126: Bl 06  
002128: FO 87 =20El  

NEXT

LDA HNDPTR

ADC #$04 ; ADVANCE 4 BYTES

STA HNDPTR

LDA (HNDPTR) ; LOW WORD OF NEXT HANDLE

BNE XFER ; >$00 MEANS REAL DATA

LDY #$02

LDA (HNDPTR),Y ; HIGH WORD OF NEXT HANDLE

BEQ FIRST ; $00 = END OF LIST = ANOTHER ROUND

XFER

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

* SHUTDOWN THINGS AND QUIT

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

SHUTDOWN PushWord ID2 ; GET DATA (PICTURE) ID

ToolCall $1102 ; DisposeAll

; RELEASE ALL DATA BLOCKS

MMSHUT PushWord ID ; MAIN ID (APPLICATION)

ToolCall $0302 ; MMShutDown

; TELL MM WE'RE DONE

TLSHUT ToolCall $0301 ; TLShutDown

SEC

XCE ; BACK TO 8 BITS...

JSR SETTXT ; BACK TO TEXT MODE

JSR HOME

QUIT JSR MLI ; DO QUIT CALL

DBF $65 ; QUIT CALL COMMAND VALUE

DA QUITTBLS ; ADDRESS OF PARM TABLE

BRK $00 ; SHOULD NEVER GET HERE...

ERROR PHA ; SAVE ERROR CODE

JSR SETTXT ; SET TEXT MODE

JSR HOME ; CLEAR SCREEN

LDY #$00 ; INIT Y-REG

LDA MSG1,Y ; GET CHAR TO PRINT

BEQ PROCODE

JSR COUT ; PRINT IT

QUITTBL ; ADDRESS OF PARM TABLE

BRK $00 ; SHOULD NEVER GET HERE...
The Memory Manager and Miscellaneous Tools

002177: C8 251 INY ; NEXT CHAR
002178: D0 F5 =216F 252 BNE :1 ; WRAPAROUND PROTECT
00217A: 68 254 PROCODE PLA ; RETRIEVE ERROR CODE
00217B: 20 DA FD 255 JSR PRBYTE ; PRINT IT
00217E: A0 00 256 LDY #$00 ; INIT Y-REG
002180: B9 BD 21 257 LDA MSSG1A,Y ; GET CHAR TO PRINT
002183: C8 258 BNE :1 ; WRAPAROUND PROTECT
002185: 20 DA FD 259 JSR COUT ; PRINT IT
002188: 68 260 INY ; NEXT CHAR
002189: D0 F5 =2180 261 BNE :1 ; WRAPAROUND PROTECT
002188: 20 OC FD 262 ERDONE JSR RDKEY ; WAIT FOR KEYPRESS
00218E: 4C SE 264 JMP QUIT ; QUIT PROGRAM...
002191: 18 265 RESUME CLC
002192: FB 266 XCE
002193: C2 30 267 REP $30 ; 16-BIT MODE...
002195: 4C 2D 21 268 JMP SHUTDOWN
002198: 04 269 QUITTBL DFB 4 ; NUMBER OF PARMS FOR QUIT
002199: 00 270 DFB $00 ; QUIT TYPE (0 = STD. QUIT)
00219A: 00 00 271 DA $0000 ; NOT NEEDED FOR STD. QUIT
00219B: 00 00 272 DA $0000 ; NOT USED AT PRESENT
00219C: 00 00 273 DA $0000 ; NOT USED AT PRESENT
00219D: 00 00 274
00219F: 03 275 OPENTBL DFB 3 ; NUMBER OF PARMS FOR OPEN = 3
0021A0: D2 21 276 PNAME DA NAME ; POINTER TO PATHNAME
0021A2: 00 23 277 DOSBUF DA $4000 ; POINTER TO PRODOS BUFFER
0021A4: 00 278 REFNUM DFB $00 ; PRODOS FILE REFERENCE NUMBER
0021A5: 00 279 DA $0000 ; NOT USED AT PRESENT
0021A6: 00 280 REFNUM1 DFB $00 ;REFERENCE NUMBER (= REFNUM)
0021A7: 00 40 281 DA $4000 ; POINTER TO DATA BUFFER (HGR2)
0021A9: 00 282 DA $2000 ; READ ENTIRE SCREEN
0021AB: 00 00 283 DA $0000 ; NUMBER OF CHARACTERS READ.
0021AD: 8D 284 MSSG1 HEX 8D ; PRINT RETURN FIRST
0021AE: D0 D2 CF C4 285 ASC "PRODOS ERROR $",00
0021BD: 8D 286 MESSG1A HEX 8D ; ANOTHER CARRIAGE RETURN

353
0 REM ANIMATION GENERATOR
5 REM SIMULATES A BOUNCING BOX...
10 X = 20: Y = 20: REM STARTING POSITION
20 XV = 0: REM HORIZONTAL VELOCITY
30 YV = 0: REM VERTICAL VELOCITY
35 ACC = 5
40 W = 10: REM WIDTH OF BOX
50 H = 9: REM HEIGHT OF BOX
60 HGR : HCOLOR = 3: HPL0T 0,0
65 CALL 62454
70 POKE -16302,0: REM FULL MODE
80 PRINT CHR$ (4); "PREFIX /RAM5"
100 REM DROP BOX
105 FOR T = 1 TO 20
110 HCOLOR = 3: HPL0T 0,0
115 CALL 62454: REM CLEAR TO COLOR
120 HCOLOR = 0: HPL0T 0,170 + H TO 279,170 + H
140 HCOLOR = 2: REM BLUE
150 FOR I = X TO X + W
The Memory Manager and Miscellaneous Tools

155 HPLT I,Y TO I,Y + H
160 NEXT I
170 X = X + XV; Y = Y + YV; YV = YV + ACC: IF INT (X / 2) <> X / 2 THEN X = X + 1:
  REM MAKE EVEN
180 IF Y > 170 THEN Y = 170:XV = ACC * 5;YV = YV - 3 * ACC;YV = YV * -1
190 IF X+W > 279 THEN 999
200 REM SAVE IMAGE
210 PRINT CHR$ (4);"BSAVE PICTURE.",CHR$ (192 + T);",A$2000,L$1FF8"
215 NEXT T
999 GET A$: TEXT

A Closer Look at the Slide Show

The program begins with clearing the hi-res screen using an internal Applesoft routine, HGR = $F3D8, which is equivalent to the HGR command in BASIC. As with the ProDOS 8 file dump program, the program then sets up the Control-Y vector as a “back door” should the program unexpectedly crash at some point. The difference in this program is that, instead of pointing directly to a ProDOS quit routine, the Control-Y vector is now set up to point to a small routine called RESUME at the end of our listing. Because we’ll need to shut down the tool sets our program has started up, and to de-allocate any memory that may have been allocated at that point, it will be necessary to make sure we’re in the full native 16-bit mode of the 65816 before jumping to the tool shutdown routines in our program, and, ultimately, to the ProDOS quit command.

The RESUME routine, lines 268-271, simply sets the processor to the 16-bit native mode before jumping to SHUTDOWN, which we’ll talk about shortly.

Lines 47-58 do something you’ll be seeing a lot of in future Apple II GS programs, namely starting up the GS tools needed for a particular application. In this case, they’re the Tool Locator and the Memory Manager. When the Memory Manager is started up, it gives us the ID for our application; this is modified to create a sub-ID for our data blocks that will hold the loaded pictures. Since we’re not creating a separate application, there is no need to startup the Miscellaneous Tools, nor to get a new ID from that tool set.

The call diagram for MMStartUp looks like this:

MMStartUp ($0202)

Stack Before Call:

| Previous Contents | +SP: |
You should also notice that the parameter block for the READ command (lines 288-292) is different from the file dump command in that this time it reads the entire $2000 bytes directly into the hi-res page, without using any other buffer area. This makes for a very fast and efficient file load.

As each picture is loaded (lines 69-108), a 16K block of memory will be allocated, and a handle to that memory block will be returned by the Memory Manager (lines 119-133). These handles, which are the only true identifiers of each memory block, will be stored in a data table at the end of the program, starting at HNDL1 (line 309). To make the program flexible and able to accomodate a variable number of pictures, the data block within the program at HNDL1 is 124 bytes long—enough room for 30 pictures plus 4 zeros at the end to flag the end of the list. If fewer pictures are read in, a zero will still be found at the end of the list because we’ve used the Merlin DS 120,0 instruction, which fills the data block entirely with zeros. You can increase the 120 to a larger value if you wish to load more pictures.

To allocate each memory block, the program shifts to the 16-bit mode and uses the NewHandle call to request a memory block. Figure 17-1 is the call diagram for NewHandle.

Figure 17-1. NewHandle ($0902)
The Memory Manager and Miscellaneous Tools

Stack After Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
<th>Long: Address of handle to new block.</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Handle</td>
<td>+SP: Stack pointer after return from routine.</td>
</tr>
</tbody>
</table>

When the handle is returned, its address is stored in the HNDL1 data table using the pointer HNDPTR, which was set up at the beginning of the program on lines 57,58. As each picture is loaded and a handle is assigned, HNDPTR will be incremented to the next storage position in the section NXTHNDL, lines 154–157.

Once the memory block to store the picture has been obtained, the picture data is moved to the allocated memory block by the PtrToHand tool call. This call uses a pointer to a memory location by transfers the specified number of bytes to a memory block specified by a handle. It’s important to note that no particular check is done to make sure that the receiving block is large enough for the number of bytes you’re writing to it, so you must make sure you’ve allocated a large enough block for the data. The call diagram for PtrToHand is shown in Figure 17-2.

Figure 17-2. PtrToHand ($2802)

Stack Before Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
<th>Long: Pointer to beginning address of block.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourcePointer</td>
<td>Long: Address of handle to destination block.</td>
</tr>
<tr>
<td>DestHandle</td>
<td>Long: Number of bytes to copy.</td>
</tr>
<tr>
<td>Count</td>
<td>+SP: Stack pointer after setup.</td>
</tr>
</tbody>
</table>

Stack After Call:

| Previous Contents | +SP: Stack pointer after return from routine. |

After incrementing the pointer to the handle storage list, NXTFILE (line 163) increments the suffix letter on the filename A to B, B to C, and so on, and loops back until a File Not Found error ($46) indicates there are no more pictures to be loaded.
Chapter 17

Once all the pictures are loaded, the SHOW routine cycles through the handle list, transferring the data back from the memory block, indicated by each handle, to the screen. The tool call HandToPtr is used for that. The call diagram is similar to PtrToHand (Figure 17-3).

Figure 17-3. HandToPtr ($2902)

Stack Before Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourceHandle</td>
</tr>
<tr>
<td>DestPointer</td>
</tr>
<tr>
<td>Count</td>
</tr>
</tbody>
</table>

Stack After Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>+-SP: Stack pointer after setup.</td>
</tr>
</tbody>
</table>

Each time through the transfer loop, the list of handles is scanned until a handle address of $00/0000 is encountered (lines 206-216), at which point the cycle is repeated. When a keypress is detected, the program jumps to the SHUTDOWN routines.

Whenever a routine using the Apple IIGS tools quits, it should first de-allocate any additional memory it has obtained. Lines 222, 223 take care of this by passing the sub-ID to the command DisposeAll, which de-allocates all memory blocks associated with the sub-ID. You must be very sure not to do this call with your application's ID; otherwise you will have de-allocated yourself at that point, so will be vulnerable immediately to another application somewhere in the computer using your memory while you're trying to execute the last few instructions of your program.

Once the memory has been de-allocated, the Memory Manager and Tool Locators are also shut down, and the ProDOS quit command will return us to whatever program selector was used to launch our application.

Obviously, the Tool Locator and Memory Manager tools are used by the system itself, so cannot really be shut off. The real purpose of the startup and shutdown calls is to just tell these tools that you would like their assistance at the beginning of your program, and to then tell them that you're finished at the
end. At the present, these calls for some tools may not do anything at all, but you should build them into all of your programs to provide for the possibility that at some point in the future the startup and shutdown calls for even apparently permanent tools like the Tool Locator may be required for proper system operation.

**The Miscellaneous Tool Set**

The Miscellaneous tool set is a collection of routines that do not fall into a major group the way the Memory Manager and other tools do. They also include some of the more fundamental tool functions that are called by the internal routines of the other tool sets to create the more complex functions.

For example, the Miscellaneous tool set includes the GetNewID function. Although your program will probably never need to use this, the function is used by the System Loader in assigning new UserlDs to applications as they are loaded. Other Miscellaneous tool set commands include reading the mouse. Again, these are rarely used by themselves in an application, but a more sophisticated tool set, the Event Manager, uses the mouse routines to create the more generalized command, GetNextEvent, which includes not only mouse clicks, but keyboard activity as well.

Although we've listed most of the tool calls in the Tool Locator and Memory Manager tools, these lists can get quite large. For example, there are over 200 calls in the QuickDraw tool set alone. For this reason, the list below just describes some of the highlights of each tool set discussed.

It should also be mentioned again that it is impossible for this book to really give a thorough discussion of all the Apple II GS tools. The intent now is to introduce you to the major concepts so that you have a foundation for the information presented in other books. For specific information on the the Apple II GS tools, I recommend *Exploring the Apple II GS* by Gary Little and *COMPUTE!*'s *Mastering the Apple II GS Toolbox* by Morgan Davis and Dan Gookin.

**Miscellaneous Tool Set Calls**

<table>
<thead>
<tr>
<th>Command Value</th>
<th>Command Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0102</td>
<td>MTStartUp</td>
<td>Starts up Miscellaneous tool set.</td>
</tr>
<tr>
<td>$0302</td>
<td>MTShutDown</td>
<td>Tells Miscellaneous tool set you’re finished.</td>
</tr>
<tr>
<td>$0D03</td>
<td>ReadTimeHex</td>
<td>Returns date and time in number value format, including the day of the week (0–6).</td>
</tr>
<tr>
<td>$0F03</td>
<td>ReadTimeASCII</td>
<td>Returns data and time as a string of characters.</td>
</tr>
<tr>
<td>$1703</td>
<td>ReadMouse</td>
<td>Returns position and status of mouse.</td>
</tr>
<tr>
<td>$1803</td>
<td>InitMouse</td>
<td>Initializes mouse.</td>
</tr>
<tr>
<td>$1C03</td>
<td>ClampMouse</td>
<td>Sets clamp values for mouse.</td>
</tr>
</tbody>
</table>
Chapter 17

$2003  GetNewID  Create new UserID.
$2103  DeleteID  Delete specified ID from ID list. Doesn’t de­
          allocate any memory.
$2503  GetTick  Returns number of ticks (1/60 of a second) 
          since computer was started up.
$2603  PackBytes  Compresses a block of data into a smaller 
          space by encoding it.
$2703  UnPackBytes  Unpacks encoded data back into its original 
          form.

The most likely Miscellaneous tool error is in connection with
GetNewID. It is $030B for IDNotAvailable, which is returned when there are
already 255 UserIDs assigned to a given Type, and an additional ID tag is not
available.

Program 17-3 shows how the Miscellaneous tool set clock-reading
routines might be used from an Applesoft BASIC program to return both the
time and the day of the week.

This program uses the string-variable passing techniques presented in
Chapter 12. The new additions, of course, are the routines to read the built-in
Apple II GS clock. Two calls are needed because ReadTimeHex returns the day
of the week, but not the time as a string. ReadTimeASCII, on the other hand,
returns the time as a string, but does not include the day of the week. Figure
17-4 and 17-5 show call diagrams for these two commands.

This routine can be tested with Program 17-4.

Figure 17-4. ReadTimeHex ($0D03)
Stack Before Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Space for Result</td>
<td>Word:</td>
</tr>
<tr>
<td></td>
<td>Allow space for result.</td>
</tr>
<tr>
<td>Space for Result</td>
<td>Word:</td>
</tr>
<tr>
<td></td>
<td>Allow space for result.</td>
</tr>
<tr>
<td>Space for Result</td>
<td>Word:</td>
</tr>
<tr>
<td></td>
<td>Allow space for result.</td>
</tr>
<tr>
<td>Space for Result</td>
<td>Word:</td>
</tr>
<tr>
<td></td>
<td>Allow space for result.</td>
</tr>
<tr>
<td>←SP: Stack pointer after setup.</td>
<td></td>
</tr>
</tbody>
</table>
The Memory Manager and Miscellaneous Tools

Stack After Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>WeekDay</td>
</tr>
<tr>
<td>Month</td>
</tr>
<tr>
<td>Year</td>
</tr>
<tr>
<td>Minute</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte: 1–7</th>
<th>Byte: Null</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte: 0–11</td>
<td>Byte: 0–30</td>
</tr>
<tr>
<td>Byte: 0–99</td>
<td>Byte: 0–23</td>
</tr>
<tr>
<td>Byte: 9–59</td>
<td>Byte: 0–59</td>
</tr>
</tbody>
</table>

SP: Stack pointer after return from routine.

Figure 17-5. ReadTimeASCII ($0F03)

Stack Before Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>BufferAddress</td>
</tr>
</tbody>
</table>

Long: Address of where to write time string.

←SP: Stack pointer after setup.

Stack After Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
</tr>
</thead>
</table>

←SP: Stack pointer after return from routine.

Program 17-3. Clock Reading

```
1  ***********************************************
2  *
3  * IIGS TIME ROUTINE *
4  * SYNTAX: CALL 768,A$ [,D] *
5  * WHERE A$ IS RETURNED WITH *
6  * TIME STRING, AND OPTIONAL D *
7  * RETURNS DAY OF WEEK (0-6), *
8  * WHERE 0 = SUNDAY. *
9  *
10  * MERLIN 16 ASSEMBLER *
11  *
12  ***********************************************
13  =0083 14 VARPNT EQU $83 ; $83,84
14  =0085 15 FORPNT EQU $85 ; $85,86
15  =0200 16 BUFF EQU $200 ; INPUT BUFFER
16  =00B7 17 CHRGOT EQU $B7
17  =DA9A 19 SAVD EQU $DA9A
```
Chapter 17

=DD6C 20 CHKSTR EQU $DD6C
=DEBE 21 CHKCOM EQU $DEBE
=DFE3 22 PTRGET EQU $DFE3
=E3E9 23 MAKSTR EQU $E3E9

=DD6A 25 CHKNUM EQU $DD6A
=EB9D 26 GIVAYF2 EQU $EB9D
=EBF2 27 QINT EQU $EBF2
=0011 28 VARTYPE EQU $11 ; STR$ = $FF, NUM = $00
=0012 29 NUMTYPE EQU $12 ; INT = $80, REAL = $00
=DA63 30 LET2 EQU $DA63
=DA6B 31 LET3 EQU $DA6B
=009D 32 FAC EQU $9D
=0006 33 DAY EQU $06 ; $3C,3D

008000: 20 B7 00 BEGIN JSR CHRGOT ; CHECK CHAR AT TXTPTR
008003: C9 2C CMP "," ; COMMA?
008005: DO 03 =800A BNE :1 ; NOPE
008007: 20 BE DE JSR CHKCOM ; GOBBLE COMMA IF NEEDED

00800A: 20 E3 DF :I JSR PTRGET ; LOCATE VARIABLE
008010: 85 86 STA FORPNT ; SAVE WC. OF DATA
008012: 84 86 STY FORPNT+l

008014: 18 CLC
008015: FB XCE
008016: C2 30 REP $30 ; FULL 16-BIT MODE

008018: A2 02 03 MTSTART LDX #$0203 ; MISC. TOOLS STARTUP
00801B: 22 00 00 EI 51 JSL $E10000 ; NO ERROR LIKELY

00801F: F4 00 00 53 WEEK PEA $0000
008022: F4 00 00 54 PLA $0000
008025: F4 00 00 55 PLA $0000
008028: F4 00 00 56 PLA $0000 ; MAKE ROOM FOR 8 BYTES TO RETURN

00802B: A2 03 0D LDX #$0D03 ; ReadTimeHex
00802E: 22 00 00 EI 59 JSL $E10000 ; READ NUMBER VALUES FOR TIME
008032: 68 60 PLA ; GET MIN, SEC & DISCARD
008033: 68 61 PLA ; GET YR, HRS, AND SO ON
008034: 68 62 PLA ; GET MONTH, DAY
008035: 68 63 PLA ; GET DAY OF WEEK (HI BYTE)
008036: 85 06 64 STA DAY ; 2 BYTES FOR LATER USE.

008038: F4 00 00 66 TIME PEA "BUFF" ; POINTER TO BUFFER
00803B: F4 00 02 67 PEA BUFF
00803E: A2 03 0F LDX #$F03 ; ReadTimeASCII
008041: 22 00 00 EI 69 JSL $E10000 ; WRITE STRING TO MEMORY

008045: A2 02 03 LDX #$0303 ; MISC. TOOLS SHUTDOWN
The Memory Manager and Miscellaneous Tools

008048: 22 00 00 E1 72 JSL $E10000 ; NO ERROR LIKELY
00804C: 38 74 SEC
00804D: FB XCE ; BACK TO 8 BITS
00804E: A2 13 77 LDX #19 ; LEN OF TIME STRING
008050: BD 00 02 78 TLOOP LDA BUFF,X ; GET A CHARACTER
008053: 29 7F 79 AND #$7F ; FIX HI BIT
008055: 9D 00 02 80 STA BUFF,X ; PUT IT BACK
008058: CA 81 DEX
008059: 10 F5 =8050 82 BPL TLOOP ; TILL WE'RE DONE
00805B: A9 00 84 LDA #<BUFF ; LOCATION OF STRING DATA
00805D: AO 02 85 LIJV #>BUFF
00805F: A2 FF 86 LDX #$FF ; TERMINATOR CHARACTER
008061: SE 14 02 87 STX BUFF+20 ; PUT AT END OF STRING
008064: 20 E9 E3 89 JSR MAKSTR
008067: 20 9A DA 90 JSR SAVD ; STRING CREATED AND SENT
00806A: 20 87 00 93 CHECK JSR CHRGOf ; CHECK NEXT CHARACTER
00806D: DO 01 =8070 94 BNE DAYWEEK ; END OF-LINE OR COLON
00806F: 60 95 RTS
008070: 20 BE DE 97 DAYWEEK JSR CHKCOM ; Gobble next comma
008073: 20 E3 DF 98 JSR PTRGET ; Locate variable
008076: 20 6A DD 99 JSR CHKNUM ; var = num?
008079: 85 85 100 STA FORPNT ; for use by LET2/LET3
008078: 84 9F 101 STY FORPNT+1 ; as addr of variable data
00807F: A4 07 102 LDY DAY+1 ; lo byte of return value (day is hi)
008081: 85 9E 103 LDA #$00 ; hi byte always zero
008083: 84 9F 104 STA FAC+1
008085: A2 90 105 LDX #$90
008087: 25 12 106 AND NUMTYPE
008089: 20 9D EB 107 JSR GIVAYF2
00808C: A5 12 108 LDA NUMTYPE
00808E: 30 03 =8093 109 BMI PJ
008090: 4C 63 DA 110 JMP LET2 ; make a real var and go home.
008093: 20 F2 EB 111 PJ JSR QINT ; XVERT TO INTEGER
008096: 4C 6B DA 112 JMP LET3 ; THAT'S ALL!
008099: 51 113 CHKSUM CHK ; CHECKSUM FOR LISTING

--End Merlin-16 assembly, 154 bytes, errors: 0

363
Chapter 17

Program 17-4. Applesoft Clock Test

5 D$ = CHR$ (4)
10 PRINT D$"BL0AD TIME.TB,A$300"
15 FOR I = 1 TO 7: READ D$(I): NEXT I
20 TEXT : HOME
30 CALL 768,A$,.D
40 VTAB 1: PRINT D$(D);" ";A$
50 IF PEEK (-16384) < 128 THEN 30
55 POKE -16368,0: END
99 DATA SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
Chapter 18

QuickDraw and the Event Manager
Chapter 18
QuickDraw and the Event Manager

The G in Apple II GS stands for graphics. In Applesoft BASIC, the normal hi-res graphics screen has 280 horizontal and 192 vertical pixel positions (pixel is an abbreviation for picture element and is a measure of the visual graphics points on the screen). There are a choice of eight colors, although effectively there are only six, since black and white are each counted twice in the list of possible colors. The hi-res screen doesn't have a resolution of 280 dots in eight colors. That's because you can't plot any color you want at every dot position. If you don't care about the color (perhaps you're using a monochrome monitor), the screen does have 280 pixels. However, in a given color, the resolution is only 140 pixels, since half the dots are no longer usable.

Double Hi-Res
Double hi-res adds more physical positions across the screen, so that the monochrome resolution is 560 pixels, but again, there is a limitation introduced by adding colors (16 possible) that returns the effective resolution to 140 pixels in color. Some people insist that because a double hi-res color can be plotted anywhere, the resolution is really 560 pixels, but all you have to do is to set the Control Panel Display option to Color while you're in the DeskTop program (a double hi-res display) to see how adding color decidedly reduces the resolution (clarity) of the screen.

The Apple II GS introduces a new graphics mode, called super hi-res graphics, which has a greater resolution with true color choice at every pixel. In the first mode, the screen is 320 pixels wide by 200 high, and there are 16 colors available for each pixel at every position. There is a second mode, 640 pixels wide by 200 high, in which only 4 colors are available for a given position. However, through some clever planning in hardware, it's possible to put more colors on the entire line by mixing colors in a process called dithering. This works by putting, for example, a red pixel next to a yellow pixel to create the color orange.
Storing Information

You may have already noticed that there seems to be a relationship between resolution and the number of colors possible. This is easy to understand once you grasp the concept of information storage. Information as a concept, in relation to computers, has to do with the mathematical idea of uniquely describing some entity. This can be a letter of the alphabet, a word in a sentence, a formula for a calculation, or a visual element in a picture. There is a finite limit to how much information you can put in a single byte, or more precisely, in a single bit of any given byte.

For current hardware, a bit can only have two states, 0 or 1. This means that the information contained in that bit is limited to two states. We create meaningful information by combining bits to represent larger numbers, for example 01001011 = $47 = 71 can represent the seventy-first ASCII character, or the letter G. The challenge in computing is to devise schemes to pack as much unique information into the smallest memory space possible.

For a graphics screen, let’s start with the idea of a screen eight pixels wide by eight pixels high. For the time being, we’ll ignore color, and will just display a graphics image in black and white. A direct solution is to use the bits in eight bytes to represent each of the possible pixel positions. If a bit is 0, we’ll say that dot is off, or black. If the bit is 1, it will represent an illuminated spot. By adjusting the values of the eight bytes, we can create an image like this:

<table>
<thead>
<tr>
<th>Byte 0: 00000000 = $00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 1: 00011000 = $18</td>
</tr>
<tr>
<td>Byte 2: 00111100 = $3C</td>
</tr>
<tr>
<td>Byte 3: 01111110 = $7E</td>
</tr>
<tr>
<td>Byte 4: 11111111 = $FF</td>
</tr>
<tr>
<td>Byte 5: 00011000 = $18</td>
</tr>
<tr>
<td>Byte 6: 00011000 = $18</td>
</tr>
<tr>
<td>Byte 7: 00011000 = $18</td>
</tr>
</tbody>
</table>

This is how it might look on a screen:
QuickDraw and the Event Manager

With a little squinting, this looks a little like an arrow. Now suppose you wanted to increase the clarity of the image. This is another way of saying you want to increase the resolution of the image. *Clarity* is a way of saying you want more information communicated to your brain. By doubling the resolution to $16 \times 16$, we'll get a better image, but it will require *four times* as much memory:

```
Word 0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 = $0000
Word 1: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 = $0000
Word 2: 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 = $0180
Word 3: 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 = $03C0
Word 4: 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 = $07E0
Word 5: 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 = $0FF0
Word 6: 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 = $1FF8
Word 7: 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 = $3FFC
Word 8: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 = $7FFE
Word 9: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 = $FFFF
```

Even without the more graphic drawing, you can probably see the arrow much more clearly using zeros and ones, but remember it took much more memory to increase the resolution. Now suppose you wanted to have four colors, or even just four shades of gray. (It really doesn't matter which because we can design the computer video output hardware to translate any value from 0 to 3 into whatever color or shade of gray we wish.)

However, adding the color or shading will take still more memory. For four possible states, we'll have to use two bits in a byte somewhere (two bits = four possible values: %00 = 0, %01 = 1, %10 = 2, %11 = 3).

As you can see, adding graphics to any computer system puts great demands on the available memory. A normal hi-res graphics screen in Applesoft BASIC uses 8K of memory for the display. The super hi-res display uses 32K. The memory's fixed size creates the trade-off situation of either 320 pixels in 16 colors or 640 pixels in four colors.
QuickDraw and the Drawing Environment

In Applesoft BASIC, there are half a dozen or so commands for drawing on the hi-res screen. The Apple IIGS has a built-in tool set called QuickDraw that contains over 200 commands for super hi-res graphics operations. These include not just the mundane clearing of the screen and line-drawing routines, but also operations for defining rectangles, ovals, complex regions, as well as filling these with both colors and patterns. In addition, you have control over the entire drawing area as things are being drawn, and you can create clipping regions to automatically omit lines from any part of the final image. In fact, there are more commands in QuickDraw than there are in all the commands in either Applesoft BASIC or 65816 machine language.

But don’t let this large number of available commands discourage you. Fortunately, the important one, QDStartUp (QuickDraw StartUp) sets all of the defaults for you, and it only takes a few lines of actual code to draw lines and rectangles, to set colors, or to do the other things you’re likely to want to use. The hundreds of other commands are provided to create an immense flexibility for more exotic applications. For example, suppose you want to create a graphics image in memory that is larger than the screen—like a blueprint for a building—and you have QuickDraw create the entire image within the computer. Using some of those more advanced commands, it is possible to draw anywhere in memory—not just the super hi-res display itself—and to have a drawing area limited only by the available memory in your computer.

When QuickDraw is started, the default drawing area is the super hi-res screen. This corresponds to the part of memory in bank $E1 from $2000 to $9FFF (32K). The drawing area itself is the first 32000 bytes (200 lines × 160 bytes per line = 32000), from $2000 to $9CFF.

Remember that 32K is really 32,768. This leaves another 768 bytes to account for. The next 200 bytes are called scan line control bytes, or SCBs. There is one SCB for each line on the screen, and these can be used to individually assign the set of 16 colors that will be used on that line, whether that line is in 320 or 640 mode (yes, it’s possible to mix them on the same screen), and more. These are assigned the area from $9DOO to $9DFF. This amounts to 256 bytes, which leaves 56 bytes unused. Apple, however, calls these reserved for future use.

The next two pages of memory, $9E00 to $9FFF are used to store the color tables, or palettes, for the picture. Each color entry in a palette requires two bytes, so a complete palette of 16 colors requires 32 bytes. Two pages of memory provide room for up to 16 separate palettes. Each line of the screen display can use its SCB to select one of 16 possible palettes for that line’s choice of colors.

When a super hi-res picture is saved to disk, the SCBs and color tables

370
QuickDraw and the Event Manager

are saved along with the picture, for a total file size of 32K (32,768 bytes).

For the most part, this information is more for background than for actual use. When QuickDraw is started up, each line is set to use the default (#0) color table, and you won't have to explicitly set up either the SCBs or a color table.

The call diagram for QDStartUp is shown in Figure 18-1.

Figure 18-1. QDStartUp ($0204)

Stack Before Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>DirectPageLoc</td>
</tr>
<tr>
<td>MasterSCB</td>
</tr>
<tr>
<td>MaxWidth</td>
</tr>
<tr>
<td>UserID</td>
</tr>
</tbody>
</table>

| Word: Address of Direct Page for QuickDraw. |
| Word: SCB to use for every screen line  |
| (0=320; $80=640). |
| Word: Image width ($0=full screen). |
| Word: ID of application |

Stack After Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
</tr>
</thead>
</table>

Left SP:

To start up QuickDraw, you first need to get three pages of memory in bank 0 to use as its QuickDraw's direct page. This is done using NewHandle in the Memory Manager, which will be detailed in the example program coming up.

The Master SCB entry is the value that QuickDraw will use for each of the 200 lines on the screen display when it starts. This is your opportunity to tell QuickDraw whether you want the 320 or 640 mode for the entire screen. Use an SCB value of $0000 for the 320 mode, $0080 for the 640 mode.

It's possible to create drawing areas both larger and smaller than the super hi-res screen. The MaxWidth parameter is the width, in bytes of the image area. For most applications, just use zero here, for the full screen width.

The last parameter is the UserID you got from MMStartUp; it tells QuickDraw who is starting it up.

**Drawing Data Structures: Pens, Lines and Rectangles**

QuickDraw supports a very advanced drawing environment: You can control virtually every aspect of the image being created. To support this environment, there are a number of specific data structures that have been created, which will be referenced while using QuickDraw.
The first is the idea of a drawing pen. The pen is a pixel image, much like the arrow image just discussed, that can be moved around on the screen, leaving a trail as it is moved.

It’s possible to redefine the pen to be any image you wish, as you have probably seen in the many Apple II GS painting programs available.

The pen pattern, as it’s called, can also be made to draw in a variety of modes, such as reversing the background it draws on or painting a color or a pattern there. These different effects are done internally by QuickDraw using the Boolean AND, EOR, or ORA functions described in earlier chapters.

The pen pattern can be set to a specific color with the command SetSolidPenPat ($3704), in which a color value in the range $0 to $F (0 to 15) is passed to the routine.

When QuickDraw is initialized, the pen is a single pixel, and the color is black. Moving the pen as such does not draw a line. Instead, specific drawing commands like LineTo, discussed shortly, create the image.

To draw an image, another data structure is used, called a point, which is defined by an X and Y position. The Y coordinate comes first, and each is a two-byte value. Thus, the data structure for a point looks like this:

```
POINT DA $0000 ; Y COORDINATE
DA $0000 ; X COORDINATE
```

This will take four bytes in memory to store. For some operations, the values for a point will be pushed directly on the stack. For others, a pointer to the POINT data structure may be put on the stack so the routine will know where to find the coordinates to use.

Examples of these types of routines are MoveTo ($3A04) and LineTo ($3C04). MoveTo repositions the current pen location to the specified coordinate. To use it, you simply push the values for the coordinates on the stack before doing the call. Figure 18-2 is the call diagram for MoveTo.

Figure 18-2. MoveTo ($3A04)

Stack Before Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
</tr>
<tr>
<td>V</td>
</tr>
</tbody>
</table>

Word: Horizontal coordinate of point.
Word: Vertical coordinate of point.
←SP:
QuickDraw and the Event Manager

Stack After Call:

| Previous Contents | ←SF: |

LineTo is similar, but it draws a line using the current pen from the current position to the point specified. The call diagram for LineTo is the same as that for MoveTo.

A rectangle is defined by two points, one for the upper left corner, the second for the lower right corner. The data structure for a rectangle looks like this:

```
RECT
V1  DA $0000 ; VERT. POSN OF UPPER LEFT
H1  DA $0000 ; HORIZ. POSN OF UPPER LEFT
V2  DA $0000 ; VERT. POSN OF LOWER RIGHT
H2  DA $0000 ; HORIZ. POSN OF LOWER RIGHT
```

This takes a total of eight bytes. Rectangle-related objects are usually drawn by passing a pointer to the rectangle, rather than the values for the rectangle itself. We say rectangle-related, because the definition of a rectangle can be used as the basis for a variety of shapes. For example, an oval can be defined as the ellipse that fits inside the defined rectangle. A square rectangle defines a circle (see Figure 18-3).

Figure 18-3. Oval Defined by a Rectangle

In addition, a rectangle-related object can either be framed (the outline drawn), painted (filled in with a color or pattern), or inverted (the interior pixels are inverted).

Figure 18-4 is the call diagram for FrameRect, which draws the outline of the rectangle using the current pen and color.
Starting QuickDraw

Program 18-1 starts up QuickDraw, draws a line and a variety of shapes on the screen, and even includes some printed text.

When you run Program 18-1, the screen should clear to black, and a diagonal red line will be drawn across the screen. You should also see four shapes drawn, including rectangles, an oval, and a rounded rectangle. Pressing any key will quit the program and return to whatever program selector started up the program.

Let’s look at the listing to see how everything works. Of course, as with the others, it begins with setting the data bank equal to our program bank, and there is also the emergency recovery code set up with the Control-Y vector. That way, if any error should occur while you’re testing the program, you can press Control-Y to get back to the assembler.

There is one thing that will be different here: If the program crashes while the super hi-res display is on, you won’t be able to see the BRK message, since that is presented on the text screen. The only indication will be the beep heard when the BRK is encountered. If your program does stop, type

E1/C029: 21

and press Return. This will clear the super hi-res display bit (bit 7) in $C029 and will return the display to the text mode. You should be able to see the register dump from the BRK instruction at that point. Make a note of the address printed for the BRK and of the contents of the Accumulator. The address will tell you which tool call was not executed properly, and the Accumulator will hold the error code. Although certain error codes are provided in this book, you should get the official Apple Toolbox Reference by Addison-Wesley, or
After noting the necessary information, press Control-Y to go back to your program selector.

After starting up the Tool Locator and Memory Manager, a sub-ID is produced by setting the Aux field (found in the ID the Memory Manager gives us) to 1 (lines 37-39). This will be used when obtaining the direct-page memory block that QuickDraw requires.

The **GETDP** (Get Direct Page) section uses NewHandle to obtain a $300-byte block in bank 0. The attribute byte $C001 marks this as immovable and locked, and in a fixed bank (bank 0), which is necessary for a direct-page block.

NewHandle will return the handle to this block, but will not return its address. Remember that a handle is a pointer to a pointer, and that the actual address of the block is now part of the handle itself. Normally, you would not use an absolute address for a memory block, but since we know this block has been designated as immovable, its address will be constant until it’s de-allocated.

The process of examining a handle is called *dereferencing*. It is done by putting the address of the handle in our own direct-page pointer, and then using indirect addressing to look at the first four bytes of the handle. The first four bytes are always the actual address of the memory owned by that handle.

Line 50 pulls the handle off the stack for the direct page gotten from NewHandle, and stores that handle’s address in byte $00 of our own direct page. Line 51 then uses indirect addressing long to look into the first four bytes of the handle. The two-byte value loaded into the Accumulator at that point will be the address in bank 0 of the new direct page. Because we know this is in bank 0, there’s no need to look at the third and fourth bytes for the high word of the address, although if you were dereferencing just any old handle, this would be required.

In the full, nonmacro version, here are the instructions to dereference a handle:

```
DEREF LDA HNDL ; GET LOW WORD OF HANDLE
STA PTR ; STORE IN OUR OWN DP POINTER
LDA HNDL+2 ; GET HIGH WORD OF HANDLE
STA PTR+2 ; STORE IN DP HIGH WORD.
LDA [PTR] ; GET LOW WORD OF ADDRESS.
STA ADDR ; SOME STORAGE LOCATION
LDY #$02 ; PREPARE FOR NEXT INSTR.
LDA [PTR],Y ; GET HIGH WORD OF ADDRESS
STA ADDR+2 ; SAVE THAT TOO.
```
Chapter 18

Program 18-1 looks much simpler because a macro, `PullLong`, sets up our own direct-page pointer, and we only need the LDA [PTR] to retrieve the low word of the address.

Once a direct-page block for QuickDraw has been obtained, we can start things going. The label `QD` (line 54) begins the section that starts up QuickDraw. Referring back to the call diagram for QuickDraw, you can see we’re starting up in the 320-pixel mode. Since the direct-page address is still in the Accumulator from the LDA [$00] instruction on line 47, this can be pushed on the stack as the first line of QD.

When QuickDraw starts, the pen pattern color is black. `SETCOLOR` (lines 60–61 sets the pen pattern to the solid color red (color #7). You can use other values here for other colors. The standard color values for the 320 mode are as follows:

<table>
<thead>
<tr>
<th>Color Entry</th>
<th>Color</th>
<th>Master Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Black</td>
<td>$0000</td>
</tr>
<tr>
<td>1</td>
<td>Dark Gray</td>
<td>$0777</td>
</tr>
<tr>
<td>2</td>
<td>Brown</td>
<td>$0841</td>
</tr>
<tr>
<td>3</td>
<td>Purple</td>
<td>$072C</td>
</tr>
<tr>
<td>4</td>
<td>Blue</td>
<td>$000F</td>
</tr>
<tr>
<td>5</td>
<td>Dark Green</td>
<td>$0080</td>
</tr>
<tr>
<td>6</td>
<td>Orange</td>
<td>$0F70</td>
</tr>
<tr>
<td>7</td>
<td>Red</td>
<td>$0D00</td>
</tr>
<tr>
<td>8</td>
<td>Beige</td>
<td>$0FA9</td>
</tr>
<tr>
<td>9</td>
<td>Yellow</td>
<td>$0FF0</td>
</tr>
<tr>
<td>10</td>
<td>Green</td>
<td>$00E0</td>
</tr>
<tr>
<td>11</td>
<td>Light Blue</td>
<td>$04DF</td>
</tr>
<tr>
<td>12</td>
<td>Lilac</td>
<td>$0DAF</td>
</tr>
<tr>
<td>13</td>
<td>Periwinkle Blue</td>
<td>$078F</td>
</tr>
<tr>
<td>14</td>
<td>Light Gray</td>
<td>$0CCC</td>
</tr>
<tr>
<td>15</td>
<td>White</td>
<td>$0FFF</td>
</tr>
</tbody>
</table>

The column titled Color Value shows the two-byte value actually stored in color table (palette) #0 when QuickDraw starts up. If you look at just a select few of the colors, you'll see how the final color is determined from the number value:

```plaintext
4 Blue $000F
7 Red  $0D00
10 Green $00E0
```

Starting with the color values for Blue, Red and Green, you can see that each of these color values has a single hexadecimal digit in just one field. In fact, that is the meaning of each field in the color value. The pattern is:

$0RGB
The color video monitor for the Apple IIGS is called an RGB Monitor for a reason: It refers to the fact that, like your color TV, the image is formed by illuminating red, green and blue dots on the screen. Other colors are created by mixing these three colors. The value in the range of $0$ to $F$ for each position in the color value tells the video hardware how brightly to illuminate the particular color dot.

For each color value, the first nibble (4 bits) is unused, and so is equal to $0$. Black is designated by setting all three color elements to $0$; white is made by turning all three up to the maximum value.

<table>
<thead>
<tr>
<th>Color</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black</td>
<td>$\text{0000}$</td>
</tr>
<tr>
<td>White</td>
<td>$\text{0FFF}$</td>
</tr>
</tbody>
</table>

Shades of gray (14 in all, plus black and white) are made by varying the matching strengths of each color.

<table>
<thead>
<tr>
<th>Color</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dark Gray</td>
<td>$\text{0777}$</td>
</tr>
<tr>
<td>Light Gray</td>
<td>$\text{0CCC}$</td>
</tr>
</tbody>
</table>

A particular color can be created by adjusting the strength of one color element:

<table>
<thead>
<tr>
<th>Color</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dark Green</td>
<td>$\text{0080}$</td>
</tr>
</tbody>
</table>

by mixing two together:

<table>
<thead>
<tr>
<th>Color</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yellow</td>
<td>$\text{0FF0}$</td>
</tr>
</tbody>
</table>

or by mixing all three together in varying proportions:

<table>
<thead>
<tr>
<th>Color</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Periwinkle Blue</td>
<td>$\text{078F}$</td>
</tr>
</tbody>
</table>

Although this program doesn't create new colors, you can add the tool call `SetColorEntry` ($1004$) to the program to create new colors if you wish. The call diagram for `SetColorEntry` is shown in Figure 18-5.

Figure 18-5. `SetColorEntry` ($1004$)

Stack Before Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>TableNumber</td>
</tr>
<tr>
<td>EntryNumber</td>
</tr>
<tr>
<td>NewColorValue</td>
</tr>
</tbody>
</table>

Word: Table # ($0$ to $F$) default = $0$.

Word: Entry you want to change ($0$ to $F$).

Word: New Color Value.

←SP:
Chapter 18

Stack After Call:

Once the color has been set, the MoveTo and LineTo commands are used to draw a line from one corner of the screen to another (lines 63–70). Lines 71–75 then change the pen color to blue (#4), and draw a rectangle form using FrameRect ($5304). Notice that the address to RECT, defined on lines 179-183, is pushed on the stack as the input for FrameRect.

The process is then repeated for a filled-in rectangle (PaintRect = $5404), and oval (PaintOval = $5904), and a rounded rectangle (PaintRRect = $5E04). For the rounded rectangle, the width and height of another imaginary rectangle are passed. The final rounded rectangle is produced by replacing square corners of the rectangle with the corner of an oval specified by the height and width parameters.

As each new geometric figure is drawn, the subroutine SHIFT (lines 127–140) changes the horizontal and vertical starting coordinates of where the drawing begins. This is rewritten directly into the definition of the rectangle used for each of the figures.

Printing Text in Super Hi Res

There are also a number of routines in QuickDraw specifically for drawing text in different styles, called fonts, on the super hi-res screen. Although additional fonts can be loaded from disk, QuickDraw starts up with a default font, whose definition is stored in ROM, as the standard font. When text is drawn, the routines use both a color for the text itself, and for the background behind the text. At startup, this defaults to a black foreground for the text and a white background.

Since our screen is black, the first thing to do is to set the foreground and background colors for the text printing routines. Lines 103–108 do this
QuickDraw and the Event Manager

with SetBackColor ($A204) and SetForeColor ($A004). Printing the text is as simple as passing a pointer to the string, which is presumed to end in a zero, to the routine DrawCString ($A604). The MoveTo command is used to position the pen at the appropriate spot before drawing the text.

In Program 18-1, the string is defined on line 172. Notice that the high bit is clear for the string (Merlin uses single quotation marks to designate high bit clear; APW uses the directive MSB OFF). The high bit must be clear for the QuickDraw text printing routines to work properly.

If you want to print a ProDOS-type string with a leading length byte, an alternate routine, DrawString ($A504), can be used; the only difference is that DrawString expects a leading length byte in the string to be printed.

RDKEY then just waits for a keypress before doing the ProDOS quit command. As you can see, QuickDraw really is quite simple to use.

You'll hear a lot about clipping windows, something called the BoundsRect, and other things, but these are for the most part managed automatically by the system, and are of only minor concern if you're actually doing windows that can be scrolled and dragged around the screen, as we'll see in the next chapter. However, even in the windowed environment, it's still very easy to create just about any image you want.

Although there are over 200 different tool calls in QuickDraw, Table 18-1 lists a few to give you an idea of what's available.

Program 18-1. Simple Quickdraw Demo

```
008000: 4B
008001: AB
```

---

```merlinasm
008000:
        ; TELL MERLIN WE'RE IN 16 BITS
        MX      %00
        REL    QD.DEMO.L
        DSK    QD.DEMO.L
        LST    OFF
        USE    UTIL.MACS
        LST    ON
        EXP    OFF

= E100A8
        PRODOS  EQU $E100A8

= C000
        KYBD  EQU $00C000

= C010
        STROBE  EQU $00C010
```

---

```merlinasm
008000: 4B
008001: AB
```

STARTUP PHK
PLB ; DATA BANK = PROG. BANK
Chapter 18

008002: E2 30 24 SETRES SEP $30 ; 8-BIT MODE
008004: A9 5C 25 LDA #$5C ; JML (JMP LONG)
008006: 8F F8 03 00 26 STAL $3F8 ; CTRL VECTORE
00800A: C2 30 27 REP $30 ; 16-BIT MODE
00800C: A9 ED 28 LDA #RESUME
00800E: SF F9 03 00 29 STAL $3F9 ; $3F9,3FA
008012: A9 00 00 30 LDA #RESUME
008014: SF FB 03 00 31 STAL $3FB ; $3FB,3FC
008016: TL ToolCall $0201 ; TLStartUp
008018: 09 00 01 32 ORA #$0100 ; OUR SUB-ID
00801A: STA ID2
00801C: GETDP PushWord #$0000 ; SPACE FOR RESULT
00801E: PushWord #$300 ; AMT OF MEMORY NEEDED
008020: PushWord ID2 ; ID FOR OUR APPLICATION
008022: PushWord #$C001 ; TYPE: LOCKED, FIXED
008024: PushLong #$0000 ; BANK = $00, NO SPECIFIC ADDR.
008026: ToolCall $0902 ; NewHandle
008028: ; OBTAIN A MEMORY BLOCK IN BANK 0
00802A: PullLong $00 ; GET HANDLE & STORE IN OUR DP
00802C: LDA [$00] ; LONG INDIRECT LOAD
00802E: ; GETS NEW DP ADDRESS FOR TOOLS
008030: PushWord #$0000 ; MASTER SCB = DEFAULT (320)
008032: PushWord #$0000 ; MAX SCREEN SIZE
008034: PushWord ID ; OUR APPLICATION'S ID
008036: ToolCall $0204 ; QDStartUp
008038: SETCOWR PushWord #$7 ; COWR = 'RED'
00803A: ToolCall $3704 ; SetSolidPenPat
00803C: BEGINLN PushWord #$0000 ; HORIZ (X) = $00
00803E: PushWord #$0000 ; VERT (Y) = $00
008040: ToolCall $3A04 ; MoveTo
008042: DRAWLN PushWord #319 ; X = 320TH PIXEL
008044: PushWord #199 ; Y = 200TH PIXEL
008046: ToolCall $3C04 ; LineTo
008048: COLOR1 PushWord #$4 ; COLOR = 'BLUE'
00804A: ToolCall $3704 ; SetSolidPenPat
00804C: DRAWRCT PushLong #RECT ; ADDR. OF RECTANGLE DEFINITION
00804E: ToolCall $5304 ; FrameRect
380
QuickDraw and the Event Manager

0080CE: 20 8B 81 77 NEXT1 JSR SHIFT
78
79 COLOR2 PushWord #$A; COLOR = 10 = ‘GREEN’
80 ToolCall $3704; SetSolidPenPat
81
82 PAINTRCT PushLong #RECT; ADDR. OF RECTANGLE DEFINITION
83 ToolCall $5404; PaintRect
84
0080FC: 20 8B 81 85 NEXT2 JSR SHIFT
86
87 COLOR3 PushWord #$6; COLOR = ‘ORANGE’
88 ToolCall $3704; SetSolidPenPat
89
90 PAINTOV PushLong #RECT; ADDR. OF RECTANGLE DEFINITION
91 ToolCall $5904; PaintOval
92
008112: 20 8B 81 93 NEXT3 JSR SHIFT
94
95 COLOR4 PushWord #$3; COLOR = ‘PURPLE’
96 ToolCall $3704; SetSolidPenPat
97
98 ROUNDRCT PushLong #RECT; ADDR. OF RECTANGLE DEFINITION
99 PushWord #15; WIDTH OF ROUNGING OVAL
100 PushWord #15; HEIGHT OF ROUNGING OVAL
101 ToolCall $5E04; PaintRRect
102
103 TITLE PushWord #$0000; COLOR = BLACK (0)
104 ToolCall $A204; SetBackColor
105 ; BACKGROUND FOR TEXT DRAWING
106
107 T2 PushWord #$000F; COLOR = WHITE (15)
108 ToolCall $A004; SetForeColor
109 ; SET TEXT COLOR = WHITE
110
111 T3 PushWord #20; X = 20
112 PushWord #199; Y = 199
113 ToolCall $3A04; MoveTo
114
115 T4 PushLong #MSSG; HIGH WORD OF TITLE DATA
116 ToolCall $A604; DrawCString
117
008178: AF 00 C0 00 118 RDKEY LDA L, KYBD; CHECK KEYBOARD
00817C: 29 FF 00 119 AND #$00FF; CLEAR HIGH WORD
00817F: C9 80 00 120 CMP #$0080; KEYPRESS?
008182: 90 F4 81 78 121 BCC RDKEY; NOPE
008184: 8F 10 C0 00 122 STAL STROBE; CLEAR KEYBOARD
008188: 4C B1 81 123 JMP SHUTDOWN
124
125
126
00818B: 18 127 SHIFT CLC; SHIFT RECTANGLE
00818C: AD 12 82 128 LDA V1; V1 = V1 + 30, ETC.
Chapter 18

00818F:  69  1E  00  129  ADC  #30
008192:  8D  12  82  130  STA  V1
008195:  AD  14  82  131  LDA  H1  ; CARRY NEVER SET, SO
008198:  69  1E  00  132  ADC  #30  ; NO NEED TO RE-CLR
00819B:  8D  14  82  133  STA  H1
00819E:  AD  16  82  134  LDA  V2
0081A1:  69  1E  00  135  ADC  #30
0081A4:  8D  16  82  136  STA  V2
0081A7:  AD  18  82  137  LDA  H2
0081AA:  69  1E  00  138  ADC  #30
0081AD:  8D  18  82  139  STA  HI  ; ENTIRE BOX SHIFTED
0081B0:  60  140  RTS

144  SHUTDOWN  ToolCall $0304  ; QDShutDown
146  PushWord ID2  ; GET THE SUB-ID
147  ToolCall $1102  ; DisposeAll
148  ToolCall $0302  ; MMShutDown
150  ToolCall $0301  ; Tl.ShutDown

0081E1:  22  A8  08  E1  153  QUIT  JSL  PRODOS  ; DO QUIT CALL
0081E5:  29  00  154  DA  $29  ; QUIT CALL COMMAND VALUE
0081E7:  F6  81  00  00  155  ADRl  QUITBLK  ; ADDRESS OF PARM TABLE
0081EB:  00  00  156  BRK  $00  ; SHOULD NEVER GET HERE . . .

0081ED:  48  160  RESUME  PHK
0081EE:  AB  161  PLB  ; SET OUR DATA BANK
0081EF:  18  162  CLC
0081F0:  FB  163  XCE  ; SET NATIVE MODE
0081F1:  C2  30  164  REP  $30  ; 16-BIT MODE
0081F3:  4C  B1  81  165  JMP  SHUTDOWN  ; TRY TO SHUTDOWN

0081F6:  00  00  00  00  166  **********************************************

0081FA:  00  00  170  DA  $0000  ; STD. QUIT

0081FC:  51  75  69  63  172  MSSG  ASC 'QuickDraw Demo #1',00

008200:  6B  44  72  61  173  * IMPORTANT! TEXT FOR QUICKDRAW MUST HAVE HIGH BIT CLEAR!
008204:  77  20  44  65
008208:  6D  6F  20  23
00820C:  31  00
QuickDraw and the Event Manager

Table 18-1. QuickDraw Toolset Calls

<table>
<thead>
<tr>
<th>Command Value</th>
<th>Command Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0204</td>
<td>QDStartUp</td>
<td>Starts up QuickDraw, clears screen, initializes Gra£Port to defaults.</td>
</tr>
<tr>
<td>$0304</td>
<td>QDSHutDown</td>
<td>Frees any memory used by QuickDraw, returns screen to text mode.</td>
</tr>
<tr>
<td>$0E04</td>
<td>SetColorTable</td>
<td>Rewrites a given color table with contents of another.</td>
</tr>
<tr>
<td>$1004</td>
<td>SetColorEntry</td>
<td>Sets value of a color in a specified table.</td>
</tr>
<tr>
<td>$1104</td>
<td>GetColorEntry</td>
<td>Returns color value for a given color in a given table.</td>
</tr>
<tr>
<td>$1504</td>
<td>ClearScreen</td>
<td>Clears entire screen to a given color.</td>
</tr>
<tr>
<td>$1804</td>
<td>OpenPort</td>
<td>Initializes a new part of memory as a Gra£Port.</td>
</tr>
<tr>
<td>$1A04</td>
<td>ClosePort</td>
<td>De-allocates a Gra£Port.</td>
</tr>
<tr>
<td>$1B04</td>
<td>SetPort</td>
<td>Make specified Gra£Port the current port.</td>
</tr>
<tr>
<td>$1C04</td>
<td>GetPort</td>
<td>Returns pointer to current Gra£Port.</td>
</tr>
<tr>
<td>$1D04</td>
<td>SetPortLoc</td>
<td>Set current Gra£Port’s information structure to a specified table.</td>
</tr>
<tr>
<td>$1E04</td>
<td>GetPortLoc</td>
<td>Write current Gra£Port information structure into a given table.</td>
</tr>
<tr>
<td>$1F04</td>
<td>SetPortRect</td>
<td>Set current port rectangle to specified rectangle.</td>
</tr>
<tr>
<td>$2004</td>
<td>GetPortRect</td>
<td>Writes current port’s rectangle values into a specified table.</td>
</tr>
<tr>
<td>$2304</td>
<td>SetOrigin</td>
<td>Adjusts PortRect and BoundsRect so that the origin of the current PortRect is equal to a given value (usually 0,0).</td>
</tr>
<tr>
<td>$2704</td>
<td>HidePen</td>
<td>Sets pen to no-draw. Does not refer to whether cursor is visible or not.</td>
</tr>
<tr>
<td>$2804</td>
<td>ShowPen</td>
<td>Sets pen to drawing mode.</td>
</tr>
<tr>
<td>$2904</td>
<td>GetPen</td>
<td>Returns pointer to location where current pen coordinates are stored.</td>
</tr>
<tr>
<td>$2C04</td>
<td>SetPenSize</td>
<td>Sets pen to given size.</td>
</tr>
<tr>
<td>$2E04</td>
<td>SetPenMode</td>
<td>Set drawing mode of pen (ORA, EOR, AND).</td>
</tr>
<tr>
<td>$3004</td>
<td>SetPenPat</td>
<td>Set pen to a given pattern.</td>
</tr>
<tr>
<td>$3704</td>
<td>SetSolidPenPat</td>
<td>Set pen to solid color.</td>
</tr>
</tbody>
</table>
## Chapter 18

<table>
<thead>
<tr>
<th>Command Value</th>
<th>Command Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3404</td>
<td>SetBackPat</td>
<td>Set background pattern.</td>
</tr>
<tr>
<td>$3804</td>
<td>SetSolidBackPat</td>
<td>Set background to a color.</td>
</tr>
<tr>
<td>$3A04</td>
<td>MoveTo</td>
<td>Move pen to given position.</td>
</tr>
<tr>
<td>$3B04</td>
<td>Move</td>
<td>Move pen a distance relative to current position.</td>
</tr>
<tr>
<td>$3C04</td>
<td>LineTo</td>
<td>Draw a line from current pen position to specified point.</td>
</tr>
<tr>
<td>$3D04</td>
<td>Line</td>
<td>Draw line from current pen position to a new relative position.</td>
</tr>
<tr>
<td>$5304</td>
<td>FrameRect</td>
<td>Draw boundary of a rectangle.</td>
</tr>
<tr>
<td>$5404</td>
<td>PaintRect</td>
<td>Paint interior of rectangle with current pen color or pattern.</td>
</tr>
<tr>
<td>$5504</td>
<td>EraseRect</td>
<td>Paint interior of a rectangle with background color or pattern.</td>
</tr>
<tr>
<td>$5604</td>
<td>InvertRect</td>
<td>Inverts pixels in a given rectangle.</td>
</tr>
<tr>
<td>$5704</td>
<td>FillRect</td>
<td>Fills interior of rectangle with specified pattern.</td>
</tr>
<tr>
<td>$5804</td>
<td>FrameOval</td>
<td>Draw boundary of oval.</td>
</tr>
<tr>
<td>$5904</td>
<td>PaintOval</td>
<td>Paint interior of oval.</td>
</tr>
<tr>
<td>$5D04</td>
<td>FrameRRect</td>
<td>Draw boundary of rounded rectangle.</td>
</tr>
<tr>
<td>$5E04</td>
<td>PaintRRect</td>
<td>Paint interior of rounded rectangle.</td>
</tr>
<tr>
<td>$BC04</td>
<td>FramePoly</td>
<td>Draw boundary of polygon.</td>
</tr>
<tr>
<td>$BD04</td>
<td>PaintPoly</td>
<td>Paint interior of polygon.</td>
</tr>
<tr>
<td>$C604</td>
<td>SetClipHandle</td>
<td>Set ClipRgn in GrafPort to region identified by a handle.</td>
</tr>
<tr>
<td>$C704</td>
<td>GetClipHandle</td>
<td>Get handle that identifies ClipRgn for a GrafPort.</td>
</tr>
<tr>
<td>$7904</td>
<td>FrameRgn</td>
<td>Draws boundary of a region.</td>
</tr>
<tr>
<td>$7A04</td>
<td>PaintRgn</td>
<td>Paints interior of a region with current pen color or pattern.</td>
</tr>
<tr>
<td>$7E04</td>
<td>ScrollRect</td>
<td>Shift pixels in a rectangle in any direction.</td>
</tr>
<tr>
<td>$7F04</td>
<td>PaintPixels</td>
<td>Transfers pixels from one region to another, all specified with a parameter block.</td>
</tr>
<tr>
<td>$D604</td>
<td>PPToPort</td>
<td>Transfer pixels from source image to current port, clipping as necessary.</td>
</tr>
<tr>
<td>$9404</td>
<td>SetFont</td>
<td>Set current font.</td>
</tr>
<tr>
<td>$9C04</td>
<td>SetTextMode</td>
<td>Set text drawing mode (ORA, EOR, AND).</td>
</tr>
<tr>
<td>$A004</td>
<td>SetForeColor</td>
<td>Set color of text drawing.</td>
</tr>
<tr>
<td>$A204</td>
<td>SetBackColor</td>
<td>Set color of text background.</td>
</tr>
<tr>
<td>$9E04</td>
<td>SetSpaceExtra</td>
<td>Set size of space character in pixels. Used to fill-justify text.</td>
</tr>
<tr>
<td>$D404</td>
<td>SetCharExtra</td>
<td>Set size of space between all characters.</td>
</tr>
<tr>
<td>$A404</td>
<td>DrawChar</td>
<td>Draw a character.</td>
</tr>
<tr>
<td>$A504</td>
<td>DrawString</td>
<td>Draw a string whose first byte is a length-byte.</td>
</tr>
<tr>
<td>$A604</td>
<td>DrawCString</td>
<td>Draw a string that terminates in a zero.</td>
</tr>
<tr>
<td>$A804</td>
<td>CharWidth</td>
<td>Return width in pixels of a character in the current font.</td>
</tr>
<tr>
<td>$A904</td>
<td>StringWidth</td>
<td>Return width in pixels of a string in the current font.</td>
</tr>
<tr>
<td>$AA04</td>
<td>CStringWidth</td>
<td>Return width in pixels of a string ending with a zero, in the current font.</td>
</tr>
</tbody>
</table>
QuickDraw and the Event Manager

<table>
<thead>
<tr>
<th>Command Value</th>
<th>Command Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4D04</td>
<td>SectRect</td>
<td>Calculates intersection of two rectangles and places result in specified rectangle data structure. (And you thought you'd never have to remember difference between union and intersection.)</td>
</tr>
<tr>
<td>$4E04</td>
<td>UnionRect</td>
<td>Calculates union of two rectangles and places result in specified rectangle data structure. (Union is sum of two rectangles, the intersection is only the common area shared, which may be zero.)</td>
</tr>
<tr>
<td>$4F04</td>
<td>PtInRect</td>
<td>Detects whether a point is in rectangle. Saves you all those greater-than and less-than tests.</td>
</tr>
<tr>
<td>$8404</td>
<td>LocalToGlobal</td>
<td>Converts a point from the local coordinates based on the BoundsRect to the global coordinates in which 0,0 is the upper left corner of the pixel image.</td>
</tr>
<tr>
<td>$8504</td>
<td>GlobalToLocal</td>
<td>Converts a point from global coordinates to local.</td>
</tr>
<tr>
<td>$8604</td>
<td>SetCursor</td>
<td>Set cursor image to a specified pixel image.</td>
</tr>
<tr>
<td>$9004</td>
<td>HideCursor</td>
<td>Make cursor invisible.</td>
</tr>
<tr>
<td>$9104</td>
<td>ShowCursor</td>
<td>Make cursor visible.</td>
</tr>
<tr>
<td>$9204</td>
<td>ObscureCursor</td>
<td>Hide cursor until next mouse movement.</td>
</tr>
<tr>
<td>$CA04</td>
<td>InitCursor</td>
<td>Reinitializes cursor to visible arrow.</td>
</tr>
<tr>
<td>$8604</td>
<td>Random</td>
<td>Returns random number so you can write those demos with lines shooting everywhere.</td>
</tr>
<tr>
<td>$8804</td>
<td>GetPixel</td>
<td>Returns value of pixel at specified point. (For all you who still miss the SCRN function from lo-res graphics that was never included in Applesoft BASIC hi-res.)</td>
</tr>
</tbody>
</table>

Table 18-2. Possible QuickDraw Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0401</td>
<td>AlreadyInitialized QuickDraw has already been started up. QD can't be started if it's already active.</td>
</tr>
<tr>
<td>$0410</td>
<td>ScreenReserved</td>
</tr>
<tr>
<td>$0411</td>
<td>BadRect</td>
</tr>
<tr>
<td>$0450</td>
<td>BadTable</td>
</tr>
<tr>
<td>$0451</td>
<td>BadColorNum</td>
</tr>
<tr>
<td>$0420</td>
<td>NotEqualChunkiness</td>
</tr>
</tbody>
</table>

The Event Manager

So far, our programs have just looked at $C000 whenever they needed a keypress, but this really isn't very appropriate for an application program. What we need is a way to monitor all the possible input from the user, and to easily
respond to keypresses, mouse clicks, special option key modifiers, and other input.

In the Apple IIgs, this is handled by the Event Manager, who's job is to provide the application program with an orderly presentation of events as they occur. An event is most easily defined as a transition from one state to another. In practice, an event is usually user-initiated and consists of a key or mouse button changing from up to down, or some other user input. In addition, the Event Manager gives you an automatic buffering of events, so that if your program happens to be busy doing something when a key is pressed, or a mouse clicked, the event won't be lost. It maintains this list of stored events in what is called the Event Queue.

Events are delivered by the Event Manager into a data structure called the Event Record, which you define in your program. By looking at the event record, you can determine which of a number of possible events have occurred. The event record itself looks like this:

```
EVENTREC        ; EVENT RECORD DATA STRUCTURE
EVENT          DA $0000     ; EVENT CODE
TYPE           ADDR $0000   ; TYPE OF EVENT (4 BYTES)
TIME           ADDR $0000   ; TIME (4 BYTES)
YPOS           DA $0000     ; Y POSN OF MOUSE
XPOS           DA $0000     ; X POSN OF MOUSE
MOD            DA $0000     ; EVENT MODIFIER
```

When an event occurs, a number code for the event is stored in EVENT, along with the position of the mouse at that instant, a time in tics (a tic is 1/60 second since the computer was started up), and additional information in the form of the TYPE and MOD bytes. Table 18-3 lists the possible event code.

When an event is detected, the application can then look at the TYPE field to obtain additional information. For example, if it is a key-down or auto-key event, the TYPE field will contain the ASCII value for the key pressed in the first byte of the TYPE field. In the case of an activate or update event, TYPE (also called MSSG) contains a handle for the window that needs to be updated or activated.

The modifier field contains additional information about the event. For keypress events, this is whether additional keys such as the shift, control, or option keys were also pressed. You can also determine whether the Caps Lock key is up or down and whether a number key is coming from the keypad or the main keyboard.
Table 18-3. Possible Event codes

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Null event</td>
</tr>
<tr>
<td>1</td>
<td>Mouse-down event</td>
</tr>
<tr>
<td>2</td>
<td>Mouse-up event</td>
</tr>
<tr>
<td>3</td>
<td>Key-down event</td>
</tr>
<tr>
<td>4</td>
<td>Undefined</td>
</tr>
<tr>
<td>5</td>
<td>Auto-key event</td>
</tr>
<tr>
<td>6</td>
<td>Update event (for windows)</td>
</tr>
<tr>
<td>7</td>
<td>Undefined</td>
</tr>
<tr>
<td>8</td>
<td>Activate event (for windows)</td>
</tr>
<tr>
<td>9</td>
<td>Switch event</td>
</tr>
<tr>
<td>10</td>
<td>Desk accessory event</td>
</tr>
<tr>
<td>11</td>
<td>Device driver event</td>
</tr>
<tr>
<td>12</td>
<td>Application-defined event</td>
</tr>
<tr>
<td>13</td>
<td>Application-defined event</td>
</tr>
<tr>
<td>14</td>
<td>Application-defined event</td>
</tr>
<tr>
<td>15</td>
<td>Application-defined event</td>
</tr>
</tbody>
</table>

Each modifier is indicated by setting a bit in the modifier word (2 bytes) as shown in Table 18-4.

Table 18-4. Event Code Bit Modifiers

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Active Flag (for windows)</td>
</tr>
<tr>
<td>1</td>
<td>Change Flag (for windows)</td>
</tr>
<tr>
<td>2–5</td>
<td>Unused</td>
</tr>
<tr>
<td>6</td>
<td>Button1 up</td>
</tr>
<tr>
<td>7</td>
<td>Button0 up</td>
</tr>
<tr>
<td>8</td>
<td>Apple (command) key</td>
</tr>
<tr>
<td>9</td>
<td>Shift key</td>
</tr>
<tr>
<td>10</td>
<td>Caps Lock key</td>
</tr>
<tr>
<td>11</td>
<td>Option key</td>
</tr>
<tr>
<td>12</td>
<td>Control key</td>
</tr>
<tr>
<td>13</td>
<td>KeyPad</td>
</tr>
<tr>
<td>14–15</td>
<td>Unused</td>
</tr>
</tbody>
</table>

Events are usually detected in the application by calling GetNextEvent ($0A06). When calling this, you can (and in fact must) include a mask, with appropriate bits set for what types of events you want to accept at that moment. The bits in the mask are set as shown in Table 18-5.
Table 18-5. Event Mask

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Unused</td>
</tr>
<tr>
<td>1</td>
<td>Mouse down</td>
</tr>
<tr>
<td>2</td>
<td>Mouse up</td>
</tr>
<tr>
<td>3</td>
<td>Key down</td>
</tr>
<tr>
<td>4</td>
<td>Unused</td>
</tr>
<tr>
<td>5</td>
<td>Auto-key</td>
</tr>
<tr>
<td>6</td>
<td>Update (for windows)</td>
</tr>
<tr>
<td>7</td>
<td>Unused</td>
</tr>
<tr>
<td>8</td>
<td>Activate (for windows)</td>
</tr>
<tr>
<td>9</td>
<td>Switch</td>
</tr>
<tr>
<td>10</td>
<td>Desk accessory</td>
</tr>
<tr>
<td>11</td>
<td>Device driver</td>
</tr>
<tr>
<td>12-15</td>
<td>Application defined</td>
</tr>
</tbody>
</table>

For example, if you only want to deal with key-down events, you can set a mask of $0008 (%0000000000001000). If you want to include mouse events and keys being held down continuously (auto-key events), you can use a mask of $002E (%0000000000101l10).

The other nice thing about the event mask is that you can retrieve certain events even though they may have occurred after others. For example, suppose there were a key-down and a mouse-down event waiting in the Event Manager for you. If your mask requested only mouse-down events, you would get the mouse-down event, and the key-down event would remain in the queue until you requested it.

In practice, because you can also just ignore events once the code has been passed to you, an event mask of $FFFF works just fine. The event queue (event buffer) holds a finite number of events, and if you don't pull them off, sooner or later the queue could fill up with the events you haven't dealt with.

The call diagram for GetNext Event is shown in Figure 18-6.

Figure 18-6. GetNext Event ($0A06)

Stack Before Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
<th>Word: Allow space for result.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space for Result</td>
<td>Word: Which types of events are of interest.</td>
</tr>
<tr>
<td>EventMask</td>
<td>Long: Pointer to the event record data structure.</td>
</tr>
<tr>
<td>EventPointer</td>
<td>→SP:</td>
</tr>
</tbody>
</table>

388
Event-Driven Programs

Part of the underlying idea of the Event Manager and other Apple IIGS tools is the idea of an event-driven program. What this means is that the program is designed around a central loop that continually looks for an event, namely a user input. It then executes a routine related to that event, and returns as soon as possible back to the main loop. This is in contrast to a modal program where the user's command puts the program into a certain mode. In that mode, the commands may be completely different than at the main level, and commands available at the main level may no longer be available.

In an event-driven program, the goal is to create an environment where there are no modes, and the user is free to execute any logical command at any time.

As an example of a modeless, event-driven program, albeit a small one, here is a program that continually checks for an event and prints a message, depending on the event.

Program 18-2 begins very much like the QuickDraw demo program, and in fact, you may wish to use a copy of that source listing as a starting point for this program to cut down on the number of lines you'll have to type in. As usual, a checksum is provided at the end of the listing to assist you in verifying that your entered listing is identical to the printed source.

The first difference in this program is that the \texttt{GETDP} routine now obtains four pages ($400) of direct-page space. This is because the Event Manager, like QuickDraw, requires a block of direct-page memory for itself. The easiest way to handle this is to just tack its space onto the end of that assigned to QuickDraw.

In this particular program, the Event Manager is started up before QuickDraw, but it really doesn't matter which one is started first. The Event Manager is started up with the call \texttt{EMStartUp}, whose input parameters include the address of the direct page it is to use, the size of the event queue—how many events it will buffer—and the minimum and maximum clamping values for the mouse. In addition, the Event Manager requires a UserID for the application starting it up. Figure 18-7 is the call diagram for \texttt{EMStartUp}.
Figure 18-7. EMStartUp ($0206)

Stack Before Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
<th>Word: Starting address in bank 0 for work area.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DirectPageAddr</td>
<td>Word: Maximum number of events in queue.</td>
</tr>
<tr>
<td>QueueSize</td>
<td>(0=default 20).</td>
</tr>
<tr>
<td>XMinClamp</td>
<td>Word: Minimum mouse X value.</td>
</tr>
<tr>
<td>XMaxClamp</td>
<td>Word: Maximum mouse X value.</td>
</tr>
<tr>
<td>YMinClamp</td>
<td>Word: Minimum mouse Y value.</td>
</tr>
<tr>
<td>YMaxClamp</td>
<td>Word: Maximum mouse Y value.</td>
</tr>
<tr>
<td>UserID</td>
<td>Word: ID Event Mgr. will use to get memory.</td>
</tr>
</tbody>
</table>

Stack After Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
<th>←SP:</th>
</tr>
</thead>
</table>

A Closer Look

Lines 64–71 start up the Event Manager with the default values, and clamping set to 0–320 for X, and 0–200 for Y, corresponding to the size of the super hi-res display screen. As before, the sub-ID for the program is used so that the memory may be disposed of during the tool shutdown phase of the program.

QuickDraw is started up in the same way as in the QuickDraw demo program, but this time the messages on the screen will be printed a little differently. Because there are six different messages printed on the screen in various places, this program includes a generalized PRINT and CR (for Carriage Return) routine for printing text. Text is printed by doing a JSR PRINT, and following the JSR PRINT with the string you want printed, which should include a length byte at the beginning (use the STR pseudo-op).

The PRINT routine itself is a variation on Program 11-1, the Stack Indirect Indexed sample program in Chapter 11. The routine uses the stack indirect indexed addressing mode in the instruction LDA 1, S to load the return address from the JSR to the print routine. This value will be one byte less than the beginning of the string data, whose first byte is the length of the string to print.

The PRINT routine then saves this address in a pointer (MSSGPTR) that will be used to set up the DrawString command.
Before doing that, however, it adjusts the return address on the stack to resume program execution immediately after the string data following the JSR PRINT that called it.

After printing a string, it then uses the QuickDraw routine StringWidth, which returns the width, in pixels of the string. This is used to advance CH to the right so that any successive PRINT calls will continue on the same line.

The PRINT routine is first called on lines 97–104, where the message at the bottom of the screen is printed. The routine CR simple moves the print position down 10 pixels, and returns CH to zero, which is equivalent to printing a carriage return in a text-based program.

That was just the setup. The working part of the program is the main loop on lines 112 to 128. Starting at MAIN, a call is done to GetNextEvent, with the mask of $FFFF to allow all events. Each time through the loop, the subroutine MOUSE is called to continually print the coordinates of the cursor as the mouse is moved around.

MOUSE (lines 179–207) uses an interesting Toolbox routine, namely Int2Dec from the Integer Math tool set, which converts a number value into a string. Int2Dec converts a two-byte value into an ASCII string in decimal. Later, we’ll use another Integer Math tool, Int2Hex, which creates an ASCII string in hexadecimal notation. The Integer Math tool set does not need to be specifically started up or shut down, so these calls are omitted from the program.

Int2Dec requires that the address of the buffer into which the string characters will be written be provided. In this case, the buffer area is part of the messages about to be printed on lines 199–205.

For error-trapping purposes, Int2Dec also requires that the length of the output string, and whether the number is a signed number (>$7FFF is a negative number), be input as well.

When entering the listing, be sure to include the extra two spaces after the 0000 characters. These are included to automatically erase text left on the screen from previous print messages as the number goes from a large value to a smaller one with fewer digits. If you’re not sure what this does, try the program without the spaces at the end to see what I mean.

When MOUSE was called in the main loop, the stack held the event flag from GetNextEvent. TEST pulls this off the stack and checks to see if an event has occurred. This value will be zero if no event has occurred, and the program will loop back to MAIN with the BEQ MAIN instruction.

If the Accumulator is nonzero, an event has occurred, and the event code is loaded into the Accumulator from within the Event Record (EVRECORD). If it’s equal to 3, it was a keydown event, and control is passes to the routine KEYDN (for KeyDown). If any other event occurs, a JSR is done to the to the
routine that prints out the identity of the event, **EVMSSG**.

KEYDN loads the first word of TYPE from the Event Record to see specifically if the Q key was pressed, for a quit. If not, it too passes control to EVMSSG.

EVMSSG is a subroutine that prints out a description of the event that occurred, the value of TYPE, and the ASCII character associated with TYPE. After first setting, the next PRINT coordinates to 0, 50, EVMSSG prints the phrase Event: . For each possible event code from 0 to 15, a message has been set up in a table of text labeled EVENTMSSG (lines 323-340). Each descriptor (including the length byte) is 16 bytes long.

By multiplying the event code by 16, the offset into the message table can be determined. The **Multiply** routine from the Integer Math tool set is used. This routine takes as its input two words (2 bytes each) for each value to be multiplied, and returns a long (4-byte) result on the stack. For our values, the lower two bytes of the result will be sufficient to contain the offset value.

After printing the event message, the TYP portion of the program converts the value of the TYPE field of the event record into a hex value, and also embeds the corresponding character into the string to be printed. Lines 249–253 show a technique of splitting up a string to make labeling pieces of it more convenient. However, this could have been done just as easily with lines 239 and 245 using an offset into a string defined with a simple STR, as was done in the MOUSE routine.

Be sure to look at how the main loop portion of the program continually processes events. In a larger program, an indirect jump table can be created to handle each type of event. However, virtually every Apple IIGS program should be designed to have a central continuous loop, as opposed to modal subsections.

When you run this demonstration program, try pressing Open Apple-Escape to go to the Classic Desk Accessory Menu on the Apple IIGS. You’ll notice that, when you return, the program picks up the desk accessory event.

As an additional challenge, you might want to try expanding the program to also print out the appropriate message for the Modifier byte. As a hint in that direction, try constructing the modifier message table the same way as the event message table, but use a loop the does 16 LSRs on the Modifier byte. On each pass through the loop, check to see whether the carry was set or clear, and print the appropriate message. A starting pointer, like MSSGPTR should be set up to point to the beginning of the table, and 16 should be added each time through the loop.
Program 18-2. Event Manager Demo

`************ EVENT MANAGER DEMO ************
* MERLIN ASSEMBLER *

008000: 48
008001: AB

008002: A9 00 00
008005: 85 66

008007: E2 30
008009: A9 5C
00800B: 8F F8 03 00
00800F: C2 30
008011: A9 AE 81
008014: 8F F9 03 00
008018: A9 00 00
00801B: 8F FB 03 00

00803C: 09 00 01

--------

MX %00 ; TELL MERLIN WE'RE IN 16 BITS
REL
DSK EM.DEMO.L

LST OFF ; DON'T PRINT MACRO LISTING
USE UTIL.MACS ; USE MACRO LIBRARY
LST ON ; LISTING BACK "ON"
EXP OFF ; DON'T EXPAND MACROS
TR ON ; DON'T PRINT ALL BYTES

= E100A8
PRODOS EQU $E100A8 ; STD. PRODOS 16 ENTRY

= 0000
PTR EQU $00 ; OUR OWN DIRECT-PAGE PTR

= 0004
MSGPTR EQU $04 ; POINTER TO ANY MESSAGE

--------

* STARTUP THE ENVIRONMENT

--------

008009: 4B
008001: AB

008002: A9 00 00
008005: 85 66

008007: E2 30
008009: A9 5C
00800B: 8F F8 03 00
00800F: C2 30
008011: A9 AE 81
008014: 8F F9 03 00
008018: A9 00 00
00801B: 8F FB 03 00

00803C: 09 00 01

--------

STARTUP PHK
PLB

LDA #ASTARTUP ; GET OUR DATA BANK
STA MSGPTR+2 ; HIGH WORD IS OUR DATA BANK

SETRES SEP $30 ; 8-BIT MODE
LDA #$5C ; JML (JMP LONG)
STAL $3F8 ; CTRL VECOR
REP $30 ; 16-BIT MODE
LDA #RESUME
STAL $3F9 ; $3F9,3FA
LDA #*RESUME
STAL $3FB ; $3FB,3FC

TL ToolCall $0201 ; TLStartUp
MM PushWord #$0000 ; SPACE FOR RESULT
ToolCall $0202 ; MMStartUp
PullWord ID ; PULL ID AND SAVE
ORA #$0100 ; SET SUB-ID

--------

393
Chapter 18

00803F: 8D 17 83 50 STA ID2
51 GETDP
52 PushLong #$0000 ; SPACE FOR RESULT
53 PushLong #$400 ; AMT OF MEMORY NEEDED
54 ; 3 PAGES FOR QUICKDRAW
55 ; 1 PAGE FOR EVENT MGR.
56 PushWord ID2 ; ID FOR OUR APPLICATION
57 PushWord #$C001 ; TYPE: LOCKED, FIXED
58 PushLong #$0000 ; BANK = $00
59 ToolCall $0902 ; NewHandle
60 PullLong PTR ; GET HANDLE & DEREFERNCE
61 LDA [PTR] ; LONG INDIRECT LOAD
62 STA DP ; SAVE THE DP ADDRESS
63
64 EM PHA ; PUSH DP ADDRESS (IN ACC.)
65 PushWord #$0000 ; QUEUE SIZE = DEFAULT = 20
66 PushWord #$0000 ; MIN X CLAMP FOR MOUSE = 0
67 PushWord #$320 ; MAX X CLAMP = 320
68 PushWord #$0000 ; MIN Y CLAMP = 0
69 PushWord #$200 ; MAX Y CLAMP = 200
70 PushWord ID2 ; PUSH OUR SUB-ID
71 ToolCall $0206 ; EMStartUp
72
73 QD LDA DP ; GET STARTING DP MEMORY ADDR.
74 CLC ; ADD $100 FOR WHAT EM JUST USED
75 ADC #$100
76 STA DP ; PUT THINGS BACK
77
78 PHA ; PUSH DP ADDRESS ON STACK
79 PushWord #$0000 ; MASTER SCB = DEFAULT (320)
80 PushWord #$0000 ; MAX SCREEN SIZE FOR BOUNDSRECT
81 PushWord ID2 ; OUR SUB-ID
82 ToolCall $0204 ; QDStartUp
83
84 TEXT PushWord #$0000 ; COLOR = BLACK (0)
85 ToolCall $A204 ; SetBackColor (FOR TEXT)
86 ; (DEFAULT WAS WHITE)
87
88 PushWord #$000F ; COLOR = WHITE (15)
89 ToolCall $A004 ; SetForeColor (FOR TEXT)
90 ; (DEFAULT WAS BLACK)
91
92 TITLE LDA #20 ; X = 20
93 STA CH
94 LDA #189 ; Y = 189
95 STA CV
96
97 JSR PRINT
98 MSSG1 STR ‘Event Manager Demo #1’
99 JSR CR ; SIMULATE A RETURN
100
101 LDA #20 ; X = 20

394
QuickDraw and the Event Manager

0080F7: 8D 1B 83 102 STA CH
0080FA: 20 B1 82 103 JSR PRINT
0080FD: 1B 50 72 65 104 MSG2 STR 'Press keys ; use “Q” to Quit'
105
106 SHOWCURS ToolCall $9104 ; SHOW CURSOR COMMAND
107
108 **********************************************
109 * MAIN EVENT LOOP
110 **********************************************
111
112 MAIN   PushWord #$0000 ; SPACE FOR RESULT
113 PushWord #$FFFF ; MASK = USE ALL EVENTS
114 PushLong #$EVRRECORD ; RECORD ADDRESS
115 ToolCall $0A06 ; GetNextEvent
116 ; EVENT ON STACK HERE . . .
117
118 JSR MOUSE ; CONTINUALLY SHOW MOUSE POSN
119
00813B: 20 BD 81 120 TEST PLA ; GET EVENT TO LOOK AT
00813E: 68 121 BEQ MAIN ; NOTHING HERE . . .
00813F: F0 E3 =8124 122
008141: AD 1F 83 123 :1 LDA EVENT ; GET EVENT CODE
008144: C9 03 00 124 CMP #$3 ; KEYDOWN?
008147: F0 06 =814F 125 BEQ KEYDN
126
008149: 20 22 82 127 :2 JSR EVMSSG ; PRINT EVENT MESSAGE
00814C: 4C 24 81 128 JMP MAIN ; BACK TO THE LOOP
129
130 **********************************************
131
00814F: AD 21 83 132 KEYDN LDA TYPE ; KEYCODE IF ANY
008152: 29 DF 00 133 AND #$DF ; CONVERT TO UC IF NEEDED
008155: C9 51 00 134 CMP '#'Q' ; ESCAPE KEY (HI BIT CLR)?
008158: D0 03 =815D 135 BNE :1 ; NOPE
00815A: 4C 63 81 136 JMP SHUTDOWN
137
00815D: 20 22 82 138 :1 JSR EVMSSG ; PRINT THE EVENT MESSAGE
139
008160: 4C 24 81 140 :2 JMP MAIN ; BACK FOR MORE
141
142 **********************************************
143
008164: SHUTDOWN ToolCall $0304 ; QDShutDown
145
146 ToolCall $0306 ; EMShutDown
147
148 PushWord ID2 ; SUBID
149 ToolCall $1102 ; DisposeAll
150
151 PushWord ID ; OUR APPLICATION'S ID
152 ToolCall $0302 ; MMShutDown
153
Chapter 18

154 ToolCall $0301 ; TLShutDown
155
156 **********************************************
157
0081A2: 22 A8 00 E1 158 QUIT JSL PRODOS ; DO QUIT CALL
0081A6: 29 00 159 DA $29 ; QUIT CALL COMMAND VALUE
0081A8: B7 81 00 00 160 ADRl QUITBLK ; ADDRESS OF PARM TABLE
0081AC: 00 00 161 BRK $00 ; SHOULD NEVER GET HERE ...
162
163 **********************************************
164
0081AE: 4B 165 RESUME PHK
0081AF: AB 166 PLB ; SET OUR DATA BANK
008180: 18 167 CLC
008181: FB 168 XCE ; SET NATIVE MODE
008182: C2 30 169 REP $30 ; 16-BIT MODE
008184: 4C 63 81 170 JMP SHUTDOWN ; TRY TO SHUTDOWN
171
172
173
0081B7: 00 00 00 00 174 QUITBLK ADRl $0000 ; NO PATHNAME
0081BB: 00 00 175 DA $0000 ; STD. QUIT
008177
008178
008179 MOUSE ; PRINT MOUSE POSITION
180
181 XPOSN PushWord XPOS ; GET X POSITION
182 PushLong #$MSG+5 ; ADDR. OF BUFFER
183 PushWord #4 ; 4 CHAR OUTPUT
184 PushWord #$0001 ; SIGNED NUMBER FLAG
185 ToolCall $260B ; Int2Dec
186 ; CONVERT TO ASCII DECIMAL STR$
187
188 YPOSN PushWord YPOS ; GET Y POSITION
189 PushLong #$MSG+5 ; ADDR. OF BUFFER
190 PushWord #4 ; 4 CHAR OUTPUT
191 PushWord #$0001 ; SIGNED NUMBER FLAG
192 ToolCall $260B ; Int2Dec
193
0081F3: A9 00 00 194 LDA #0 ; X = 0
0081F6: 8D 1B 83 195 STA CH
0081F9: A9 0A 00 196 LDA #10 ; Y = 10
0081FC: 8D 1D 83 197 STA CV
198
0081FF: 20 B1 82 199 JSR PRINT
008202: 0A 58 20 3D 200 XMSG STR 'X = 0000'
00820D: 20 04 83 201 JSR CR
202
008210: 20 B1 82 203 JSR PRINT
008213: 0A 59 20 3D 204 YMSG STR 'Y = 0000'

396
QuickDraw and the Event Manager

00821E: 20 04 83  JSR CR
008221: 60  RTS

**********************************************
* PRINT THE EVENT MESSAGE
**********************************************

008222: A9 00 00  EVMSSG LDA #0 ; X = 0
008225: 8D 1B 83  STA CH
008228: A9 32 00  LDA #50 ; Y = 50
00822B: 8D 1D 83  STA CV
00822E: 20 Bl 82  JSR PRINT
008231: OA 45 76 65  EMSSG STR 'Event: '
008236: 18  CLC ; ALREADY CLEAR, BUT IT LOOKS BETTER
008237: 68  PLA ; PULL RESULT LOW WORD
008238: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
00823B: 85 04  STA MSSGPTR ; STORE IT IN OUR TEMPORARY WC.
00823C: 68  PLA ; PULL HIGH WORD OF RESULT &
00823D: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
00823F: 85 04  STA MSSGPTR ; MSSG ADDRESS
008240: 68  PLA ; PULL RESULT LOW WORD
008241: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
008243: 85 04  STA MSSGPTR ; MSSG ADDRESS
008244: 68  PLA ; PULL RESULT LOW WORD
008245: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
008246: 85 04  STA MSSGPTR ; MSSG ADDRESS
008247: 68  PLA ; PULL RESULT LOW WORD
008248: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
008249: 85 04  STA MSSGPTR ; MSSG ADDRESS
00824A: 68  PLA ; PULL RESULT LOW WORD
00824B: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
00824C: 85 04  STA MSSGPTR ; MSSG ADDRESS
00824D: 68  PLA ; PULL RESULT LOW WORD
00824E: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
00824F: 85 04  STA MSSGPTR ; MSSG ADDRESS
008250: 68  PLA ; PULL RESULT LOW WORD
008251: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
008252: 85 04  STA MSSGPTR ; MSSG ADDRESS
008253: 68  PLA ; PULL RESULT LOW WORD
008254: 18  CLC ; ALREADY CLEAR, BUT IT LOOKS BETTER
008255: 68  PLA ; PULL RESULT LOW WORD
008256: 69 2F 83  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
008259: 85 04  STA MSSGPTR ; STORE IT IN OUR TEMPORARY LOC.
00825A: 68  PLA ; PULL RESULT LOW WORD
00825B: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
00825C: 85 04  STA MSSGPTR ; MSSG ADDRESS
00825D: 68  PLA ; PULL RESULT LOW WORD
00825E: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
00825F: 85 04  STA MSSGPTR ; MSSG ADDRESS
008260: 68  PLA ; PULL RESULT LOW WORD
008261: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
008262: 85 04  STA MSSGPTR ; MSSG ADDRESS
008263: 68  PLA ; PULL RESULT LOW WORD
008264: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
008265: 85 04  STA MSSGPTR ; MSSG ADDRESS
008266: 68  PLA ; PULL RESULT LOW WORD
008267: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
008268: 85 04  STA MSSGPTR ; MSSG ADDRESS
008269: 68  PLA ; PULL RESULT LOW WORD
00826A: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
00826B: 85 04  STA MSSGPTR ; MSSG ADDRESS
00826C: 68  PLA ; PULL RESULT LOW WORD
00826D: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
00826E: 85 04  STA MSSGPTR ; MSSG ADDRESS
00826F: 68  PLA ; PULL RESULT LOW WORD
008270: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
008271: 85 04  STA MSSGPTR ; MSSG ADDRESS
008272: 68  PLA ; PULL RESULT LOW WORD
008273: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
008274: 85 04  STA MSSGPTR ; MSSG ADDRESS
008275: 68  PLA ; PULL RESULT LOW WORD
008276: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
008277: 85 04  STA MSSGPTR ; MSSG ADDRESS
008278: 68  PLA ; PULL RESULT LOW WORD
008279: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
00827A: 85 04  STA MSSGPTR ; MSSG ADDRESS
00827B: 68  PLA ; PULL RESULT LOW WORD
00827C: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
00827D: 85 04  STA MSSGPTR ; MSSG ADDRESS
00827E: 68  PLA ; PULL RESULT LOW WORD
00827F: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
008280: 85 04  STA MSSGPTR ; MSSG ADDRESS
008281: 68  PLA ; PULL RESULT LOW WORD
008282: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
008283: 85 04  STA MSSGPTR ; MSSG ADDRESS
008284: 68  PLA ; PULL RESULT LOW WORD
008285: 69 83 2F  ADC #EVENTMSSG ; ADDR. OF MSSG DATA
008286: 85 04  STA MSSGPTR ; MSSG ADDRESS
008287: 68  PLA ; PULL RESULT LOW WORD
008288: E2 30  SEP $30 ; 8-BIT ACC.
00828A: AD 21 83  LDA TYPE
00828D: 8D AA 82  STA CHAR
008290: C2 30  REP $30 ; BACK TO 16 BITS
008292: 20 B1 82  JSR PRINT
008295: 17  TMSSG DFB TMSGEND-TMSGG-1
008296: 54 79 70 65  ASC 'Type: '
008297: 24 30 30 30  HEXSTR ASC '$0000 = ' ; BUFFER FOR Int2Hex
008298: 24 30 CHAR ASC 'X'
008299: 253 TMSGEND
Chapter 18

0082AD: 20 04 83 254 JSR CR
255
0082B0: 60 256 RTS ; GO BACK FOR MORE...
257
258
259**********************************************
260
261PRINT PushWord CH ; HORIZ. CURSOR POSN
262PushWord CV ; VERT. CURSOR POSN
263ToolCall $3A04 ; MoveTo
264
265; ADDR. OF MSSG-1 ON STACK
266; FROM ‘JSR’ INSTR.
267LDA 1,S
268STA MSSGPTR ; (MSSGPTR) = MSSG ADDRESS
269LDY #$00 ; 1ST CHAR OF MSSG DATA (LEN)
270LDA (MSSGPTR),Y ; GET LEN OF STRING TO PRINT
271AND #$00FF ; CLEAR HIGH BYTE OF ACC.
272SEC ; TRICK TO ADD +1 TO RESULT
273ADC 1,S ; ADD TO LOW WORD OF RTS ADDRESS
274STA 1,S ; REWRITE RTS TO AFTER MSSG TEXT
275
276PushLong MSSGPTR ; ADDR. OF MSSG TEXT
277ToolCall $A504 ; DrawString
278
279PushWord #$0000 ; SPACE FOR RESULT
280PushLong MSSGPTR ; ADDR. OF MSSG
281ToolCall $A904 ; StringWidth
282
283CLC
284PLA ; GET WIDTH OF STRING
285ADC CH ; MOVE CURSOR TO RIGHT
286STA CH
287
288RTS ; RETURN TO CODE ‘AFTER’ MSSG TEXT
289
290
291
292CR CLC
293LDA CV
294ADC #10 ; SIMULATE A CARRIAGE RETURN
295STA CV ; Y = Y + 10
296LDA #$00
297STA CH ; X = 0
298RTS
299
300
301
302ID DA $0000 ; OUR APPLICATION’S ID #
303ID2 DA $0000 ; SUBID FOR DP
304
398
QuickDraw and the Event Manager

008319: 00 00 305 DP DA $0000 ; TEMP STORAGE FOR THE DP VALUE
        306
00831B: 00 00 307 CH DA $0000 ; HORIZ. CURSOR POSITION
00831D: 00 00 308 CV DA $0000 ; VERT. CURSOR POSITION
        309
        310
        311
4E 75 6C 6C
4D 6F 75 73
4D 6F 75 73
48 65 79 20
55 6E 64 65
41 75 74 6F
55 70 64 61
55 6E 64 65

EVRECORD ; DATA BLOCK WRITTEN BY EV MGR.
313
00831F: 00 00 314 EVENT DA $0000 ; EVENT CODE
008321: 00 00 315 TYPE ADRL $0000 ; TYPE OF EVENT
008325: 00 00 316 TIME ADRL $0000 ; TIME SINCE STARTUP
008329: 00 00 317 YPOS DA $0000 ; Y POSITION OF MOUSE
00832B: 00 00 318 XPOS DA $0000 ; X POSITION OF MOUSE
00832D: 00 00 319 MOD DA $0000 ; EVENT MODIFIER
        320
        321
        322
EVRECORD ; DATA BLOCK WRITTEN BY EV MGR.
313
00831F: 00 00 314 EVENT DA $0000 ; EVENT CODE
008321: 00 00 315 TYPE ADRL $0000 ; TYPE OF EVENT
008325: 00 00 316 TIME ADRL $0000 ; TIME SINCE STARTUP
008329: 00 00 317 YPOS DA $0000 ; Y POSITION OF MOUSE
00832B: 00 00 318 XPOS DA $0000 ; X POSITION OF MOUSE
00832D: 00 00 319 MOD DA $0000 ; EVENT MODIFIER
        320
        321
        322
        323 EVRECORD ; DATA BLOCK WRITTEN BY EV MGR.
4E 75 6C 6C
4D 6F 75 73
4D 6F 75 73
48 65 79 20
55 6E 64 65
41 75 74 6F
55 70 64 61
55 6E 64 65

EVENTMSS ; TABLE OF EVENT DESCRIPTORS
324
00832F: 4E 75 6C 6C 325 E0 ASC 'Null Event ',00 ; EACH ENTRY = 16 CHARS!
00833F: 4D 6F 75 73 326 E1 ASC 'Mouse Down ',00
00834F: 4D 6F 75 73 327 E2 ASC 'Mouse Up ',00
00835F: 4B 65 79 20 328 E3 ASC 'Key Down ',00
00836F: 55 6E 64 65 329 E4 ASC 'Undefined ',00
00837F: 41 75 74 6F 330 E5 ASC 'Auto-Key ',00
00838F: 55 70 64 61 331 E6 ASC 'Update ',00
00839F: 55 6E 64 65 332 E7 ASC 'Undefined ',00
0083AF: 41 63 74 69 333 E8 ASC 'Activate ',00
0083BF: 53 77 69 74 334 E9 ASC 'Switch ',00
0083CF: 44 65 76 69 335 E10 ASC 'Desk Acc. ',00
0083DF: 44 65 76 69 336 E11 ASC 'Device Driver ',00
0083EF: 55 73 65 72 337 E12 ASC 'User1 ',00
0083FF: 55 73 65 72 338 E13 ASC 'User2 ',00
00840F: 55 73 65 72 339 E14 ASC 'User3 ',00
00841F: 55 73 65 72 340 E15 ASC 'User4 ',00
341
342 ***************************************************
343
00842F: 78 344 CHKSUM CHK ; CHECKSUM FOR VERIFICATION

--End Merlin-16 assembly, 1072 bytes, errors: 0
Chapter 19

The Window and Menu Managers
In past chapters, you’ve seen how to use QuickDraw and the Event Manager and how to use other Apple II GS tools to make your programming easier. Of course, what you really want to do is to put all those neat-looking windows and menus on the screen. This chapter will show you how to do it.

**QuickDraw and Windows**

When you started up QuickDraw, it was already operating in a window environment. A window just means that the entire drawing region will be subdivided into a smaller portion for the drawing or viewing actions at that moment. If you imagine these subregions as rectangles, it’s easy to explain that QuickDraw starts up with both the drawing space and the active area both set to the entire screen. How could it be otherwise? Just imagine that your drawing space was either bigger than the screen or than just one corner of it. At startup, the upper left corner of the screen has the coordinates 0,0; and the lower right corner (at least in the 320 mode) has 199, 319.

QuickDraw doesn’t have to use these coordinates, however. It can make the upper left or lower right corner almost any values you wish, as long as the entire screen remains in the range of \(-16384\) to \(+16384\) both horizontally and vertically. This imaginary drawing space can be represented as a field on which the screen is moved around (see Figure 19-1).

The area representing the total document, a picture, for example, is called the *BoundsRect*, for *Boundaries Rectangle*, and defines the absolute maximum area for drawing. It’s possible to set the BoundsRect to a larger or smaller space, and to even point it off into some other part of memory for drawing off-screen, but this is fairly unusual and need not concern you for the moment.

The one thing that is constant for all of the regions discussed here is that the units of measurement, or scaling units, are the same, and are defined in pixel units. That is to say, you cannot arbitrarily take the screen area of 320
pixels wide and rescale it to a new width of 100 units in the same area. All measurements within QuickDraw and the Window Manager are done in the same pixel units.

There is another rectangle that defines what part of the image (in this case, the screen) will be drawn to at a particular moment. It’s easy to see that if you wanted to create a smaller window than the entire screen, you could either have a smaller BoundsRect, or create a new rectangle that represented a subdivision of the BoundsRect.

This subdivision is called the PortRect, and it represents that actual part to be drawn to. This rectangle exists to provide for the possibility that the entire document might be larger than the screen, and that a smaller rectangle would be required to clip the display to just that part available to the screen.

When QuickDraw starts up, the PortRect and BoundsRect are both set to the full screen. When you want to use a window, the Window Manager will automatically set the PortRect to the interior of the window you’re using (called the content region), and automatically adjust it as the window is sized smaller or larger.

While drawing in the PortRect, it’s also possible to further clip the drawing within regions. Regions are different from rectangles in that they can be any shape at all. These regions are called the ClipRgn (for Clipping Region) and VisRgn (for Visible Region). The ClipRgn is a mask set up by the application to describe how to clip the image about to be drawn. For example, suppose you wanted to draw a circle with grid lines across it, like this:
Rather than having to start and end each vertical line on the circumference of the circle within some routine, it is much easier to define a circular ClipRgn, and then draw a rectangular grid over it. The ClipRgn will make it so that only the lines within the circle are drawn.

The VisRgn is used most often by the Window Manager to further clip what it has to draw when one window is laid on top of another, and only a certain part is visible. It's similar to the program's use of the ClipRgn, but generally not something you have to be aware of.

All of the parameters for drawing including the BoundsRect, PortRect, Clipping Regions, the pen color, the pen state, and more are grouped together into a total definition of the drawing state called the GrafPort (for Graphics Port). The GrafPort data structure (which is a considerable entity) defines all the variables for drawing at that instant. Still more impressive is the fact that the Apple IIGS can maintain a flexible number of individual GrafPorts all at the same time, and quickly switch between them as necessary. This is what makes the multiwindow, multiapplication environment possible on the Apple IIGS.

Generally speaking, though, you won't have to worry about any of these regions other than the PortRect when you use the Window Manager, because all the necessary clipping will take place automatically. All of these rectangles and regions so far mentioned are part of QuickDraw. The purpose of the Window Manager is to insulate you from having to deal with regions and coordinate systems, and to automate the creation and maintenance of windows in which your program will display its data and messages.

**The Window Manager: ROM vs. RAM Tool Sets**

There is one very outstanding difference between the Window Manager and the QuickDraw tool sets: The Window Manager must be loaded into memory from disk for your program to use it. QuickDraw at least starts out as a ROM-based tool, and—even though patches are made to the basic routines during the boot process for ProDOS 16—it's available to any assembly language routine that wants to call it, including ProDOS 8 system files and routines added to Applesoft BASIC.
The Window Manager, on the other hand, must be loaded from an application running under ProDOS 16, and the System Loader automatically deactivates all RAM tools when switching to ProDOS 8, thus making their use from that environment very difficult.

However, in the ProDOS 16 environment, loading any RAM-based tools your program may require is very simple. The Tool Locator tool set includes a command called LoadTools, which only requires that you provide it with a pointer to a list of the code numbers for the tool sets you wish to load. That list also includes the minimum version number for each tool that is acceptable to you. For most applications, specifying zero for the version number will work fine, as this tells the LoadTool command to load whatever version is on the disk.

Tools are stored on the disk with the names TOOL014, TOOL015, and so forth, where the number corresponds to the tool set number given in Chapter 16. The Window Manager is tool number 14, and can be loaded using LoadTools with a code segment like this:

```
TOOLS PushLong #TOOLTABL ; PUSH ADDR. OF TABLE
ToolCall $0E01 ; LoadTools
```

and somewhere in your program . . .

```
TOOLTABL DA 1 ; NUMBER OF TOOLS TO LOAD
  DA 14 ; WINDOW MGR. = #14
  DA 0 ; 0 = ANY VERSION WILL DO
```

All you do is call LoadTools and tell it where the list of the RAM tools you want to load are. RAM tools are always loaded from the SYSTEM/TOOLS directory of the startup disk. Of course, this is one spot in your program where an error is even more likely to be returned if the desired tool is not on the system disk, and so it's generally a good idea to include some code for handling such an error.

You can load more tools by changing the 1 at the beginning of TOOLTABL to equal however many tools will be in the list. For each tool, you must include the tool number and the version number. Zero is the usual default for the version number.

### Window Definitions

Like the Event Manager with its Event Record, the Window Manager makes heavy use of something, coincidentally enough, called the Window Record. The Window Record includes not only the Grafport for each window, but also information such as the various regions currently displayed in that window, whether there is a title bar at the top, whether there are scroll bars, where to go in memory to execute a window update routine when needed, and more.
When a window is created using the Window Manager tool `NewWindow`, the application receives a handle to the Window Record, which is used from then on as a unique identifier for that window.

There are many different types of windows. How you set up your Window Record will determine how windows produced will appear and operate.

A Window Record is created using `NewWindow` and a defined parameter list. The contents of that parameter list are shown here and include sample settings for a window with scroll bars, and an UPDATE routine, and can be used as a model for your own Window Records.

**Window Parameter List**

The window parameter is listed below as it would appear in an assembly listing. Notice that lowercase labels are acceptable in a source listing, and help make the longer labels more readable.

```
WPTR     ADRL $0000 ; STORAGE FOR HANDLE IDENTIFIER
WTITLE   STR 'Window Mgr. Demo #1'
WINDOW

paramlength DA windend-WINDOW ; LENGTH OF DATA BLOCK
wFrame   DA %1101111111110101 ; WINDOW FRAME DEFINITION
          ; BIT 15 = TITLE
          ; BIT 14 = CLOSE BOX
          ; BIT 13 = (NOT) AN ALERT BOX
          ; BIT 12 = VERTICAL SCROLL BAR
          ; BIT 11 = HORIZ. SCROLL BAR
          ; BIT 10 = GROW BOX
          ; BIT 9 = FIXED ORIGIN ON GROW OR ZOOM
          ; BIT 8 = ZOOMABLE
          ; BIT 7 = DRAGGABLE
          ; BIT 6 = ACTIVATE ON CONTENT
          ; BIT 5 = WINDOW IS VISIBLE
          ; BIT 4 = (NO) INFORMATION BAR
          ; BIT 3 = (NO) INDEPENDENT CTRLS
          ; BIT 2 = ALLOCATED BY NEWWINDOW
          ; BIT 1 = (NOT) CURRENTLY ZOOMED
          ; BIT 0 = HIGHLIGHTED

wTitle   ADRL WTITLE ; POINTER TO TITLE
wRefCon  ADRL $0000 ; REFERENCE CONSTANT
wZoom    DA 0,0,0,0 ; ZOOM RECTANGLE
wColor   ADRL $0000 ; COLOR TABLE
wOrigin  DA 0,0 ; ORIGIN OFFSET
wDataSiz DA 200,320 ; HEIGHT, WIDTH DATA AREA
wMaxSiz  DA 150,290 ; HEIGHT, WIDTH MAX WINDOW
wScroll  DA 4,16 ; VERT., HORIZ. SCROLL INCREMENT
```
The first two lines, WPTR and WTITLE, are not actually part of the window record, but they are kept close by for aesthetic reasons. Each window is uniquely identified by a handle to its Window Record, and WPTR is a storage location to keep the handle to a window. WTITLE is the string of text that you would like to be used as the title to the window.

The window parameter list actually begins at WINDOW, which marks the beginning of the data structure. Paramlength, the first entry in the table, is an error-checking device. A properly set up parameter list should use $1E bytes, and the Window Manager will check this number to make sure your table is properly constructed.

wFrame is a two-byte value that defines which of the many possible window elements will be used in the frame of your window. Each bit in wFrame determines whether a particular element is active. The wFrame value used in this example produces a draggable, resizable window that is typical of the windows used in many Apple II GS applications.

Each bit is assigned as follows:

**Bit 0: HiLited.** This is set to 1 when your window is highlighted. This is automatically set when the window is first started up with the Window Manager call NewWindow, so the value in the assembled table doesn’t matter. The main use is to provide the option of checking to see whether a window has been highlighted while you weren’t looking.

**Bit 1: Zoomed.** This indicates whether the window is in a zoomed or unzoomed state. Setting this in your definition tells the window manager which state to start off with. The zoom box is a control, present at the upper right corner of a window, if so specified in the Window Record. The user clicks in the zoom box to automatically resize the window, most often to the full size of the screen. If the Zoom bit (bit 8) in the wFrame word is not set, this bit is ignored.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>wPage</td>
<td>DA</td>
<td>40,160</td>
<td>VERT., HORIZ. PAGE INCREMENT</td>
</tr>
<tr>
<td>wInfoRefCon</td>
<td>ADRL</td>
<td>$0000</td>
<td>INFO BAR REFERENCE CONSTANT</td>
</tr>
<tr>
<td>wInfoHeight</td>
<td>DA</td>
<td>0</td>
<td>INFO BAR HEIGHT (NONE)</td>
</tr>
<tr>
<td>wFrameDefProc</td>
<td>ADRL</td>
<td>$0000</td>
<td>FRAME PROCEDURE ADDR. (NONE)</td>
</tr>
<tr>
<td>wInfoDefProc</td>
<td>ADRL</td>
<td>$0000</td>
<td>INFO BAR PROCEDURE ADDR (NONE)</td>
</tr>
<tr>
<td>wContDefProc</td>
<td>ADRL</td>
<td>UPDATE</td>
<td>CONTENT PROCEDURE ADDR.</td>
</tr>
<tr>
<td>wPosition</td>
<td></td>
<td></td>
<td>CONTENT REGION OF WINDOW</td>
</tr>
<tr>
<td>WV1</td>
<td>DA</td>
<td>30</td>
<td>UPPER LEFT VERT. POSITION</td>
</tr>
<tr>
<td>WH1</td>
<td>DA</td>
<td>20</td>
<td>UPPER LEFT HORIZ. POSN</td>
</tr>
<tr>
<td>WV2</td>
<td>DA</td>
<td>100</td>
<td>LOWER RIGHT VERT. POSN</td>
</tr>
<tr>
<td>WH2</td>
<td>DA</td>
<td>200</td>
<td>LOWER RIGHT HORIZ. POSN</td>
</tr>
<tr>
<td>wPlane</td>
<td>ADRL</td>
<td>-1</td>
<td>PUT WINDOW AT FRONT ($FFF. = -1)</td>
</tr>
<tr>
<td>wStorage</td>
<td>ADRL</td>
<td>$0000</td>
<td>STORAGE</td>
</tr>
<tr>
<td>windend</td>
<td></td>
<td></td>
<td>END OF DATA STRUCTURE</td>
</tr>
</tbody>
</table>
Bit 2: Allocated. This is set to indicate that the memory used for the Window Record was not allocated automatically using the NewWindow command. This must be zero if wStorage is zero (the most common situation).

Bit 3: Control Tie. This is usually set to 0 to specify that when the window is not active, its controls, such as scroll bars, will be deactivated. Otherwise the controls are independent of the window’s active status.

Bit 4: Information Bar. Some windows may include a bar below the title bar called the information bar. In a word processor this might be an ongoing display of the number of words in the document, the line and page you were on, or whatever else the designer wished to display. Leave this set to 0 for now. Set bit 4 to 1 if you want an information bar below the title bar.

Bit 5: Visible. This bit can be set to zero if you want to allocate a window without making it visible. ShowWindow can then be called later, when you want the window to appear. For most applications, however, set this bit to 0 (visible).

Bit 6: QContent. Although it hasn’t been discussed yet, there is a part of the Window Manager called TaskMaster that will handle certain operations automatically for you. Setting this bit tells TaskMaster that you want it to automatically activate a window that the user has clicked the mouse in, and then to pass the mouse-down event to your application. If this is set to 0, TaskMaster will still activate the window, but it will not pass the mouse-down event to your application. This means the user will have to double-click in the window to actually make something happen, once to activate the window, a second time for your application to see the click at all.

How you set bit 6 is a combination of personal choice in program design, and a consideration of the impact of automatically passing the event through. Try it both ways in the demonstration programs that follow, and see which you prefer.

Bit 7: Move. This determines whether the window can be moved by dragging the title bar. This setting is entirely up to you. Generally speaking, a setting of 1 for this bit is most common (window is draggable).

Bit 8: Zoom. If set to 1, the Window Manager will automatically put a zoom box in the upper right corner of the window frame. The window must have a title bar specified (bit 15 = 1) if this is to be used. Zooms can also be automatically handled by TaskMaster, so this is a nice touch.

Bit 9: Flex. This bit tells the Window Manager what to do when the window grows larger than the data you have to fill it with. If set to 1, the origin (the upper left corner) of the window area stays fixed and white space appears in the lower right corner as the window in enlarged. If the bit is clear, the white space will be padded in the upper left of the window, and the origin will
move down and to the right (that is, what you’re drawing will move with the lower right corner).

**Bit 10: Grow.** Setting this tells the Window Manager to put a grow box in the lower right corner of the window. This can only be specified if there is also a horizontal scroll bar (bit 11 = 1), a vertical scroll bar (bit 12 = 1), or both. The grow box allows the user to resize the window dynamically on the screen. This bit is set in the demonstration program.

**Bit 11: Bottom (horizontal) Scroll.** This tells the Window Manager to include the scroll bar for horizontally adjusting the content region of the window. If the window grows or is set to the maximum size, the scroll control will be deactivated automatically. Set to 1 for a horizontal scroll bar.

**Bit 12: Right (vertical) Scroll.** This tells the Window Manager to include the scroll bar for vertically adjusting the content region of the window. If the window grows or is set to the maximum size, the scroll control will be deactivated automatically. Set to 1 for a vertical scroll bar.

**Bit 13: Alert.** Used by the Window Manager for creating dialog boxes and so forth that have a double-line frame, called an alert box. For windows you define in your application, this bit should be 0.

**Bit 14: Close.** If this bit is set to 1, the Window Manager will automatically provide a close box in the upper left corner of the window. If the user clicks in the close box, this event will be passed to your application, at which point you have to handle it by closing the window, erasing the contents, or whatever other action you desire.

**Bit 15: Title.** Set this bit to display a title at the top of your window. If you do set this bit, you should include a pointer to the string to use in the wTitle part of the Window Record.

That finishes the bit definitions in wFrame. Here are the other parameters in the list:

- **wTitle** is a pointer to the location of the title of the window, presuming that the Title bit (bit 15) in wFrame has been set. The string should begin with a length byte.

- **wRefCon** (for window Reference Constant) defines a storage area that your application can store a 4-byte value in. What you store there, if anything, is entirely up to you. Some programs use these four bytes to store the handle to the Window Record (this example uses WPTR for this, instead).

- **wZoom** defines the maximum rectangle size to be used when the user clicks in the zoom box. This presumes that bit 8 has been set in the wFrame mask. Specifying a rectangle of 0,0,0,0 defaults a zoom to the entire size of the screen, minus the menu bar, if it’s there. If you start with the zoomed flag (bit 1 in wFrame) set to 1, the Window Manager uses this value to determine the
opening size of the window. If bit 0 in wFrame is clear (not zoomed), the Window Manager uses the wPosition rectangle as the starting size for the window. Once things are rolling along, clicking in the zoom box alternates between the wZoom rectangle and the previous size of the window, even if that has changed from wPosition because the user has resized the window.

For wZoom, and most other QuickDraw point and rectangle definitions, remember that the Y coordinate is given first, then the X coordinate. Thus wZoom is defined as V1,H1,V2,H2 where V1,H1 are the vertical and horizontal coordinates of the upper left corner of the window, and V2,H2 are the vertical and horizontal coordinates of the lower right corner of the window.

Although most windows look fairly standard, and use black, gray and white to draw the window, it is possible—and easy—to redefine the different parts of the window frame. wColor in the window parameter list specifies which color table you would like used in drawing the window frame. The default entry for this is $0000, which defaults to color table #0, but you can put the address of one of the other color tables or a table you’ve put somewhere else in memory here.

wOrigin determines where in the underlying coordinate system the window will be opened. Ordinarily, you would start with this offset value being 0,0, but you can make it anything you want.

Let’s suppose that you have a drawing that is 200 pixels in height and width. For sake of illustration, further suppose that the window you are going to open to display this image is only 100 pixels in height and width. This necessitates a choice on your part: You can either open the window on the upper left corner of your picture (origin = 0,0), or choose to more-or-less center the window on the document (origin = 25,25), or perhaps even the last place the person made a change to the picture (origin = variables).

wDataSiz represents that maximum size of the data the window will be scrolling over, which is necessary for the scroll bar thumbs (the movable box) to be properly scaled and positioned.

wMaxSiz lets you specify what the maximum size of the content (interior) portion of the window will be. With any kind of controls, such as the title bar or other control, the window itself will be larger than this, so remember to take this into account when calculating your largest size. You can set this to zero if growing to the maximum size of the DeskTop is acceptable to you.

wScroll sets the number of pixels you want the window to change when the arrow controls are pressed (when using a scroll bar).

wPage permits the user to click in the open space on either side of the scroll thumb to move a screen (or page) at a time.

wInfoRefCon is a storage location in which you can put a 4-byte value
Chapter 19

that will be passed to the routine that draws the contents of the information bar (see wInfoDefProc parameter description). This could be the pointer to a string, or to any other value your routines might require. (Set this to zero for most cases.)

\texttt{wInfoHeight} defines how tall, in pixels, the information bar will be. Set this to zero if you’re not using an information bar.

\texttt{wFrameDefProc} (for window Frame Definition Procedure) defines the address of a routine which will draw the entire window frame. Use a zero here unless you’re very creative. Although most windows you see all seem to be about the same, it’s possible to have your own routine create the window, instead of the Window Manager, if you wish.

\texttt{wInfoDefProc} (for window Information Bar Definition Procedure) specifies the address of the routine that will be used to fill in the information bar. If you’re not using an information bar, set this to zero.

Whenever a window is moved, opened, or made visible by another window on top of it being moved or closed, it is necessary to update (redraw) the contents of that window. Many times, this update will be required because of something completely unrelated to that window, such as the opening or moving of a desk accessory, or the activity of other windows on the DeskTop.

\texttt{wContDefProc} (for window Content Definition Procedure) defines the address in memory where the routine to draw the contents of your window is located. In the sample listing discussed earlier, this is shown as UPDATE, the label to a routine elsewhere in the program. This may be set to zero if you want to manage all the updating yourself, but it is much easier to have the Window Manager call your routine for you as needed.

\texttt{wPosition} is used to determine its initial size and position on the DeskTop when your window is first opened. This is usually specified assuming that 0,0 is the upper left of the DeskTop. This will be the same as the upper left of the screen if there is no menu bar. If a menu bar is used, 0,0 corresponds to the upper left corner of the active DeskTop area just below the menu bar.

\texttt{wPlane} allows the positioning of windows in other front to back positions. When a window is opened, you normally want it in front of any other windows on the DeskTop at that point. However, you may want to be able to position the window in other front to back positions. If you set \texttt{wPlane} = $FFFFFFFFFF (-1), the window will be opened in front of any other windows. If \texttt{wPlane} = 0, the window will be opened behind any other windows on the DeskTop. For any other position, you must specify the handle of the window behind which the new window should be placed. At that point, it is up to your application to determine and manage the handles to other windows on the DeskTop as needed.
**wStorage** is an optional address to where the Window Record will be created. Normally, this is zero, and the Window Manager, through NewWindow, sets aside about 300 bytes somewhere in memory to keep its own Window Record. However, this parameter is provided to allow for the possibility that you might want to allocate your own Window Record somewhere in memory other than that normally set up by NewWindow. This is flagged by bit 2 in wFrame being set, and by specifying the address of a Window Record area you have provided for in the wStorage field.

**Local and Global Coordinates**

You have seen how the screen is given a set of coordinates, usually in the range of 0–199 and 0–319. If you think about it, though, what you really want are your own set of coordinates within the window as it is moved around on the DeskTop. Ideally, moving the window should have no effect on your application’s attempts to draw data within the window. The difference in coordinate systems leads to the concepts of the global and local coordinate systems. When specifying where to open a window on the DeskTop, you are dealing with global coordinates, in which 0,0 (the origin) is usually in the upper left corner.

When drawing in a window whose scroll bars are in their home position (not scrolled), you want the upper left corner to be 0,0, regardless of where the window is at on the DeskTop. Within the window, your location is defined in local coordinates, local to that window. As the scroll bars are used, the Window Manager will automatically adjust the local coordinate system so that the coordinates of the upper left corner of the screen is appropriate for the current scroll position.

Obviously, 0,0 can’t be simultaneously the upper left corner of both the screen and the window. The next best thing though is to have the Window Manager switch coordinate values as needed, depending on whether you are drawing in a window or dealing with the DeskTop. When the window has 0,0 within it, the global coordinates of the upper left of the screen will temporarily become negative. However, this won’t matter to you because at that moment your window will be your frame of reference (an appropriate choice of words).

**TaskMaster**

With the tools you’re familiar with so far, managing windows would still be pretty involved. Imagine your program in the main loop calling the Event Manager. A mouse-down event comes back, and now you have to determine whether it was in a window, which window it was, and whether it was on a scroll bar, on the close box, and so on. Once you’ve determined where the event took place, you then have to be able to move the window if necessary, or
Chapter 19

resize it, or otherwise handle the possible changes to the window.

As it happens, there are specific routines, like FindWindow ($170E), that will help you determine where a mouse-down took place, and other tool calls, like SizeWindow ($1C0E), that you can use, but there is an even easier way. Part of the Window Manager includes a routine called Taskmaster.

TaskMaster is an automated function built into the Window Manager that handles things like scrolling, zooming, dragging a window, and more—all without the slightest bother to your application. When the user clicks in a window, or drags the grow box or the title bar, the appropriate action is automatically handled by TaskMaster. The window is resized, or repositioned on the DeskTop, or has done whatever else needs to be done without you having to put any equivalent routines in your own program. What this really means is that you can write a very professional-looking program in far fewer lines of code than would otherwise be required.

TaskMaster itself involves more than just the Event Manager, QuickDraw, and the Window Manager. The scroll bars in a window are called controls, and require the services of another Apple IIGS tool, the Control Manager. A control is any user-input display item including scroll bars, push buttons, check boxes, dials, and other elements in any window or dialog box. In addition, TaskMaster will also help with user input with menus, and will return to your application a number code for the menu item chosen. This, however, requires interaction with another Apple IIGS tool, the Menu Manager.

Not too surprisingly, the Menu Manager is the set of routines related to creating, displaying and manipulating pull-down menus on the Apple IIGS.

If you want to write a program that only uses simple windows without scroll bars or grow boxes, and that uses GetNextEvent for all the input, it is possible to use the Window Manager, QuickDraw, and the Event Manager by themselves. However, for any program using TaskMaster, the Control Manager must be loaded and started up if the windows will use scroll bars or other controls, and the Menu Manager must also be loaded and started up if you intend to use menus in your application.

Like the Window Manager, the Control Manager and the Menu Manager are RAM-based tools that must be loaded using the LoadTool command in the Tool Locator. All three tools can be easily loaded with a code segment something like this:

```
TOOLS  PushLong #TOOLTABL ; PUSH ADDR. OF TABLE
ToolCall $0E01 ; LoadTools
```
The Window and Menu Managers

and somewhere in your program . . .

TOOLTABL DA 3  ; NUMBER OF TOOLS TO LOAD
DA 14  ; WINDOW MGR. = #14
DA 0  ; 0 = ANY VERSION WILL DO
DA 15  ; MENU MGR. = #15
DA 0  ; 0 = ANY VERSION WILL DO
DA 16  ; CONTROL MGR. = #16
DA 0  ; 0 = ANY VERSION WILL DO

The Menu Manager

To put menus in a program without the Apple IIGS tools, you first have to allocate an area at the top of the screen and print the available menu headers there, and then you have to wait for a mouse-down event in a menu header, indicating the user wants to view a menu. You also have to create a data structure somewhere in memory that contains a list of all of the choices for each menu item, along with any additional indicator bytes, such as whether a particular menu item has a check mark by it or is disabled (printed in dimmed text). You might also want to allow the user to press a keyboard equivalent of the menu choice. Of course, that means every time a key is pressed, you’ll have to scan your complete list of menu entries to see if there is a match.

The Menu Manager is a tool set of routines that makes this process easier than having to write everything yourself. There are routines like DrawMenuBar, which automatically draws the entire menu bar, and InsertMenu, which tells the Menu Manager to add a new list of menu choices to the menu bar. There are even functions like EnableItem and DisableItem to make a menu choice disabled or not. But the best news is yet to come: TaskMaster also helps with the menu selecting process itself, and the net result is that you don’t have to look for the mouse-down in a menu, or even a keyboard equivalent. TaskMaster will handle all of this for you, and will return as an event an identifying code number for whatever, if any, menu item selected by the user.

The menu bar in an application is supported by an internal menu list that keeps track of each menu (indicated by the various headings, or menu titles) in the list, and the individual items (menu items) in each menu. The menu list can be added to or altered at any time by adding new menus across the top, or by changing individual menu items in a particular menu.

Table 19-1 lists some of the Menu Manager calls used most often.
Table 19-1. Menu Manager Tool Set Calls

<table>
<thead>
<tr>
<th>Command Value</th>
<th>Command Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$010F</td>
<td>MenuStartUp</td>
<td>Starts up Menu Manager.</td>
</tr>
<tr>
<td>$030F</td>
<td>MenuShutDown</td>
<td>Shuts down Menu Manager and frees the top of the screen, the memory allocated to menus, and so forth.</td>
</tr>
<tr>
<td>$2D0F</td>
<td>NewMenu</td>
<td>Adds a new menu to menu list.</td>
</tr>
<tr>
<td>$2E0F</td>
<td>DisposeMenu</td>
<td>Removes a menu from menu list.</td>
</tr>
<tr>
<td>$130F</td>
<td>FixMenuBar</td>
<td>Menu Mgr. recalculates width of each menu and entries for proper appearance after changes have been made.</td>
</tr>
<tr>
<td>$2A0F</td>
<td>DrawMenuBar</td>
<td>Draws current menu bar, along with menu titles.</td>
</tr>
<tr>
<td>$2C0F</td>
<td>HiliteMenu</td>
<td>Highlights, or unhighlights, the title of a specified menu. Call after TaskMaster returns a menu event to unhighlight the title.</td>
</tr>
<tr>
<td>$0D0F</td>
<td>InsertMenu</td>
<td>Inserts a new menu in menu list.</td>
</tr>
<tr>
<td>$0E0F</td>
<td>DeleteMenu</td>
<td>Deletes menu from menu list.</td>
</tr>
<tr>
<td>$0F0F</td>
<td>InsertItem</td>
<td>Inserts item in a menu.</td>
</tr>
<tr>
<td>$100F</td>
<td>DeleteItem</td>
<td>Deletes item from menu.</td>
</tr>
<tr>
<td>$300F</td>
<td>EnableItem</td>
<td>Displays menu item normally.</td>
</tr>
<tr>
<td>$310F</td>
<td>DisableItem</td>
<td>Displays menu item in dimmed text and disallows it from being selected.</td>
</tr>
<tr>
<td>$320F</td>
<td>CheckItem</td>
<td>Adds or removes check mark from an item.</td>
</tr>
</tbody>
</table>

**Menu Definition**

Defining a menu centers around creating a data structure which contains the items in a particular menu. One of the simplest possible menu definitions would look like this:

```
MENU ASC '>> Title \N1',00
ASC '--Item1 \N257',00
ASC '--Item2 \N258',00
ASC '.'
```

A menu definition is made up of successive ASCII strings, each terminated by a zero (or alternatively, a carriage return (decimal 13 = $0D)). The beginning of each string contains two more-or-less arbitrary characters that help define the menu title, each item in the menu, and the end of the items for that menu.

In the example, the characters >> signify the beginning of a menu and designate that string as the title of that menu. The next two lines each begin with the characters --. These signify the individual items in the menu, and the leading characters must be different from the title characters. The menu is terminated by using a character different than the item characters. Notice that
only one termination character is needed, and it is not necessary to terminate
that string with a zero or carriage return.

You may use any leading characters you wish as long as the item char-
acters are different from the title and terminator characters. For example, an
equally legal menu definition could look like this:

```
MENU ASC '## Title \N1',00
ASC '.. Item1 \N257',00
ASC '.. Item2 \N258',00
ASC '?'
```

As a matter of appearance, it's generally best to put a space on either
side of menu titles, and to end each menu item with a space, but this is not
required.

The title and item names themselves are delimited at the end by the
reverse slash ( \ ), after which follows the identifier number for that menu or
menu item. Each menu title must be uniquely numbered in the range of 1 to
249, and each menu item (also unique) must be in the range of 256 to 65535.
The missing numbers of 250 to 255 are reserved for special editing items,
specifically:

<table>
<thead>
<tr>
<th>Item #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>Undo</td>
</tr>
<tr>
<td>251</td>
<td>Cut</td>
</tr>
<tr>
<td>252</td>
<td>Copy</td>
</tr>
<tr>
<td>253</td>
<td>Paste</td>
</tr>
<tr>
<td>254</td>
<td>Clear</td>
</tr>
<tr>
<td>255</td>
<td>Close</td>
</tr>
</tbody>
</table>

Undo    Cancels last editing operation.
Cut     Cuts selected text and puts it on clipboard.
Copy    Copies selected text and puts it on clipboard.
Paste   Inserts text on clipboard in document.
Clear   Cuts selected text but doesn't put on clipboard.
Close   Closes active window.

By using these ID numbers for these functions, any Desk Accessory that
uses TaskMaster will automatically have access to these menu choices while its
window is active.

Although you can give the menu items almost any number value you
want, most program listings you'll see start with 256 for the first menu choice,
and number from there. This is so they can use the same indexed indirect ad-
dressing mode for the command handler discussed in Chapter 10. For example,
suppose you had the menu ID numbers 256, 257 and 258. These could all be
handled with a code segment like this:

```
LDA MITEM ; GET MENU ITEM #
SEC
SBC 256 ; ADJUST TO 0,1,2 . . .
ASL ; TIMES 2 = 0,2,4 . . .
JSR (MENTBL,X) ; USE TABLE OF ADDRESSES
JMP MAIN ; BACK TO MAIN LOOP
```
If you want to be really tricky and save a line of code, you can also use the AND instruction to mask off the high byte from the value for the menu item. This is equivalent to subtracting 256 as long as you don’t have a menu item number greater than 511 ($1FF):

```assembly
LDA MITEM ; GET MENU ITEM #
AND #$00FF ; ADJUST TO 0,1,2...
ASL ; TIMES 2 = 0,2,4...
JSR (MENTBL,X) ; USE TABLE OF ADDRESSES
JMP MAIN ; BACK TO MAIN LOOP
```

Even though it only saves one byte, it’s mentioned here since you may come across it in sample listings.

You can also include a few special characters at the beginning of your title or menu item string for certain effects. The first is the use of a dash character as the first character of a menu item string, like this:

```
ASC ‘- \N258D’,00
```

The Menu Manager interprets a dash to signify a horizontal line in the menu at that position. Notice that as a menu item, it must still have its own ID number, in this example, 258. The D at the end is used to disable that line, so the user doesn’t inadvertently select it. Special menu modifiers like D will be discussed in more detail shortly.

In a complete menu, the use of the dash would look like this:

```
MENU ASC ‘>> Title \N1’,00
ASC ‘. Item1 \N257’,00
ASC ‘. \N258D’,00
ASC ‘. Item2 \N259’,00
ASC ‘%’
```

The other special menu name is the symbol @ (at sign) for the Apple logo character as a menu bar title. This is only used as the title for the first (left-hand) title in the menu bar, and appears as follows:

```
MENU ASC ‘>>@ \N1X’,00
ASC ‘—Item1 \N256’,00
ASC ‘.’
```

The @ symbol must not have any spaces or other characters other than the title character and end-of-title delimiter. The Apple menu is also different in that a special form of highlighting must be specified, called color-replace
highlighting. Because the title, that is the Apple icon, is in color, the normal inverting would make the Apple colors change as well. In color-replace highlighting, only the white background of a colored object is inverted. If for some reason you had a color title or an item name, you would request this type of highlighting. Color-replace highlighting is signified with the X character at the end of the title definition string.

The Apple menu is used as the heading for any desk accessories that may be available, and any other special functions you wish to include. However, it's not possible to define a menu with zero entries. Because there may not be any desk accessories available when your program runs, you should always include at least one of your own menu items in the Apple menu. Usually this is the About This Program... item that many programmers use to tell about the application.

**Designators**

Each string in the list of menu items has encoded into it not only the string and identifying number for the menu title or item, but also optional special designator characters that indicate whether that menu item is disabled, what the keyboard equivalent is, if any, and other such information. You have already seen some of these in the form of the X and D characters mentioned a little earlier. The reverse slash (\) is used to terminate a menu title or item name. There is no way to include the character \ in a name itself. The special designator characters are placed at the end of the title or item string following the name terminator (\) and may be used in any order you wish.

These designators may be used in either a title or a menu item name and are used as follows:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Title or item ID # (decimal) follows.</td>
</tr>
<tr>
<td>H</td>
<td>Title or item ID # (hex) follows.</td>
</tr>
<tr>
<td>D</td>
<td>Disables title or name.</td>
</tr>
<tr>
<td>X</td>
<td>Uses color-replace highlighting.</td>
</tr>
</tbody>
</table>

Item numbers may be either included in the string as an ASCII string value or defined as a two-byte word. The character N designates an ASCII number, which is stored a character at a time. For example, in the string:

```
ASC '\-Item1 \N256',00
```

The number 256 is stored as $32 $35 $36, the ASCII characters for 256. If for some reason you want to encode the menu item as a number value, you
can define the line like this:

```
ASC '--Item I \H'
DA $103
ASC 'D',00
```

In this case the special character $H$ tells the Menu Manager to expect a two-byte value for the menu item, which is then followed by the special character $D$ for a disabled menu item, and the string terminator $00$.

There are also a few other special characters which can only be used in the menu item names, not in a title. They are as follows:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Displays the item in bold text.</td>
</tr>
<tr>
<td>I</td>
<td>Displays the item in italic text.</td>
</tr>
<tr>
<td>U</td>
<td>Underlines the item name.</td>
</tr>
<tr>
<td>*</td>
<td>Keyboard (Open Apple) equivalent letters follow (two characters, always).</td>
</tr>
<tr>
<td>C</td>
<td>(Check)Mark character follows.</td>
</tr>
<tr>
<td>V</td>
<td>Puts dividing line under this name.</td>
</tr>
</tbody>
</table>

The $B$, $I$, and $U$ characters are self-explanatory. The asterisk is used to specify the characters that you will accept as a keyboard equivalent for that menu item. For example, if we wanted to accept Open Apple–Q to quit our program, this menu item would do it the trick:

```
ASC '--Quit \N258*Qq',00
```

Generally, the second keyboard character is the lowercase equivalent of the first character, so that the status of the Caps Lock key won’t matter when the user presses the Q key, but in theory the two characters can be anything you want.

The letter $C$ is used to designate the check-mark character that will be used at the far left of the menu item. This is usually used to show that a menu item, for example show page breaks, is active. The check character can be anything you wish. For example, this would define an $x$ as the mark for a menu item:

```
ASC '--Show Page Breaks \N258Cx',00
```

As you noticed in the Event Manager demonstration program, ASCII characters 17, 18, 19 and 20 ($12$, $13$, $14$ and $15 = \text{Control-Q, Control-R, Control-S, and Control-T}$) have been given special definitions. To get a real check mark in your menu, you can embed the proper ASCII value in your menu string like this:

```
ASC '--Show Page Breaks \N258C',12,00
```
This puts the value $12 (18 decimal) after the C in the menu item definition.

Although the dash is available to create a dividing line between menu items as mentioned earlier, you can also include a V character in the special characters at the end of the name to create a solid underline that goes all the way across the menu underneath a given item. This avoids having to define a disabled item to create the line, and it also means you can fit more items in the complete vertical menu.

A menu item with an underline divider below it would look like this:

```
ASC '--Item1 \N256V',00
```

You can combine as many of these special characters as you wish in a single menu-item definition. Although crowded, this is legal:

```
ASC '--Item1 \N256VX8IUCx*Aa',00
```

A typical menu list could look like this:

```
MENU2 ASC '>> Title2 \N2',00
ASC '--Item1 \N257V*Aa',00
ASC '--Item2 \N258D*Bb',12,00
ASC '--Item3 \N259V',00
ASC '--Item4 \N260*Qq',00
ASC '--- \N261D',00
ASC '.
```

A menu is added to the Menu Manager’s menu list by calling `NewMenu` ($2DOF) and passing it the address of the new menu data structure, like this:

```
PushLong MENU2 ; POINTER TO OUR DEFINITION
ToolCall $2DOF ; NewMenu
PullLong MENUHD ; SAVE MENU MGR. HANDLE
```

**Construction of the Menu Bar**

Defining a menu doesn’t put it in the menu bar, however; this just returns a handle to the menu definition so you can add it when you want later. Constructing the menu bar itself is done by repeatedly calling the tool `InsertMenu` ($0DOF). Although you can pass this tool a position value for where you want the new menu inserted, a simple application can just keep calling the routine and inserting each new menu at the far left. Thus, if you want the menu titles FILE, EDIT, and PENS in the menu bar, you call `InsertMenu` three times with PENS, EDIT and FILE, in that order.

```
PushLong MENUHD ; PUSH HANDLE TO MENU
PushWord #$0000 ; INSERT AT LEFT
ToolCall $0DOF ; InsertMenu
```

421
Chapter 19

Because the Apple icon menu involves desk accessories that the application itself will not know about, there is a call to the Desk Manager tool set, **FixAppleMenu ($1E05)** that is done to automatically add any available desk accessories to the specified menu. Since this is usually menu #1 with the Apple icon, this call looks like:

```
PushLong #$0001 ; MENU NUMBER
ToolCall $1E05 ; FixAppleMenu
```

Once all the menus have been defined and inserted in the menu bar, the Menu Manager needs to be told to calculate some internal constants for the size of the longest menu names, and the height of each name and the menu bar using the current font. This is done with **FixMenuBar ($130F)**:

```
PushWord #$0000 ; SPACE FOR RESULT
ToolCall $130F ; FixMenuBar
PullWord HT ; RETURNS HEIGHT OF BAR
```

This call returns the height of the menu bar in pixels, but this is unlikely to be needed by your application, so is usually ignored. (You do have to pull it off the stack, but you don’t have to keep the result.) You must call FixMenuBar whenever you change any menu item or list.

That pretty much takes care of it, except for one thing: We’ve got to draw the new menu on the screen. This is done with **DrawMenu**:

```
ToolCall $2A0F ; DrawMenu
```

Program 19-1 will create just about the simplest menu bar there is, and it will open a single window.

**A Simple Menu Bar**

Although Program 19-1 looks like the previous ProDOS 16 examples, there are some significant additions. When run, it opens a window on the DeskTop that you can drag by its title bar, resize with the zoom or grow box, or close (along with quit the program) by clicking in the go-away box. There is a very empty menu bar at the top of the screen with the Apple icon. If you have any New Desk Accessories, such as the clock, on your startup disk, these will appear in this menu.

Try opening a desk accessory and moving it around on the DeskTop with the sample window. Notice what happens when one window overlaps another. Click in the content region of each window successively and notice how each is activated while the other becomes inactive. When you’re done with the program, click in the go-away (close) box, and the program will return to whatever program selector was used to run it.
Although similar to the Event Manager demo program, this program adds some new techniques needed for a program that supports the Window and Menu Managers. The first change is found in the **STARTUP** section, where the instruction **TDC** (Transfer Direct Page to C Accumulator) is found. This instruction copies the Direct Page Register address (a two-byte value) into the Accumulator. The letter C is used for the Accumulator to remind you that two bytes are always transferred, regardless of whether the Accumulator is currently in the 8- or 16-bit mode at that instant.

Once transferred to the Accumulator, the direct-page value is saved in **MYDP**. This will be needed in the routine that refreshes the contents of our window, called the *update* routine. This routine is called by TaskMaster automatically, and at that time the direct page on entry to our routine will be that used by TaskMaster, not our program. If our routine needs any direct-page space, such as for an indirect pointer, it will have to temporarily save TaskMaster's direct page, switch to ours, do the update, and then restore TaskMaster's direct page. This will require that our direct page value be on hand when **UPDATE** is called. Storing it in MYDP accomplishes this.

The next new item is the generation of a third UserID in the **MM** (Memory Manager startup) routine. Strictly speaking, the **ORA $0200** on line 49 does not produce the second ID possible ($1202 for example), but rather produces the third ($1303) because the Accumulator at that point already holds $1102 from the previous instructions ($1102 ORA $0200 = $1302). In our case though, it really doesn’t matter as long as the UserID is different from the other two.

In **GETDP**, because we'll need direct page for the Menu Manager and Control Managers in addition to that previously obtained for the Event Manager and QuickDraw, a total of 6 pages ($600) of memory are obtained, and the base address is stored in our variable **DP**. As each block of direct page is assigned, DP will be incremented by the amount just used so that each new assignment will begin at the next available address (see lines 114 to 117 for example).

**LOADTOOLS**, on line 92, loads the RAM-based tools using the table **TOOLTBL** on lines 343-350. If you examine **TOOLTBL**, you'll see that the Menu Manager, Window Manager and Control Manager are loaded from the startup disk.

Once these are loaded, WM, CTRL and MENU start up each tool, followed by **DESK**, which starts up the Desk Accessory Manager in preparation of setting up the Apple icon.

**APPLE** begins the portion of the program that actually creates and displays the menu bar. NewHandle is first used to allocate memory in the Menu
Chapter 19

Manager for MENU1, which will be the Apple icon menu. Looking at MENU1, you can see there is only one entry with a message you can customize. This menu item has a dividing underline to set it off from any desk accessories that may be listed. At least one menu item must be defined here to protect against the possibility that there are no desk accessories on the disk, which would result in an empty menu.

Once the handle for the menu has been obtained and stored, the menu is added at the left of the menu bar with InsertMenu. DESKACC then calls FixAppleMenu to add any desk accessories to this menu. FIXBAR completes the menu bar definition followed by DRAWMENU, which displays the complete menu bar. If you had further menus you wanted to define and add to the menu bar, those instructions would be inserted before the APPLE routine, and they would be inserted in reverse order. There will be an example of this in an upcoming program.

Now it's time to open a window. Since this is a generalized routine that may be called many times within a program, OPEN is set off as a separate routine on lines 209 to 215. Let's see what OPEN does. As you can see, the routine is very short. All it does is pass a pointer to a window definition, WINDOW (lines 373–416), and call NewWindow ($090E), from which is returned the handle to the Window Manager's window record for this window. This is saved for future reference in WPTR, and the OPEN routine is finished.

If the visible bit is set in the wFrame definition, the act of calling NewWindow automatically opens the defined window at the position and size indicated within the window parameter list. TaskMaster will automatically keep track of the window from then on, until the window is closed, which will be discussed shortly. The parameter list for the defined window is based on the list described earlier in this chapter. You should look it over carefully to make sure you understand each component of the data structure.

The main event loop is on lines 153–184, in the sections MAIN, EVT and DOCMDS. MAIN is similar to the Event Manager demonstration program, except that this time we call TaskMaster ($1D0E) on lines 153–156. Notice that the call structure with the event mask, points to each item in the identical as in GetNextEvent.

The main difference in using TaskMaster over GetNextEvent is that the event record, EVRECORD (lines 314–339) must be extended to add two new variables, TDATA and TMASK. TDATA is where TaskMaster will return information about which menu item (first two bytes, TDATA) and which menu title (third and fourth bytes, TDATA + 2) were chosen. TMASK is a mask used to tell TaskMaster which events it should handle automatically. Each bit in
The Window and Menu Managers

TMASK specifies a certain type of event, as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Menu keys</td>
</tr>
<tr>
<td>1</td>
<td>Update handling</td>
</tr>
<tr>
<td>2</td>
<td>Find window</td>
</tr>
<tr>
<td>3</td>
<td>Menu select</td>
</tr>
<tr>
<td>4</td>
<td>Open New Desk Accessories.</td>
</tr>
<tr>
<td>5</td>
<td>Clicks in New Desk Accessory windows.</td>
</tr>
<tr>
<td>6</td>
<td>Drag window</td>
</tr>
<tr>
<td>7</td>
<td>Select window if click in content region.</td>
</tr>
<tr>
<td>8</td>
<td>Track go-away box</td>
</tr>
<tr>
<td>9</td>
<td>Track zoom box</td>
</tr>
<tr>
<td>10</td>
<td>Track drag box</td>
</tr>
<tr>
<td>11</td>
<td>Handle scrolling</td>
</tr>
<tr>
<td>12</td>
<td>Handle special menu (#s 250–255) events.</td>
</tr>
<tr>
<td>13–31</td>
<td>These bits always clear (reserved).</td>
</tr>
</tbody>
</table>

By selectively setting specific bits, you can elect to have TaskMaster automate only certain functions. Functions not handled by TaskMaster will be passed on to your application as an event. When TaskMaster is called, an event code indicating some type of window or DeskTop event is returned on the stack. The event could be related to the user clicking in a window, in a window control like a zoom box, or in a menu event like selecting an item from a menu.

<table>
<thead>
<tr>
<th>Event Code</th>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0000</td>
<td>wNoHit</td>
<td>No event.</td>
</tr>
<tr>
<td>$0010</td>
<td>wInDesk</td>
<td>In desktop, but not in a window or menu.</td>
</tr>
<tr>
<td>$0011</td>
<td>wInMenuBar</td>
<td>In the menu.</td>
</tr>
<tr>
<td>$0013</td>
<td>wInContent</td>
<td>Within the content region of a window.</td>
</tr>
<tr>
<td>$0014</td>
<td>wInDrag</td>
<td>Within the window’s drag region.</td>
</tr>
<tr>
<td>$0015</td>
<td>wInGrow</td>
<td>Within the window’s grow box.</td>
</tr>
<tr>
<td>$0016</td>
<td>wInGoAway</td>
<td>Within the window’s go-away (close) box.</td>
</tr>
<tr>
<td>$0017</td>
<td>wInZoom</td>
<td>Within the window’s zoom box.</td>
</tr>
<tr>
<td>$0018</td>
<td>wInInfo</td>
<td>Within the window’s information bar.</td>
</tr>
<tr>
<td>$0019</td>
<td>wInSpecial</td>
<td>Special menu item selected.</td>
</tr>
<tr>
<td>$001A</td>
<td>wInDeskItem</td>
<td>Desk Accessory was selected.</td>
</tr>
<tr>
<td>$001B</td>
<td>wInFrame</td>
<td>Within the frame of a window.</td>
</tr>
<tr>
<td>$8xxx</td>
<td>wInSysWindow</td>
<td>Within a desk accessory window.</td>
</tr>
</tbody>
</table>

Not all of these codes will always be returned. They are actually originally generated by the Window Manager tool **FindWindow** ($170E), which TaskMaster uses to see where a click or other mouse-down event occurred. If the TaskMaster TMASK has been set to handle the event, your application may not see an event code at all. For example, if the Track Zoom Box is enabled (bit 9 in TMASK), TaskMaster will handle the zoom action for you. Other events,
Chapter 19

like selecting a menu item, are returned by TaskMaster as an event, at which your application should check TDATA and TDATA + 2 to handle the menu item selected.

The Main Loop
Let's look at the main event loop to see how various events returned by TaskMaster are handled in the sample program.

GETEV starts the processing by pulling the result of the TaskMaster call off the stack. If the value is zero, then either no event has occurred or TaskMaster has handled some particular event for us and nothing remains to be done. The application will spend most of its time looping back to MAIN in this part of the program.

When an event that TaskMaster could not complete occurs, the BEQ test fails, and program execution falls through to EVT. At this point, I've included a JSR to a do-nothing routine called SPECIAL that could be used to do any special processing for every event, regardless of what type it is. This routine will be expanded in an upcoming program example. Remember that the Accumulator now holds the TaskMaster event code (any routine at SPECIAL must take care to preserve this), and line 167 checks to see if the event was a menu event ($11). If so, a JSR to the routine DOMENU is called.

DOMENU looks at TDATA for the menu item number that was selected, and tests to see if is was our custom message. In this case, it is ignored by jumping to the exit to the DOMENU routine, but a real application would handle the menu choice any way it wanted at that point. Any additional menu items can be tested and handled in a similar manner.

When a menu item has been selected, TaskMaster leaves the menu title inverted when it returns to your application. After completing whatever action for that menu item you wish, your application should unhighlighting the menu title. NORMAL does this by calling HiLiteMenu ($2C0F). This tool is used by passing it either a zero (for unhighlight) or a nonzero value (for highlight), and the menu title number for the menu you wish to highlight. NORMAL sets the highlight to unhighlight and uses TDATA + 2 to determine the current menu title that was selected.

The idea behind unhighlighting being the last action in a menu event handler is that the user can see something has happened while waiting for the routine to do its job. This may seem unnecessary for a routine that acts instantly, but for any function that may take more than a few seconds, it lets the user know something has happened. Particularly with a keyboard equivalent, leaving the menu title highlighted may be the user's only indication as to why the keyboard appears to be no longer responding.
If the TaskMaster event doesn’t indicate a menu event, we next check for a click in the close box ($16). If there is one, the program jumps to SHUTDOWN. In addition to the usual shutting down of any tools and de-allocating any memory we’ve used, SHUTDOWN now closes the window with the routine CLOSE (lines 235-249). Closing a window can mean two different things, and it’s important to understand the difference.

When the user closes a window, it may mean just hiding the window from view, or it may mean actually disposing of the window and any associated documents or memory blocks in use related to that window. Our CLOSE routine does the latter, completely removing our window from the control of the Window Manager and TaskMaster, and de-allocating any memory associated with that window. CLOSE in this form should be called only when you’re absolutely done with a window and its data.

If there are several windows open on the screen when your application receives the close event, you will need to first determine which window was active and needs to be closed. The Window Manger routine FrontWindow ($150E) returns the handle identifier for the currently active (and therefore in front) window. In Program 19-1 there is only one window active (TaskMaster will handle closing any desk accessory windows automatically), so the handle returned by FrontWindow is pulled off the stack and put in PTR as an easy way to ignore it.

Any memory associated with that window should then be disposed of, and the third UserID, ID3 is passed to DisposeAll to do this. This SHELL program does not have any document or application-generated memory associated with the window, but this step is included to show what would normally be done here.

Finally, the window is closed and Window Manager-associated memory de-allocated with the call CloseWindow. Remember, this is not the same as temporarily hiding a window. That technique will be discussed shortly.

Returning our attention to the main event loop, if a close event was not detected on line 172, then control passes to the handler of Event Manager events, in the routine DOCMDS (DO CoMmanDS). The Event Manager event codes will all be in the range of 0 to 15, as was explained in the Chapter 18 on the Event Manager. Knowing this, we can create a vector table to be used with indirect indexed addressing, as was also described in previous chapters, and also mentioned in this chapter with respect to handling menu commands.

Lines 354–369, titled CMDS, define the entry points for the routine to handle each possible event. This SHELL program is pretty fancy, as you can tell; every possible event is routed to IGNORE (a simple RTS). However, if your program were a word processor, you would look for keydown events and
process them accordingly. A drawing program might use the mouse-down event to start drawing. The DOCMDS routine (lines 179–184) shifts the event code (equivalent to multiplying by 2), and then does a JSR to the appropriate routine.

Program 19-1. SHELL Program

```
MX %00 ; TELL MERLIN WE'RE IN 16 BITS
REL DSK SHELL.L
LST OFF ; DON'T LIST MACROS
USE UTIL.MACS ; USE MACRO LIBRARY
LST ON ; LISTING BACK "ON"
EXP OFF ; DON'T EXPAND MACROS
TR ON ; DON'T PRINT ALL BYTES

=E100A8

PRODOS EQU $E100A8 ; STD. PRODOS 16 ENTRY

=0000

PTR EQU $00 ; OUR OWN DIRECT-PAGE PTR

; $00,01,02,03

********** STARTUP THE ENVIRONMENT **********

STARTUP PHK
PLB
TDC
STA MYDP

SETRES SEP $30 ; 8-BIT MODE
LDA #$5C ; JML (JMP LONG)
STAL $3F8 ; CTRLY VECTOR
REP $30 ; 16-BIT MODE
LDA #RESUME
STAL $3F9 ; $3F9,3FA
LDA #RESUME
STAL $3FB ; $3FB,3FC

TL ToolCall $0201 ; TOOL LOCATOR STARTUP

MM PushWord #$0000 ; SPACE FOR RESULT
ToolCall $0202 ; MEMORY MGR. STARTUP
```
The Window and Menu Managers

45 PullWord ID ; SAVE OUR ID
46 ORA #$0100 ; LIKELY = $1002
47 STA ID2 ; CREATE SUB-ID
48 ORA #$0200 ; LIKELY = $1102
49 STA ID3 ; 2ND SUB-ID FOR PICTURE DATA
50
51 GETDP PushLong #$0000 ; LIKELY = $1302
52 PushLong #$0000
53 PushLong #$600
54 ; 3 PAGES FOR QUICKDRAW
55 ; 1 PAGE FOR EVENT MGR.
56 ; 1 PAGE FOR MENU MGR.
57 ; 1 PAGE FOR CTRL. MGR.
58 PushWord ID2 ; SUBID
59 PushWord #$CO00 ; TYPE: LOCKED, FIXED
60 PushLong #$3000
61 ToolCall $0902 ; NewHandle
62
63 PullLong PTR ; GET HANDLE FOR NEW DP
64 LDA [PTR] ; LONG INDIRECT LOAD
65 STA DP ; SAVE THE DP ADDRESS
66
67 EM PushWord #$0000
68 PushWord #$0000
69 PushWord #$0000
70 PushWord #$0000
71 PushWord #$0000
72 PushWord #$0000
73 PushWord ID2
74 ToolCall $0206 ; EMStartUp
75
76 CLC ; PUSH DP ADDRESS (IN ACC.)
77 LDA DP ; QUEUE SIZE = DEFAULT = 20
78 ADC #$100 ; MIN X CLAMP FOR MOUSE = 0
79 STA DP ; MAX X CLAMP = 320
80
81 QD PushWord #$0000
82 PushWord #$0000
83 PushWord ID2
84 ToolCall $0204 ; QDStartUp
85
86 CLC ; PUSH DP ADDRESS ON STACK
87 LDA DP ; MASTER SCB = DEFAULT (320)
88 ADC #$300 ; MAX SCREEN SIZE FOR BOUNDSCRECT
89 STA DP ; SUBID
90
91 LOADTOOLS PushLong #TOOLTBL
92 ToolCall $0E01 ; LoadTools
93
94
Chapter 19

95 WM
96 PushWord ID2
97 ToolCall $020E ; WMStartUp
98 PushLong #$0000 ; 0 = DRAW ENTIRE SCREEN
99 ToolCall $399E ; Refresh (DRAW DESKTOP)
100
101 CTRL
102 PushWord ID2 ; SUBID
103 PushWord DP ; DP AREA FOR CTRL MGR.
104 ToolCall $0210 ; CtrlStartUp
105
106 CLC
107 LDA DP $100
108 STA DP
109
110 MENU
111 PushWord ID2 ; SUBID
112 PushWord DP ; DP AREA FOR MENU MGR.
113 ToolCall $020F ; MenuStartUp
114
115 CLC
116 LDA DP #100
117 STA DP
118
119 DESK
120 ToolCall $0205 ; DeskStartup
121 APPLE
122 PushLong #$0000 ; SPACE FOR RESULT
123 PushLong #MENU1 ; ADDR. OF MENU STRUCTURE
124 ToolCall $2D0F ; NewMenu
125 PullLong MENU1HD ; HANDLE FOR 1ST MENU
126
127 PushLong MENU1HD ; INSERT AT THIS POSITION
128 PushWord #$0000 ; (FRONT)
129 ToolCall $0D0F ; InsertMenu
130
131 DESKACC
132 PushWord #$0001 ; APPLE MENU ID
133 ToolCall $1E05 ; FixAppleMenu
134
135 FIXBAR
136 PushWord #$0000 ; ADD DA NAMES TO APPLE
137 ToolCall $130F ; MENU
138
139 FLA ; SPACE FOR RESULT
140
141 DRAWMENU ToolCall $2A0F ; FixMenuBar
142 ; CALC SIZES FOR EVERY MENU
143 ; GET MENU HEIGHT & DISCARD

008186: 68 ; DrawMenuBar

430
The Window and Menu Managers

08192: 20 FA 81
144 WIND JSR OPEN ; OPEN A WINDOW
145
146 SHOWCURS ToolCall $9104 ; ShowCursor
147
148
149 **********************************************
150 * MAIN EVENT LOOP
151 **********************************************
152
153 MAIN PushWord #$0000 ; SPACE FOR RESULT
154 PushWord #$FFFF ; ALLOW ALL EVENTS
155 PushLong #EVRECORD ; RECORD ADDRESS
156 ToolCall $1DOE ; TaskMaster
157
081B7: 68
158 GETEV PLA ; GET EVENT CODE IN ACC.
159 BEQ MAIN ; NO EVENT
160
161 **********************************************
162 * HANDLE THE EVENT
163 **********************************************
164
081BA: 20 F9 81
165 EVT JSR SPECIAL ; ANY SPECIAL EVENT HANDLING
166
167 081BD: C9 11 00
168 CMP #$11 ; MENU EVENT?
169
170 081C0: D0 06 =81C8
171 BNE .:1 ; NOPE
172
173 081C2: 20 DB 81
174 JSR DOMENU ; HANDLE IT
175
176 081C5: 4C A0 81
177 JMP MAIN ; BACK FOR MORE
178
179 081C8: C9 16 00
180 CMP #$16 ; CLOSE BOX
181
182 081CB: D0 03 =81D0
183 BNE DOCMDS
184
185 081CD: 4C 60 82
186 JMP SHUTDOWN ; ALL DONE!
187
188 081D0: AD F7 82
189 DOCMDS LDA EVENT ; EVENT * 2
190
191 081D3: 6A
192 ASL ; PUT IN X REG
193
194 081D4: AA
195 TAX
196
197 081D5: FC 1D 83
198 JSR (CMDS,X)
199
200 081D8: 4C A0 81
201 JMP MAIN ; BACK FOR MORE
202
203 081DB: AD 97 83
204 DOMENU LDA TDATA ; MENU ITEM NUMBER
205
206 081DE: C9 00 01
207 CMP #256 ; APPLE MENU MSSG?
208
209 081E1: D0 03 =81E6
210 BNE :.1 ; NOPE
211
212 081E3: 4C E6 81
213 JMP NORMAL ; IGNORE (BUT UNHILITE)
214
215 :.1 ; OTHER TESTS HERE...
216
217 NORMAL PushWord #$0000 ; 0 = UNHILIGHT
218
219
Chapter 19

196 PushWord 7DATA + 2 ; MENU HEADER #
197 ToolCall $2C0F ; HiLiteMenu
198
199 RTS
0081F8: 60
200
201 *************************************************
202
203 SPECIAL ; SPECIAL EVENT HANDLING
204
0081F9: 60
205 RTS
206
207
208
209 OPEN PushLong #$0000 ; SPACE FOR RESULT
210 PushLong #WINDOW ; ADDR. OF WINDOW DEFINITION
211 ToolCall $090E ; NewWindow
212
213 PullLong WPTR ; GET WIND. MGR. HANDLE FOR THIS
214
008219: 60
215 RTS
216
217 *************************************************
218
00821A: 8B
219 UPDATE PHB ; SAVE OTHER'S DATA BANK
00821B: 4B
220 PHK
00821C: AB
221 PLB ; DATA BANK = OURS
222
00821D: 0B
223 PHD ; SAVE THE DIRECT PAGE
00821E: AD F1 82
224 LDA MYDP ; USE OURS FOR NOW . . .
008221: 5B
225 TCD
226
227 * DO ANY WINDOW UPDATING HERE...
228
008222: 2B
229 PLD ; RESTORE THE DIRECT PAGE
008223: AB
230 PLB ; RESTORE THE ORIG. DATA BANK
008224: 6B
231 RTL ; BACK TO TASKMASTER!
232
233
234
235 CLOSE PushLong #$0000 ; SPACE FOR RESULT
236 ToolCall $150E ; FrontWindow
237
238
239
240
241 PullLong PTR ; STORE HANDLE, PTR IS AVAIL.
242

432
The Window and Menu Managers

243 PushWord ID3 ; SUBID FOR PICTURE DATA
244 ToolCall $1102 ; DisposeAll
245
246 PushLong WPTR ; HANDLE TO WINDOW RECORD
247 ToolCall $0B0E ; CloseWindow
248
00825E: 60

249 RTS
250
00825F: 60

251 IGNORE RTS
252

008260: 20 25 82

253 SHUTDOWN JSR CLOSE ; CLOSE THE WINDOW
254
255

0082CE: 22 A8 00 E1

257 SHUTDOWN JSR CLOSE ; CLOSE THE WINDOW
258
259 ToolCall $0305 ; DeskShutdown
260
261 ToolCall $030F ; MenuShutdown
262
263 ToolCall $0310 ; CtlShutdown
264
265 ToolCall $030E ; WMShutdown
266
267 ToolCall $0304 ; QDShutdown
268
269 ToolCall $0306 ; EMShutdown
270
271 PushWord ID2
272 ToolCall $1102 ; DisposeAll
273
274 PushWord ID
275 ToolCall $0302 ; MMShutdown
276
277 ToolCall $0301 ; TLShutdown
278

279
280
0082DA: 00 00 00 00

281 QUIT JSL PRODOS ; DO QUIT CALL
282 DA $29 ; QUIT CALL COMMAND VALUE
283 ADRL QUITBLK ; ADDRESS OF PARM TABLE
284 BRK $00 ; SHOULD NEVER GET HERE...
285
286

0082CE: 22 A8 00 E1

287

0082DA: 00 00 00 00

288 QUITBLK ADRL $0000 ; NO PATHNAME
289 DA $0000 ; STD. QUIT
290
291

0082EO: 4B

292

0082E1: AB

293 RESUME PHK ; SET OUR DATA BANK
294 PLB
Chapter 19

0082E2: 18 295 CLC ; SET NATIVE MODE
0082E3: FB 296 XCE ; 16-BIT MODE
0082E4: C2 30 297 REP $30 ; TRY TO SHUTDOWN
0082E6: 4C 60 82 298 JMP SHUTDOWN

0082E9: 00 00 302 ID DA $0000 ; OUR APPLICATION'S ID #
0082EB: 00 00 303 ID2 DA $0000 ; SUB-ID FOR MEMORY BLOCKS
0082ED: 00 00 304 ID3 DA $0000 ; 2ND SUB-ID

0082EF: 00 00 306 DP DA $0000 ; TEMP STORAGE FOR THE DP VALUE
0082F1: 00 00 307 MYDP DA $0000 ; TO STORE OUR DP ADDRESS

0082F3: 00 00 309 YLOC DA $0000 ; LOCAL Y POSN
0082F5: 00 00 310 XLOC DA $0000 ; LOCAL X POSN

EVRECORD ; DATA BLOCK WRITTEN BY EV MGR.

0082F7: 00 00 316 EVENT DA $0000 ; EVENT CODE
0082F9: 00 00 00 00 317 TYPE ADRL $0000 ; TYPE OF EVENT
0082FD: 00 00 00 00 318 TIME ADRL $0000 ; TIME SINCE STARTUP
008301: 00 00 319 YPOS DA $0000 ; Y-POSITION OF MOUSE
008303: 00 00 320 XPOS DA $0000 ; X-POSITION OF MOUSE
008305: 00 00 321 MOD DA $0000 ; EVENT MODIFIER

008307: 00 00 00 00 323 TDATA ADRL $0000 ; TASK MASTER MENU & ITEM #
00830B: FF 324 TMASK DFB %11111111 ; TASK MASTER EVENT MASK

00830C: 1F 333 DFB %00011111 ; (HIGH BYTE)

00830D: 00 00 339 DA $0000 ; BITS 16-31 ALWAYS CLEAR

434
The Window and Menu Managers

00830F: 03 00
008311: 0E 00
008313: 00 00
008315: 0F 00
008317: 00 00
008319: 10 00
00831B: 00 00

00831D: 5F 82
00831F: 5F 82
008321: 5F 82
008323: 5F 82
008325: 5F 82
008327: 5F 82
008329: 5F 82
00832B: 5F 82
00832D: 5F 82
00832F: 5F 82
008331: 5F 82
008333: 5F 82
008335: 5F 82
008337: 5F 82
008339: 5F 82
00833B: 5F 82

00833D: 00 00 00 00
008341: 11 41 70 70
008353: 4E 00
008355: E5 DD

WPTR  ADRL $0000
WTITLE  STR 'Apple IIGS Window'
WINDOW  DA WINDEND-WINDOW

-------------------------------
TOOLBL DA 3   ; THREE TOOLS TO LOAD

DA 14   ; WINDOW MGR
DA 0    ; ANY VERSION
DA 15   ; MENU MGR
DA 0    ; ANY VERSION
DA 16   ; CONTROL MGR.
DA 0    ; ANY VERSION

CMDS  DA IGNORE   ; NULL EVENT
C1  DA IGNORE    ; MOUSE DOWN
C2  DA IGNORE    ; MOUSE UP
C3  DA IGNORE   ; KEY DOWN
C4  DA IGNORE   ; UNDEFINED
C5  DA IGNORE   ; AUTOKEY
C6  DA IGNORE   ; UPDATE WINDOW EVENT
C7  DA IGNORE   ; UNDEFINED
C8  DA IGNORE   ; ACTIVATE WINDOW EVENT
C9  DA IGNORE   ; SWITCH EVENT
C10 DA IGNORE  ; DESK ACCESSORY EVENT
C11 DA IGNORE  ; DEVICE DRIVER EVENT
C12 DA IGNORE ; APPLICATION DEFINED EVENT
C13 DA IGNORE ; APPLICATION DEFINED EVENT
C14 DA IGNORE ; APPLICATION DEFINED EVENT
C15 DA IGNORE ; APPLICATION DEFINED EVENT

-------------------------------
WPTR  ADRL $0000   ; POINTER TO WINDOW RECORD
WTITLE  STR 'Apple IIGS Window'
WINDOW  DA WINDEND-WINDOW

-------------------------------

(strpos)
Chapter 19

390 ; BIT 4 = (NO) INFORMATION BAR
391 ; BIT 3 = (NO) INDEPENDENT CTRLs
392 ; BIT 2 = ALLOCATED BY NEWWINDOW
393 ; BIT 1 = (NOT) CURRENTLY ZOOMED
394 ; BIT 0 = HIGHLIGHTED
395 ; POINTER TO TITLE
396 ; REFERENCE CONSTANT
397 ; ZOOM RECTANGLE
398 ; COLOR TABLE
399 ; ORIGIN OFFSET
400 ; HEIGHT, WIDTH DATA AREA
401 ; HEIGHT, WIDTH MAX WINDOW
402 ; VERT., HORIZ. SCROLL.
403 ; VERT., HORIZ. PAGE INCREMENT
404 ; INFO BAR REFERENCE CONSTANT
405 ; INFO BAR HEIGHT (NONE)
406 ; FRAME PROCEDURE ADDR. (NONE)
407 ; INFO BAR PROCEDURE ADDR (NONE)
408 ; CONTENT PROCEDURE ADDR.
409 ; CONTENT REGION OF WINDOW
410 ; UPPER LEFT VERT. POSITION
411 ; UPPER LEFT HORIZ. POSN
412 ; LOWER RIGHT VERT. POSN
413 ; LOWER RIGHT HORIZ. POSN
414 ; PUT WINDOW AT FRONT ($FFF.. = -1)
415 ; STORAGE
416 ; END OF DATA STRUCTURE
417
418
419
420 ; APPLE MENU
421 ; END OF MENU
422 ; END OF MENU
423 ; STORAGE FOR HANDLE
424 ; STORAGE FOR HANDLE
425 ; CHECKSUM FOR VERIFICATION

--End Merlin-16 assembly, 973 bytes, errors: 0

436
Monitoring Events in a Window

Although the demonstration program just presented should work properly for you, it lacks routines to actually update the window with any information. In addition, it would be worthwhile to have a program that could provide a little more information about what is actually going on in the system each time an event occurs.

EVENT.DISPLAY, our next program, combines the SHELL program just presented with parts of the Event Manager demonstration program from the previous chapter. EVENT.DISPLAY will display a description of each event as it occurs within the window on the DeskTop, and it will print out not only the mouse global mouse coordinates (relative to the BoundsRect—the DeskTop) but also the local coordinates within the window.

By experimenting with activities like moving the mouse around, opening a New Desk Accessory and moving it around, and other actions on the DeskTop, you’ll gain a better understanding of the entire DeskTop and window environment, and also how TaskMaster handles certain events and passes others through to the application.

Because the listing for the total program would be fairly large (about 1000 lines), we’ll step through how to combine the two existing programs using source listings you’ve already entered, rather than retyping the entire program.

Start with the SHELL program, and save it under the new name, EVENT.DISPLAY. You may also want to change the title banner at the top of the source listing to this:

```
*****************************************************
* WINDOW WITH EVENT DISPLAY  *
* MERLIN ASSEMBLER            *
*****************************************************
```

Having saved this new source file, load Program 18-2, EVENT MANAGER DEMO from Chapter 18, and delete lines 310–320, which are the Event Record. It is already present in the SHELL. Next, delete lines 301–305. This removes the duplicate labels ID, ID2, and DP, which are also already in SHELL. Finally, delete lines 1–176, and the checksum instruction at the end. Save the resulting program segment under the name EVENT.SEGMENT. This will be added to the SHELL program in a moment. This program segment could be linked into the SHELL program by defining the necessary EXT (external) and ENT (entry) labels for the Merlin or APW assemblers, but, for a relatively short program like this, it’s probably just as easy to combine them in one listing.

The easiest way to do this is to first use the text copy command in your assembler to copy the entire source list for EVENT.SEGMENT onto the clipboard of the assembler’s editor. In Merlin, this is done by positioning the cursor
Chapter 19

on the first line of the program and then pressing Open Apple-C to start the
copy selection. Next, press Open Apple-Q to select from that point to the very
end of the listing. Finally, press Open Apple-C again to complete the copy of
this text onto *Merlin*’s clipboard.

Now, load the program EVENT.DISPLAY and press Open Apple-N to
move to the end of that listing; then move the cursor to line 428, which should
correspond to the checksum instruction. Pressing Open Apple-V (for paste) at
that point should make the entire EVENT.SEGMENT code appear as it is
pasted from the *Merlin* clipboard into the program. Check to make sure every­
thing looks OK, and then resave this new version back to the disk under the
current name, EVENT.DISPLAY. If you’re using the *APW* assembler, use follow
the same procedure using the *APW* editor commands to copy and paste text.

Now, the only remaining changes are to add a few lines here and there
to make the UPDATE routine fill in the event message within our window, and
to do a few other minor changes.

For each change, we’ll reprint the neighboring lines so you can see ex­
actly what the finished portion should look like.

The beginning of the program first needs to be modified to set up
MSSGPTR as was done in the Event Manager demonstration program. You’ll
recall that MSSGPTR will be used as a pointer into the list of event descrip­
tions as each event occurs.

```
PTR EQU $00 ; OUR OWN DIRECT PAGE PTR
MSSGPTR EQU $04 ; POINTER TO ANY MESSAGE
STARTUP PHK
PLB
TDC
STA MYDP
LDA #$STARTUP ; GET OUR DATA BANK
STA MSSGPTR+2 ; WRITE HIGH WORD TO OUR DATA BANK
SETRES SEP $30 ; 8-BIT MODE
```

The next modification is to the main event loop, which starts at the label
MAIN.

```
MAIN PushWord #$0000 ; SPACE FOR RESULT
PushWord #$FFFF ; ALLOW ALL EVENTS
PushLong #$EVRECORD ; RECORD ADDRESS
ToolCall $1D0E ; TaskMaster
```

438
The only change here is to insert a JSR to the routine MPOSN, which will print out the current mouse position for each pass through the event loop, regardless of whether an event has actually occurred or not. MPOSN will be described shortly.

In the SHELL program there was a JSR to a routine called SPECIAL, which consisted of nothing more than a JSR. This is where we will print out the description of each and every event that is passed on by TaskMaster. Add the instruction JSR PRINTEV to the SPECIAL routine:

```
SPECIAL ; SPECIAL EVENT HANDLING
  JSR PRINTEV ; EVENT VERSION
RTS
```

With the Window Manager, a vector can be set up in the window parameter list to point to the routine which should be called whenever the window needs to be updated by TaskMaster. For example, this would occur when the window was resized, or when another window that partially obscured ours was moved out of the way. The SHELL program already contains the foundation of any update routine, namely the setting of the data bank and program registers to those of our application and then restoring them after the update action takes place. Notice that the UPDATE routine always ends with an RTL (never an RTS). Simply insert the JSR EVMSSG into the update routine so that it looks like:

```
UPDATE PHB ; SAVE OTHER'S DATA BANK
  PHK
  PLB ; DATA BANK = OURS
  PHD ; SAVE THE DIRECT PAGE
  LDA MYDP ; USE OURS FOR NOW...
  TCD
  JSR EVMSSG ; PRINT EVENT MGR. MMSG
  PLD ; RESTORE THE DIRECT PAGE
  PLB ; RESTORE THE ORIG. DATA BANK
  RTL ; BACK TO TASKMASTER!
```

Now for the MPOSN routine. At the beginning of the EVENT.SEGMENT code that you added to the program, insert two new routines, MPOSN and PRINTEV. These routines correspond to the code that would be executed whenever the application updated its own window. The key points are that two
Window Manager routines, StartDrawing and SetOrigin, must be at the beginning and end of any routine which draws in a window during the application.

Although the local coordinates of the upper left corner of the visible window shift as the scroll bars are adjusted, QuickDraw internally always keeps the upper left corner at 0,0. When UPDATE is called, the Window Manager temporarily sets the origin back to the proper value while our UPDATE routine is drawing on the screen, but then it immediately reverts back to 0,0 when our routine exits.

When executing a window update within an application (but separate from the window definition update vector), it is necessary to manually reset the coordinates of the window to the proper local coordinates with the command StartDrawing. StartDrawing and SetOrigin are used by passing the handle to the specified window record. Once the window update is complete, the tool call SetOrigin is used to return the upper left corner of the window back to 0,0 for the Window Manager.

MPOSN prints the local and global coordinates of the mouse on a continual basis. PRINTEV is only called from within SPECIAL, which is executed whenever an actual event occurs.

```
MPOSN  PushLong WPTR   ; PRINT MOUSE POSITION
       ToolCall $4D0E   ; StartDrawing
                        ; MUST DO THIS FOR A DRAW
                        ; OUTSIDE THE UPDATE ROUTINE
                        ; TO SET ORIGIN CORRECTLY

JSR   MOUSE

PushWord #$0000   ; X = 0
PushWord #$0000   ; Y = 0
ToolCall $2304    ; SetOrigin
                        ; MUST DO THIS FOR A DRAW
                        ; OUTSIDE THE UPDATE ROUTINE
                        ; TO RETURN ORIGIN TO 0,0

RTS

*********************************************
• PRINT EVENT DESCRIPTION
•••••••••••••••••••••••••••••••••••••••••••••
PRINTEV PHA   ; SAVE EVENT CODE IN ACC.
       PushLong WPTR   ; HANDLE FOR WINDOW TO DRAW IN
       ToolCall $4D0E   ; StartDrawing
       JSR   EVMSSG    ; UPDATE EVENT DESCRIPTION

PushWord #$0000   ; X = 0
PushWord #$0000   ; Y = 0

*********************************************
```

440
The Window and Menu Managers

ToolCall $2304 ; SetOrigin (BACK TO 0,0)

PLA ; PUT EVENT CODE BACK IN ACC.

RTS

This would pretty much take care of things, except that the MOUSE routine from the Event Manager demonstration program only printed out the global coordinates of the mouse. Rewrite the MOUSE routine as follows:

MOUSE

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PushLong #YLOC</td>
<td>DATA TO REWRITE</td>
</tr>
<tr>
<td>ToolCall $0C06</td>
<td>GetMouse (LOCAL)</td>
</tr>
</tbody>
</table>

XPOSN

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PushWord XPOS</td>
<td>GET X POSITION</td>
</tr>
<tr>
<td>PushLong #XMSG+5</td>
<td>ADDR. OF BUFFER</td>
</tr>
<tr>
<td>PushWord #4</td>
<td>4 CHAR OUTPUT</td>
</tr>
<tr>
<td>PushWord #$0001</td>
<td>SIGNED NUMBER FLAG</td>
</tr>
<tr>
<td>ToolCall $2608</td>
<td>Int2Dec</td>
</tr>
<tr>
<td>PushWord #XWC</td>
<td>CONVERT TO ASCII DECIMAL STR$</td>
</tr>
</tbody>
</table>

YPOSN

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PushWord YPOS</td>
<td>GET Y POSITION</td>
</tr>
<tr>
<td>PushLong #YMSG+5</td>
<td>ADDR. OF BUFFER</td>
</tr>
<tr>
<td>PushWord #4</td>
<td>4 CHAR OUTPUT</td>
</tr>
<tr>
<td>PushWord #$0001</td>
<td>SIGNED NUMBER FLAG</td>
</tr>
<tr>
<td>ToolCall $2608</td>
<td>Int2Dec</td>
</tr>
<tr>
<td>PushWord #YMS</td>
<td>CONVERT TO ASCII DECIMAL STR$</td>
</tr>
</tbody>
</table>

X2

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PushWord XLOC</td>
<td>GET X LOCAL</td>
</tr>
<tr>
<td>PushLong #XMSG+23</td>
<td>ADDR. OF BUFFER</td>
</tr>
<tr>
<td>PushWord #4</td>
<td>4 CHAR OUTPUT</td>
</tr>
<tr>
<td>PushWord #$0001</td>
<td>SIGNED NUMBER FLAG</td>
</tr>
<tr>
<td>ToolCall $2608</td>
<td>Int2Dec</td>
</tr>
<tr>
<td>PushWord #XWC</td>
<td>CONVERT TO ASCII DECIMAL STR$</td>
</tr>
</tbody>
</table>

Y2

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PushWord YLOC</td>
<td>GET Y LOCAL</td>
</tr>
<tr>
<td>PushLong #YMSG+23</td>
<td>ADDR. OF BUFFER</td>
</tr>
<tr>
<td>PushWord #4</td>
<td>4 CHAR OUTPUT</td>
</tr>
<tr>
<td>PushWord #$0001</td>
<td>SIGNED NUMBER FLAG</td>
</tr>
<tr>
<td>ToolCall $2608</td>
<td>Int2Dec</td>
</tr>
<tr>
<td>LDA #0</td>
<td>X = 0</td>
</tr>
<tr>
<td>STA CH</td>
<td></td>
</tr>
<tr>
<td>LDA #10</td>
<td>Y = 10</td>
</tr>
<tr>
<td>STA CV</td>
<td></td>
</tr>
<tr>
<td>JSR PRINT</td>
<td></td>
</tr>
<tr>
<td>STR 'X = 0000' Local X = 0000</td>
<td></td>
</tr>
<tr>
<td>JSR CR</td>
<td></td>
</tr>
<tr>
<td>JSR PRINT</td>
<td></td>
</tr>
</tbody>
</table>

YMSG

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR 'Y = 0000' Local Y = 0000</td>
<td></td>
</tr>
<tr>
<td>JSR CR</td>
<td></td>
</tr>
</tbody>
</table>

RTS
Chapter 19

The main change here is to lengthen the output strings for XMSG and YMSG to include the local coordinates, and to use the tool call GetMouse, which returns the coordinates of the mouse in local (within-the-window = PortRect) coordinates. GetMouse writes the position into a data structure indicated by the pointer pushed on the stack when the routine is called.

In our new program, the local mouse coordinates will be written into YLOC and XLOC, which were included in the original SHELL program. As usual, the Y point is allocated first in the data structure.

When you've made the changes indicated here, double-check the listing, then assemble and link it as usual. Remember to save the new listing to disk before you assemble and link it.

If the changes are made exactly as described here, the new checksum at the end of the listing should be $28, and the program should assemble to be 1780 bytes long ($6F4).

Running the Event Display Program

When you run the Event Display program, there are some things to try and notice, things that illustrate some very important concepts about how the Window Manager works.

When the program first runs, before you even move the mouse, an event is displayed called update. This is the Window Manager telling the application that the window has been opened and may need to be updated. In our application, the update takes place when SPECIAL calls PRINTEV, which then calls the EVMSSG routine. The contents will also be automatically updated via the Window Manager and the UPDATE routine specified in the window parameter list whenever the window is resized or scrolled, or the like, bringing a previously obscured portion of the window into view.

With the cursor in the upper left corner of the screen, notice that the global coordinates of the mouse, as reported by the Event Manager in the event record (EVRECORD), are 0,0. Because the upper left corner of the content region of our application's window is also 0,0 in local coordinates, the DeskTop corner at this moment corresponds to \(-20, -30\) (local).

Move the cursor to the upper left corner of the content region of the window to demonstrate this, and notice that the content region's global coordinates are 20,30 as specified in the wPosition field of the window parameter list. You can move the cursor and explore the coordinates assigned to the entire window and DeskTop areas at this point. The lower right corner of the content region should have global coordinates of 200,100, also as specified in the window parameter list.

Now drag the window by its title bar to a new location on the DeskTop.
Notice that the local coordinates within the window remain fixed, while the global coordinates of the window change as expected.

Click the mouse in the content region of the window, and see the mouse-down and mouse-up events displayed on the screen. Notice that, as you click in the title bar, the mouse-down and mouse-up events are not passed through by TaskMaster, because it has already handled these itself. Click in the zoom box to make the window fill the screen. The event passed through now becomes update. Click a few times in the window, then zoom back to the previous size.

Now press the down-arrow key once on the vertical scroll bar in the window frame. The text printed in the window is automatically clipped as it scrolls out of view. Move the cursor and see that the local coordinates of the upper left corner of the content region are now 0,4. The vertical scroll increment as specified in the window parameter list was four pixels. Experiment with the scroll controls, including paging and sliding the thumb control, and observe how the local coordinates automatically change.

If at some point your application wanted to know what was currently in view in the window, there is a QuickDraw routine called `GetPortRect ($1E04)`, that returns the current rectangle in local coordinates; this routine can be used to determine exactly what part of the underlying data area is being viewed. There is also a Window Manager routine, `GetCOrigin ($3F0E)`, that returns the current local coordinates of the origin (upper left corner) of the content window at that moment.

Before continuing, return the window scroll bars to their starting position so that 0,0 is the origin of the window once more, and you can see the complete event display. Now click once in the menu bar—in the middle, away from the Apple icon. Notice that no event is passed through. Now select the Apple menu, and select Your Message. . . . The menu will flash when it is selected, and then the NORMAL routine in our DOMENU section of the program will unhighlight the title.

If you have a desk accessory listed (if you don't, try to put one on your system disk to try this out), select it to open a second window on the DeskTop. Click in one window and then the other to see how one is activated and the other is de-activated automatically. If you click in a window, you’ll see the event activate show up. This tells the application that windows are being changed, and is different from an update event. When an activate or update event occurs, the handle to the related window is passed in the TYPE field of the event record (also called the message field).

After you’ve thoroughly explored the DeskTop and window environment, click in the close box of the application window to quit the program.
Chapter 20

A Drawing Program for the Apple II GS
QuickDraw and the Window Manager don’t care whether the drawing they’re doing is on the screen or a real document in memory. This also means that just drawing something in a window does not automatically constitute any permanent record of that image anywhere in the computer.

If all you want to do is to draw something on the screen, and you don’t need an associated permanent record, then the techniques presented so far will be adequate.

Unfortunately, this is rarely the case. It’s a simple matter to rewrite the SHELL program to draw lines on the screen whenever a mouse-down event is detected. However, as soon as you scrolled whatever you drew out of view, it would be lost forever because there is no underlying document associated with the window. In Event Display program, when you changed the viewing window with the scroll bars or the grow box, any part of the message clipped from being printed just didn’t exist. The illusion, of course, is that the missing or clipped letters are just out of sight behind the window frame, but that’s all it is—an illusion.

If you want actions taken in your window to be permanently recorded, it’s up to you to create a block of memory somewhere else in the computer, and to store data there as is appropriate to your application.

This information may not always be graphics images. If you’re writing a database, for example, the document itself would consist of the words and numbers that made up your file. The display would be done by expressing a portion of that data as a graphics printout within an open window. You normally would not, as a matter of course, generate an entire graphics version of the document that the window then scrolled over.

For instance, if the origin of the scrolled window was currently 0,200, and you knew that each line of text in your list of names was 10 pixels high, a quick calculation (200/10 = 20) would tell you that you could start printing in the window with the twentieth name in your list. A look at the PortRect would
tell you how high the window was, and dividing that height also by 10 would
tell you how many names to print. All text to the left and right would be auto-
matically clipped depending on the horizontal scroll position and width of the
window. Thus, your program would only print, for example, records 20
through 25. It would not have converted the entire document to a graphics im-
age that the window then scrolled over.

In the case of a paint program, there is an underlying graphics docu-
ment. That document is a pixel image of the entire picture, and is stored in
memory as continuous block of memory allocated by the memory manager.

QuickDraw allows for the definition of a new drawing area by either cre-
ating a new GrafPort or by redirecting the memory controlled by the current
GrafPort to a new location. The memory controlled by the GrafPort is de-
scribed by a data structure called the PortLocInfo, which looks like this:

```
PortLocInfo
PortSCB DFB $00 ; MASTER SCB BYTE
DFB $00 ; RESERVED BYTE
Loc ADRL $0000 ; LONG ADDRESS OF MEMORY BLOCK
Width DA $0000 ; WIDTH IN BYTES OF IMAGE
BoundsRect ; RECTANGLE COORDINATES IN PIXELS
V1 DA $0000 ; VERTICAL UPPER LEFT COORD.
H1 DA $0000 ; HORIZ. UPPER LEFT COORD.
V2 DA $0000 ; VERTICAL LOWER RIGHT COORD.
H2 DA $0000 ; HORIZ. LOWER RIGHT COORD.
```

### Defining a Block of Memory

To define a block of memory equivalent to the super hi-res screen, the follow-
ing PortLocInfo structure could be used:

```
PICBLK ; PortLocInfo for a new image
DFB $00 ; $00 = 320 MODE
DFB $00 ; UNUSED
ADRL $E10000 ; SUPER HI-RES SCREEN
DA 160 ; 160 BYTES WIDE
DA 0,0 ; UPPER LEFT = 0,0
DA 200,320 ; LOWER RIGHT = 320,200
```

The PortLocInfo data structure can be generalized into a definition of
any pixel memory area, called the LocInfo block. By changing the address of the
pixel image, and passing this to the GrafPort using a QuickDraw call
SetPortInfo, you can redirect QuickDraw to draw anywhere into memory.
We'll see how to do that a little later.

You can use a LocInfo data structure in two other QuickDraw com-
mands, PPToPort ($D604) and PaintPixels ($F04).
PPToPort (Paint Pixels To Port) will be used for our update routine. It transfers part of a pixel image somewhere in memory into the current PortRect of the GrafPort, using the current clipping windows. The result is that the entire window is updated using the pixels from a graphic document we’ll create in memory. Figure 20-1 is the call diagram for PPToPort.

Figure 20-1. PPToPort ($D604)

Stack Before Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>DestinationX</td>
</tr>
<tr>
<td>DestinationY</td>
</tr>
<tr>
<td>TransferMode</td>
</tr>
</tbody>
</table>

←SP: Stack pointer after setup.

Stack After Call:

<table>
<thead>
<tr>
<th>Previous Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>←SP: Stack pointer after return from routine.</td>
</tr>
</tbody>
</table>

The source parameter block determines where the pixel image is in memory. The address of the image is stored within the Info block for the image must usually be filled in by dereferencing the handle to the memory block you’ve allocated as the graphics document.

PPToPort allows you to specify a rectangle as just part of the entire document to copy into the current window’s PortRect. This can be determined by calling the GetPortRect, but Program 20-1 just takes the brute force approach of copying the entire document image to the window, letting the normal clipping functions filter out the excess.

Destination X and Y specify where in the destination port the image will be transferred. For our update routine, both the source rectangle and the destination point will use 0,0. These are equivalent to the local coordinates when dealing with the window. Again, the fact that the window may be currently scrolled so that its origin is no longer 0,0 is not a concern since we’ll be copying the entire image over the window, and the clipping regions of the Window Manager will ensure that only the content region of the active window is changed.

The transfer mode refers to a code value that tells the routine whether to
use an AND, ORA, or other function in transferring the pixels from our document to the window. If you wanted to blend two images, for example, you could use a different transfer mode than a straight copy. Table 20-1 is a summary of the possible transfer modes.

Table 20-1. Transfer Mode

<table>
<thead>
<tr>
<th>Value</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0000</td>
<td>Copy</td>
<td>Copy pixels to destination, overwriting whatever is already there.</td>
</tr>
<tr>
<td>$8000</td>
<td>NotCopy</td>
<td>Copy inverse of pixels to destination, as in Copy.</td>
</tr>
<tr>
<td>$0001</td>
<td>OR</td>
<td>Use OR logic to overlay (blend) pixels.</td>
</tr>
<tr>
<td>$8001</td>
<td>NotOR</td>
<td>Overlay inverted pixels.</td>
</tr>
<tr>
<td>$0002</td>
<td>XOR</td>
<td>Exclusive OR pixels with destination. This allows you to undo the transfer by repeating the transfer with XOR again.</td>
</tr>
<tr>
<td>$8002</td>
<td>NotXOR</td>
<td>Exclusive OR of inverted pixels.</td>
</tr>
<tr>
<td>$0003</td>
<td>BIC</td>
<td>A special logic function for Bit Clear ANDing. This clears bits in the destination corresponding to bits set in the source image.</td>
</tr>
<tr>
<td>$8003</td>
<td>NotBIC</td>
<td>BIC function using inverse of pixel image.</td>
</tr>
</tbody>
</table>

One of the easiest ways to experiment with these would be to add the QuickDraw tool call SetPenMode ($2E04) to set the drawing mode to the value of your choice. SetPenMode looks like this in a program:

```
PushWord #MODE
ToolCall $2E04
```

In fact, once you get this program running as it is presented here, it would be an excellent exercise for you to add a new menu of pen modes to try out what you’ve learned.

**Updating the Screen**

Getting back to the update routine, PPToPort is designed specifically for transferring a portion of the existing document to the current window (port). This is fine for the update routine, but how do we transfer the image of a line just drawn on the screen into the actual document somewhere in memory? To do this we could probably change the current port to be our document, and the source document to be the PortRect of the window, and then call PPToPort, but there is another routine, called PaintPixels, which is generalized for transferring pixels from one place to another.

PaintPixels is passed a pointer to another parameter block, which in turn points to several LocInfo data structures to determine which pixels to transfer. The structure of the PaintPixels parameter block is as follows:
A Drawing Program for the Apple IIGS

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer</td>
<td>PtrToSourceLoclnfo</td>
<td>Pointer to the LocInfo structure for the source pixel.</td>
</tr>
<tr>
<td>Pointer</td>
<td>PtrToDestLocInfo</td>
<td>Pointer to the LocInfo for the destination image.</td>
</tr>
<tr>
<td>Pointer</td>
<td>PtrToSourceRect</td>
<td>Pointer to rectangle within source image.</td>
</tr>
<tr>
<td>Pointer</td>
<td>PtrToDestPoint</td>
<td>Pointer to point at which transfer will begin.</td>
</tr>
<tr>
<td>Word</td>
<td>Mode</td>
<td>Transfer mode to use.</td>
</tr>
<tr>
<td>Handle</td>
<td>MaskHandle (ClipRgn)</td>
<td>Handle to region to use as mask over destination.</td>
</tr>
</tbody>
</table>

Setting up the clipping region for MaskHandle is the new item here. A region is a special data structure within QuickDraw that can describe all sorts of strange shapes, like the profile of W.C. Fields and his cigar. You can use a region as a clipping mask so that only a portion of an entire pixel image will be transferred to the destination.

For our purposes, a rectangle will do just fine; this can be defined using two calls: **NewRgn ($6704)**, which creates a region data structure, including the automatic allocation of some memory from the Memory Manager; and **SetRect ($6B04)**, which sets a given region equal to a rectangle passed to that routine. With these two calls, we can create a rectangular region equal to our document in memory that will satisfy the requirement for the PaintPixel command.

You may have already guessed that there are a number of options in using these commands that will all produce the same results. We could just as easily set the source and destination rectangles equal to the entire screen and the entire document respectively, and then set the Clip Region equal to just the part of our document to be updated from the window. But in the end the results are all the same. You can design your application to use whichever approach best suits your programming style.

**Constructing the Paint Program**

Hopefully, this gives you an idea of where we’re headed to put together a simple sketching program using QuickDraw and the Window Manager. To do this, we’ll again start with the SHELL program, Program 19-1, and then we’ll add some routines to turn it into a drawing program. As before, start off by loading the source file SHELL, and save it under a new name. We’ll call the new program SIMPLE.SKTCH.

Now add a block of new code to the end of the program, just before the checksum byte. Use Program 20-1 for the new addition.

451
Chapter 20

Program 20-1. Sketcher
(See instructions above before entering program.)

************************************************
* SKETCHER ADDITIONS TO SHELL   *
************************************************

DOCSETUP PushLong #$0000 ; SPACE FOR RESULT
         PushLong #$8000 ; HOW MUCH MEM WE NEED
         PushWord ID3   ; SUB-ID
         PushWord #$0000 ; MEM ATTRIBUTE = NO RESTRICTIONS
         PushLong #$0000 ; LOCATION NOT IMPORTANT
         ToolCall $0902 ; NewHandle
         PullLong PICHNDL ; GET HANDLE TO MEMORY BLOCK
         JSR ERASE       ; ERASE PICTURE AREA

SETREGN PushLong #$0000 ; SPACE FOR RESULT
         ToolCall $6704 ; NewRgn (GET AN OFFSCREEN
                       ; DRAWING AREA.)
         PullLong PREGION ; GET HANDLE AND SAVE IT
         PushLong PREGION
         PushWord #$0000 ; H1 = 0
         PushWord #$0000 ; V1 = 0
         PushWord #320   ; H2 = 320
         PushWord #200   ; V2 = 200
         ToolCall $6B04 ; SetRect
                       ; MAKE THE REGION A RECTANGLE
         RTS

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

ERASE   PushLong PICHNDL
         ToolCall $2002 ; HLock
                       ; MAKE SURE IT DOESN'T MOVE

         LDA [PICHNDL] ; LONG INDIRECT LOAD
         STA PTR        ; GET THE MEM ADDRESS
         LDY #$02
         LDA [PICHNDL],Y
         STA PTR+2      ; (PTR) = ADDR. OF PICTURE

CLR      LDA #$EEE ; CLEAR BLOCK OF MEMORY TO COLOR #14
         LDY #$0000 ; BEG. OF BLOCK
         :1
         STA [PTR],Y
         INY
         CPY #32000 ; DONE YET?
         BCC :1         ; NOPE

452
A Drawing Program for the Apple IIgs

UNLOCK
PushLong PICHNDL
ToolCall $2202 ; HUnlock
PushLong #WINRECT ; POINTER TO WINDOW RECTANGLE
ToolCall $2004 ; GetPortRect
; MAKE WINRECT = WINDOW
PushLong #WINRECT ; THE WINDOW RECTANGLE
ToolCall $3A0E ; InvalidRect
; FORCE TASKMASTER TO UPDATE
RTS

**************************************************
PAINT
PushLong PICHNDL
ToolCall $2002 ; HLock
LDA [PICHNDL] ; LONG INDIRECT LOAD
STA PICLOC ; SET THE MEM ADDRESS
LDY #$02
LDA [PICHNDL],Y
STA PICLOC + 2
PushLong #PICBLK ; POINTER TO PARM BLOCK
PushLong #PAINTRECT ; POINTER TO DATA RECT
PushWord #$0000 ; DESTINATION X = 0
PushWord #$0000 ; DESTINATION Y = 0
PushWord #$0000 ; XFER MODE = 'COPY'
ToolCall $D604 ; PPtoPort
; COPY THE PICTURE TO THE WINDOW
PushLong PICHNDL
ToolCall $2202 ; HUnlock
RTS

**********************************************
PENS
PushWord #$0000 ; 0 = UNCHECK
PushWord PEN ; CHANGE CHECK MARK (MENU ITEM #)
ToolCall $320F ; CheckItem (UNCHECK)
PushWord #$FFFF ; BOOLEAN TRUE = CHECK ITEM
PushWord TDATA ; GET MENU ITEM AND PUSH
STA PEN ; SET NEW CURRENT PEN (= MENU ITEM #)
ToolCall $320F ; CheckItem (CHECK)
RTS

*******************************************************************************
COLORS
PushWord #$0000 ; 0 = UNCHECK
PushWord COLOR ; CHANGE CHECK MARK (MENU ITEM #)
ToolCall $320F ; CheckItem (UNCHECK)
Chapter 20

PushWord #$FFFF ; BOOLEAN TRUE = CHECK ITEM
PushWord TDATA ; GET MENU ITEM AND PUSH
STA COLOR ; SET NEW CURRENT PEN (MENU ITEM #)
ToolCall $320F ; CheckItem (CHECK)

RTS

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

MDOWN PushLong #CURSOR ; NEW CURSOR ADDRESS
ToolCall $8E04 ; SetCursor

SETPEN LDA PEN ; (MENU ITEM #)
SEC
SBC #261 ; RESULT = 0, 1, 2
ASL
ASL ; RESULT * 4 = 0, 4, 8
INC ; RESULT = 1, 5, 9
PHA ; PUSH PEN WIDTH
PHA ; PUSH SAME FOR HEIGHT
ToolCall $2C04 ; SetPenSize

SETCOL LDA COLOR ; (MENU ITEM #)
CMP #265 ; BLACK MENU ITEM
BEQ BLACK
CMP #266 ; RED MENU ITEM
BEQ RED

BLUE PushWord #$0004 ; 4 = BLUE
JMP SETCOLOR

RED PushWord #$0007 ; 7 = RED
JMP SETCOLOR

BLACK PushWord #$0000 ; 0 = BLACK

SETCOLOR ToolCall $3704 ; SetSolidPenPat (COLOR)

DRAW PushLong WPTR ; IDENTIFY THE WINDOW
ToolCall $4D0E ; StartDrawing
 ; MUST DO THIS FOR A DRAW
 ; OUTSIDE THE UPDATE ROUTINE
 ; TO RETURN ORIGIN TO ACTUAL
 ; LOCAL COORDINATES

PushLong #YPOS ; TABLE TO REWRITE
ToolCall $0C06 ; GetMouse (LOCAL)

LDA XPOS ; GET X POSN OF MOUSE
STA OLDX ; SAVE ORIG. POSN
LDA YPOS ; GET Y POSN OF MOUSE
STA OLDY ; SAVE THAT TOO . . .
A Drawing Program for the Apple IIgs

STILL
PushWord #$0000 ; SPACE FOR RESULT
PushWord #$0000 ; MOUSE BUTTON (#0)
ToolCall $0E06 ; StillDown
            ; SEE IF MOUSE BUTTON STILL DOWN
PLA
BNE :1
JMP UP ; DONE DRAWING . . .

:1
PushWord OLDX ; X POSN OF MOUSE
PushWord OLDY ; Y POSN OF MOUSE
ToolCall $3A04 ; MoveTo

LOCL
PushLong #YPOS ; ADDR. OF Y AND X DATA
ToolCall $0C06 ; GetMouse (LOCAL)
PushWord XPOS ; X POSN OF MOUSE
PushWord YPOS ; Y POSN OF MOUSE
ToolCall $3C04 ; LineTo

LDA XPOS ; MAKE OLD X = CURRENT X
STA OLDX
LDA YPOS ; OLD Y = CURRENT Y
STA OLDY
JMP STILL ; BACK FOR MORE . . .

UP
ToolCall $9004 ; HideCursor (DON'T COPY THAT!)
PushLong #$0000 ; SPACE FOR RESULT
ToolCall $C704 ; GetClipHandle
PullLong CLIP ; SAVE CLIP REGION HANDLE
PushLong #WINBLK ; POINTER TO WINDOW LOC INFO BLOCK
ToolCall $1E04 ; GetPortInfo
            ; WRITE IT TO WINBLK
PushLong #WINRECT ; POINTER TO WINDOW RECTANGLE
ToolCall $2004 ; GetPortRect
            ; WRITE IT TO WINRECT

LDA WINRECT ; SET PICRECT = WINRECT
STA PICRECT ; SO PaintPixels WILL XFER
LDA WINRECT+2 ; WINDOW AREA TO OUR PICTURE
STA PICRECT+2

PushLong #PAINTBLK ; POINTER TO PARM BLOCK
ToolCall $7F04 ; PaintPixels
            ; COPY NEW IMAGE TO PICTURE IN MEM
Chapter 20

PushWord #$0000 ; X = 0
PushWord #$0000 ; Y = 0
ToolCall $2304 ; SetOrigin
   ; MUST DO THIS FOR A DRAW
   ; OUTSIDE THE UPDATE ROUTINE
   ; TO RETURN ORIGIN TO 0,0

ToolCall $CA04 ; InitCursor
RTS

******************************************
SHOW PushLong WPTR
ToolCall $130E ; ShowWindow
PushWord #257 ; ITEM # FOR 'ERASE'
ToolCall $300F ; EnableItem
PushWord #259 ; ITEM # FOR CLOSE
ToolCall $300F ; EnableItem
PushWord #258 ; ITEM # FOR OPEN
ToolCall $310F ; DisableItem
LDA #MDOWN ; ADDRESS FOR MDOWN ROUTINE
STA C1 ; REWRITE MDOWN VECTOR
RTS

******************************************
HIDE PushLong WPTR
ToolCall $120E ; HideWindow
PushWord #257 ; ITEM # FOR ERASE
ToolCall $310F ; DisableItem
PushWord #259 ; ITEM # FOR CLOSE
ToolCall $310F ; DisableItem
PushWord #258 ; ITEM # FOR OPEN
ToolCall $300F ; EnableItem
LDA #IGNORE ; ADDRESS FOR IGNORE ROUTINE
STA C1 ; REWRITE MDOWN VECTOR
RTS

******************************************
PEN DA #261 ; CURRENT PEN CHOICE (MENU ITEM #)
COLOR DA #265 ; CURRENT COLOR CHOICE (MENU ITEM #)
OLDX DA $0000 ; OLD MOUSE X
OLDY DA $0000 ; OLD MOUSE Y
A Drawing Program for the Apple IIgs

PAINTBLK ADRRL WINBLK ; WINDOW LOCINFO BLOCK
ADRL PICBLK ; PICTURE LOCINFO BLOCK
ADRL WINRECT ; POINTER TO WINDOW RECTANGLE
ADRL PICRECT ; POINTER TO PICTURE POINT
DA $0000 ; XFER MODE (COPY)
CLIP ADRRL $0000 ; CLIP REGION HANDLE

******************************************************************************
CURSOR DA 3 ; CURSOR HEIGHT (3 SLICES)
DA 2 ; CURSOR WIDTH (2 WORDS)
IMAGE HEX 0000,0000 ; LAST WORD MUST ALWAYS BE $0000!
HEX 0f00,0000 ; F + F = BLACK CURSOR
HEX 0000,0000 ; 0 + 0 = NO IMAGE
MASK HEX 0F00,0000 ; F + 0 = WHITE BORDER
HEX FFF0,0000
HEX 0F00,0000
HOTSPOT DA 1 ; Y = 1
DA 1 ; X = 1
******************************************************************************
PICBLK DA $0000 ; 0 = 320 MODE
PICLOC ADRRL $0000 ; POINTER TO PICTURE DATA
DA 160 ; WIDTH OF IMAGE IN BYTES
DA 0,0,200,320 ; RECT OF DATA
PICRECT DA 0,0,200,320 ; RECT OF PIXELS TO XFER
PAINTRECT DA 0,0,200,320 ; RECT OF PIXELS FOR 'PAINT'
PREGION ADRRL $0000 ; STORAGE FOR HANDLE

******************************************************************************
WINBLK DA $0000 ; WILL BE WRITTEN BY GETPORTLOC
WINLOC ADRRL $E12000 ; ADDR. OF SUPER HI-RES SCREEN
DA 160
DA 0,0,0,0 ; MAX SCREEN SIZE
WINRECT DA 0,0,0,0 ; RECT OF PIXELS TO XFER
WREGION ADRRL $0000 ; STORAGE FOR HANDLE

******************************************************************************
CHKSUM CHK ; CHECKSUM FOR VERIFICATION
After you’ve added this to the end of your source listing, make the following changes to the main body of the program itself. As before, each addition will be presented in the context of its surroundings, and we’ll use the introduction of each addition as the opportunity to discuss the related text that you’ve added at the end of the listing.

The first modification is to add a new direct-page pointer, \texttt{PICHNDL}, in the same spot as you added the pointer \texttt{MSSGPTR} for the Event Display program. \texttt{PICHNDL} will be used to dereference the address of our memory block that holds the actual document we’re painting.

\begin{verbatim}
PRODOS EQU $E100A8 ; STD. PRODOS 16 ENTRY
PTR EQU $00 ; OUR OWN DIRECT PAGE PTR ; $00,01,02,03
PICHNDL EQU $04 ; HANDLE OF OUR PICTURE

This program will make use of two new menus, one called \texttt{FILE} that will have the Quit option in it, along with some options for opening and closing the document window, and erasing the document if you want to start over. The second menu is called \texttt{PENS}, and offers choices of pen sizes and colors for drawing. You’ll recall from earlier discussions that additional menus are added by first telling the Menu Manager about them via the \texttt{NewMenu} command, and then adding these menus with \texttt{InsertMenu}. Menus are inserted in the order in which they appear from right to left, so the Apple icon menu will always be defined last, at least using this method. \texttt{InsertMenu} actually lets you insert a menu anywhere, but remember that menu position zero always does the insert at the far left.

\begin{verbatim}
DESK ToolCall $0205 ; DeskStartup
PENMENU PushLong #$0000 ; SPACE FOR RESULT
   PushLong #MENU3 ; ADDR. OF MENU STRUCTURE
   ToolCall $2D0F ; NewMenu
   PullLong MENU3HD ; STORE HANDLE TO 3RD MENU

   PushLong MENU3HD
   PushWord #$0000 ; INSERT AT THIS POSITION (FRONT)
   ToolCall $0D0F ; InsertMenu (ADD TO MENU BAR)

FILE PushLong #$0000 ; SPACE FOR RESULT
   PushLong #MENU2 ; ADDR. MENU STRUCTURE
   ToolCall $2D0F ; NewMenu
   PullLong MENU2HD ; STORE 2ND MENU HANDLE

   PushLong MENU2HD
   PushWord #$0000 ; INSERT AT THIS POSITION (FRONT)
   ToolCall $0D0F ; InsertMenu

APPLE PushLong #$0000 ; SPACE FOR RESULT
\end{verbatim}
\end{verbatim}
These lines show the new menus being defined and inserted just previous to the Apple icon menu. The new menu definitions should be added right after the definition for MENU1 in the shell program, and will look like this:

```
MENU1 ASC '>>@ \N1X',00 ; APPLE MENU
    ASC '--Your Message Here... \N256V',00
    ASC '.
'; END OF MENU

MENU1HD ADRL $0000 ; STORAGE FOR HANDLE

MENU2 ASC '>> File \N2',00 ; FILE MENU
    ASC '--Erase Picture \N257V*Ee',00
    ASC '--Open Window \N258D*Oo',00
    ASC '--Close Window \N259V*Hh',00
    ASC '--Quit \N260*Qq',00
    ASC '.

MENU2HD ADRL $0000

MENU3 ASC '>> Pens \N3',00 ; PENS MENU
    ASC '--Small Pen \N261C',12,00
    ASC '--Medium Pen \N262C ',00
    ASC '--Large Pen \N263C ',00
    ASC '--- \N264D',00
    ASC '--Black \N265C',12,00
    ASC '--Red \N266C ',00
    ASC '--Blue \N267C ',00
    ASC '.

MENU3HD ADRL $0000

The menu items for Small Pen and Black include a startup default check mark, defined with the value $12 after the C special character. Once the menus have been defined and the menu bar has been drawn, we will want to open up the document window. Just prior to that, though, it wouldn't hurt to create the document itself by allocating some memory and clearing it to whatever background color you want to use. A good place to insert the routine to set up a new document would be at the label WIND (for Window) in the SHELL program:

```
DRAWMENU ToolCall $2A0F ; DrawMenuBar
    ; DRAW MENU, LOWER WINDOW

WIND JSR DOCSETUP ; START A NEW DOCUMENT

JSR OPEN ; OPEN A WINDOW, SET DEFAULTS

SHOWCURS ToolCall $9104 ; ShowCursor
```

The instructions for DOCSETUP (Document Setup) have already been entered at the end of the listing. The beginning of DOCSETUP first uses the
Memory Manager call NewHandle to allocate a 32K block of memory equivalent to the super hi-res display screen. Although the actual pixel area is only 32,000 bytes and leaves some unused memory in the block, it is a convenient memory definition. In a more advanced drawing program, you would use the extra memory to store a copy of the SCBs and color tables used for that particular drawing. Since this program has no provision for loading or saving pictures, we won’t have to worry about that.

You’re also not limited to a document size equal to that of the screen. It’s only arbitrary that the window parameter list and the document itself have specified a data area equal to the screen. By adjusting wDataSize in the window parameter list, obtaining more memory for the document, and adjusting the LocInfo parameter block accordingly, we can create a document limited only by the amount of available memory.

Once memory has been obtained, DOCSETUP does a JSR to the ERASE routine. ERASE works by dereferencing the handle to the memory block to determine exactly where in memory the picture is currently located. Before dereferencing a movable block, the block must be temporarily locked to make sure some other application doesn’t force a memory compaction that would move the block while our program was trying to erase it. Locking the block was not a concern in dereferencing the direct page block, because the attributes of the direct page specified the block as unmovable at creation time.

The drawing program could have requested that the document memory be fixed, but this is not a good practice because it unnecessarily jams up memory that other applications may want to use.

Once the block is locked and dereferenced, the CLR (CLeaR) loop puts the value $E in every pixel location. This corresponds to dark gray, but you can use any value you want. Each nibble should have the same value ($5555, $AAAA, $3333, and so on). Once the block is erased, it is unlocked, and we continue with the DOCSETUP routine.

SETREGN is then used to create a rectangular region that will be used by PaintPixels as the Clip Region each time we copy a new line that has been drawn on the screen into the picture in memory. Once the region has been set up, and the handle to it is stored in PREGION (Picture REGION), DOCSETUP is done, and we resume the overall setup portion of our program.

Once everything is set up and the window opened, this brings us to the main event loop of the program. SPECIAL will remain a do-nothing routine, and the JSR SPECIAL could be removed if you were so inclined. In the interest of minimizing changes while you’re debugging, though, we’ll leave it alone.
More Changes

The main change to the beginning of the event loop will be to replace the JSR SHUTDOWN response to the close box with a JSR HIDE:

******************************************
* HANDLE THE EVENT
******************************************

```
EVT JSR SPECIAL ; ANY SPECIAL EVENT HANDLING
CMP #$11 ; MENU EVENT?
BNE :I ; NOPE
JSR DOMENU ; HANDLE IT
JMP MAIN ; BACK FOR MORE
:I
CMP #$16 ; CLOSE BOX
BNE DOCMDS
JSR HIDE ; CLOSE THE WINDOW
JMP MAIN
```

HIDE is the routine that will actually be called when the user selects Close from the File menu or clicks in the close box. HIDE has already been added to the end of the listing, and is fairly simple. There is a Window Manager command, HideWindow ($120E), that makes the specified window invisible. While we’re doing this, however, we’ll want to change the menu choices to disable Close and Erase (since there’s nothing on the screen to close or erase), and enable the choice Open Window. This is done with the Menu Manager commands EnableItem ($300F) and DisableItem ($310F).

Since we can’t draw in a hidden window, the HIDE routine finishes by rewriting the MDOWN (Mouse-DOWN) vector to ignore mouse events. You might think the answer would be to set the TaskMaster mask TMASK, or the event mask used in the actual call to TaskMaster to ignore a mouse-down event. But then mouse-down events would be ignored everywhere—something you probably wouldn’t want.

HIDE, along with the other menu functions, is supported in the DOMENU part of the event loop. In the SHELL program, only one menu item is tested for. In this program, we can set up an indexed indirect command processor to jump to the appropriate routine for each menu choice. This is only possible when each menu item is successively numbered (or at least with minimal skipping of items).
The routines supported in the DOMENU handler are IGNORE, ERASE, SHOW, HIDE, SHUTDOWN, PENS and COLORS. You’ve already seen some of these; let’s look at the new ones.

SHOW is the opposite of the HIDE routine, and calls the Window Manager tool ShowWindow. Like HIDE, the remainder of SHOW then properly enables and disables related menu choices. Erase and Close are enabled, Open is disabled. The mouse-down vector in the command table is also reactivated to use the MDOWN routine, which will be discussed shortly.

For simplicity’s sake, PENS and COLORS both just store the menu item value as the code number for a pen size or color. This makes it easy to change the check mark for each menu item as a new choice is selected. It’s in the actual drawing routine, MDOWN, that converts the current code value for the pen and color to a value usable by the QuickDraw SetPenSize and SetSolidPenPat routines.
Revising UPDATE

The next part of the SHELL program to change is the UPDATE routine that will bring new portions of the document into view as we scroll or resize the window. This addition will be minor, and will consist of a JSR to the PAINT routine that fills in the window.

UPDATE  PHB  ; SAVE OTHER'S DATA BANK
      PHK
      PLB  ; DATA BANK = OURS
      PHD
      LDA  MYDP  ; SAVE THE DIRECT PAGE
      TCD
      JSR  PAINT  ; PAINT THE PICTURE IN THE WINDOW
      PLD  ; RESTORE THE DIRECT PAGE
      PLB  ; RESTORE THE ORIG. DATA BANK
      RTL

The PAINT Routine

PAINT was included in the text added at the end, and looked like this:

PAINT  PushLong PICHNDL
      ToolCall $2002  ; HLock
      LDA  [PICHNDL]  ; LONG INDIRECT LOAD
      STA  PICLOC  ; SET THE MEM ADDRESS
      LDY  #$02
      LDA  [PICHNDL],Y
      STA  PICLOC+2
      PushLong #PICBLK  ; POINTER TO PARM BLOCK
      PushLong #PAINTRECT  ; POINTER TO DATA RECT
      PushWord #$0000  ; DESTINATION X = 0
      PushWord #$0000  ; DESTINATION Y = 0
      PushWord #$0000  ; XFER MODE = 'COPY'
      ToolCall $D604  ; PPToPort
                       ; COPY THE PICTURE TO THE WINDOW
      PushLong PICHNDL
      ToolCall $2202  ; HUnlock
      RTS

The idea here is to first dereference the handle to our document (the picture), and to then store this address in the LocInfo structure for our picture, PICBLK. In addition, a rectangle definition, PAINTRECT has been set up to
equal the entire size of our document. PPToPort is then called to actually transfer the pixels from memory onto the active window. After unlocking the memory block, PAINT then returns to UPDATE, and we're all done repainting the window.

The last change to make to the original SHELL program is to rewrite the mouse-down vector (previously IGNORE) to point to the heart of our drawing program, MDOWN:

```
CMDS DA IGNORE ; NULL EVENT
C1 DA MDOWN   ; MOUSE DOWN
C2 DA IGNORE  ; MOUSE UP
C3 DA IGNORE  ; KEY DOWN
```

The Mouse-Down Drawing Loop
Here's where the real work gets done. MDOWN is called whenever the main event loop determines that the mouse is being held down. If the program were only slightly more refined, it would also check the TaskMaster code on the stack to make sure that the mouse-down was in the content region of the window, but that would preclude a demonstration that will be the grand finale of the book.

Once the mouse-down is detected, the gears start turning and the program gets ready to draw. First, the cursor is changed from an arrow to a very little dot. This is done with `SetCursor ($8E04)`. A cursor definition consists of a data structure that includes the height of the cursor in rows (slices), and the width in words. One of the restrictions to a cursor definition is that the last word of each slice by $0000. A cursor definition consists of two patterns. The first is the image itself, wherein any black portion is indicated with a nibble set to $F$.

To keep the cursor from disappearing on a black background, there is also a mask as part of the cursor record, which includes a larger pattern of $F$'s that create the border. Wherever there is an $F$ in both the image and the mask, the cursor is solid black. Wherever both the image and the mask are $0$, there is no cursor, that is, you can just see whatever's under the cursor. Wherever the cursor image is $0$ and the mask is $F$, a white pixel is shown, which creates the outline to the cursor.

If you want to get really fancy, you can put an $F$ in the mask but leave the corresponding pixel in the image $0$. This will invert whatever is on the screen at that point or pattern, producing a really interesting result. The $0$ and $F$ are also used to specify black and white. If you want a colored cursor, you can use other pixel values as you desire.

Finally, each cursor record also contains a definition of the hot spot for
that cursor. The hot spot is just the part of the cursor that will be assigned to the current mouse position as the mouse is moved around. It's called the hot spot because this also has the effect of being the part of the cursor that counts when you are checking to see if the user clicked on a certain part of the screen, for example in a window control.

With this information, you may wish to experiment with this program in designing new and more creative cursors.

Once the new cursor has been assigned, the routines SETPEN and SETCOL use the store PEN and COLOR values to determine a value to use for the QuickDraw SetPenSize and SetSolidPenPat routines.

Now, although the pen and cursor have been set up, the coordinate system of the window is still set for the QuickDraw internal routines with the origin (upper left corner of the content region) at 0,0. The command StartDrawing fixes this, which is followed by the GetMouse instruction to figure out where the mouse is in terms of the local coordinates of the window. To enable continuous drawing, and to help with some other techniques to be discussed later, we also start two new position variables called OLDX and OLDY. This will allow us, as we get each new mouse position, to know where the previous one was to use as a starting point to draw a line.

The section STILL uses an Event Manager routine called StillDown that returns a zero or nonzero value that tells us whether or not the mouse is still down after the initial mouse-down. On each pass through the STILL(down) loop, we'll draw a line between the last mouse position on the screen and the current one. Only when StillDown fails, which means the user has released the mouse button, do we complete the drawing operation and copy the line just drawn to the document in memory.

The part of code beginning with the local label :1 and continuing to the jump back to STILL executes the actual drawing loop, and continually cycles the mouse position just used into the OLDX and OLDY variables for the next pass through the loop.

This is all pretty straightforward. The tricky part comes when the user releases the mouse button, and the program branches to UP, which then has to copy the new lines into the document in memory.

The first thing UP does is to hide the cursor completely. Otherwise, the image of the cursor would be copied into our picture as well. The remaining block of code then determines the current clip handle and PortLoclnfo for the window's GrafPort, and saves this in CLIP and the storage block WINBLK. There are two Loclnfo structures used in the PaintPixel transfer—one each for the destination and source images. WINBLK is the LocInfo structure for the window; PICBLK is the corresponding LocInfo structure for our picture.
Chapter 20

The next information needed is the actual local coordinate description of
the content rectangle of the window. That is, where do the users think they’re
drawing on the document? \texttt{GetPortRect} ($2004$) will write this into a Rect data
structure for us, and, in the interest of efficiency, I’ve set it up to write directly
into the WINBLK LocInfo structure that will be used in just a moment by
PaintPixels. When that rectangle is returned, I also copy it into the destination
LocInfo structure for PICBLK.

Now the moment of truth. PAINTBLK is the supervisory parameter
block for PaintPixels, and it is already set up to point to WINBLK and PICBLK
as the source and destination LocInfo data structures. It’s actually the PaintPixel
call that does all the work in copying the currently visible part of the window
onto the corresponding portion of our document in memory.

Once that’s completed, SetOrigin is used to return the origin of the win-
dow back to 0,0 for QuickDraw, the cursor is restored to the arrow with
\texttt{InitCursor} ($CA04$), and the routine returns to the main event loop.

One last comment. On first reading, it’s easy to become disoriented with
all the new terms referring to clipping regions, the PortRect, and so on. Try not
to be discouraged, though. The learning phase of any new vocabulary is a
stressful time while your brain sorts out all the new words and concepts.

One way to make your programming easier to to just concentrate on the
terms used in the particular tool call you need to use to accomplish a particular
function. Learn to be a tool browser. Get in the habit of skimming through
your tool reference books to find the particular tool that sounds relevant at a
given moment, and don’t worry about all those other cryptic entries flashing
by, trying to distract you.

\textbf{Trying Out the Drawing Program}

After you’ve made all the above changes, you should get a final checksum of
$05$. However, don’t panic if you get a different number on your first assembly.
If you don’t get this checksum, it’s possible you’ve added an extra space or
character to a menu item, and it may not be a fatal error. Try running your pro-
gram and see if everything works all right.

If you have problems, remember to type E1/C029: 21 when you hear
the BRK bell, and look at the place where the program broke and at the con-
ten of the Accumulator for clues as to what went wrong. Then type Control-Y
to try to return to your program selector. If you do have problems, be sure to
double-check all the uses of PushLong and PushWord in your source file to
make sure they agree with the listings in this book. When those fingers get fly-
ing along, it’s easy to go into autopilot an absent-mindedly type a PushLong
when it should have been PushWord, or vice versa.
A Drawing Program for the Apple IIgs

When the program is running successfully, try drawing with the different pens and colors. Also notice that your drawing isn’t destroyed by closing the window. It’s right there when you open the window again.

You might also notice that even when you press the mouse button outside of the window, the cursor still changes and the routine tries to draw outside the window. Of course, nothing happens because the Window Manager has already established a clipping window that prevents you from drawing outside of the window. This could have been prevented by just checking the Task-Master code returned on the stack to make sure we were in the content region of the window when the mouse-down event occurred. However, leaving it this way lets you explore another issue.

As an experiment, try drawing a circle or box that goes outside the window area, and then use the scroll bars or grow box to make the window bigger to see what happened. What you’ll see is that the only part of your line that was copied into your document was the portion that was drawn within the content portion of the window and then copied to the document with PaintPixels.

This makes sense, but brings up the question: Suppose you wanted to draw a square or an object bigger than the window under program control, perhaps in an object-oriented drawing program? If you’ve ever seen MacPaint or MousePaint for the Apple IIe, you may have noticed that when you try to use an eraser or to fill a background with the paint function, only the visible part of the screen is changed, leaving a big unaffected area for everything offscreen at the time of the action.

One solution is to continually duplicate any drawing action in the visible window, and to also draw directly into the offscreen document. With this approach, you should be able to uncover and view an entire shape, even if it was drawn with the cursor outside the active window. Of course, a real program wouldn’t let the user do this, but the technique used will illustrate how you could do any QuickDraw command into a document that was not entirely visible through an active window.

The Better Sketcher Program

Save the source listing for SIMP.SKTCH once it’s all working, and then save a duplicate under the new name BETTER.SKTCH. The only part of the program that needs to be changed is the MDOWN drawing routine.

To do this, rewrite the MDOWN routine starting at LOCL so that it looks like this:

```
LOCL
PushLong #YPOS ; ADDR. OF Y AND X DATA
ToolCall $0C06 ; GetMouse (LOCAL)
```
Chapter 20

```assembly
PushWord XPOS ; X POSN OF MOUSE
PushWord YPOS ; Y POSN OF MOUSE
ToolCall $3C04 ; LineTo

OFFSCRN PushLong #WINBLK ; POINTER TO WINDOW LOC INFO BLOCK
ToolCall $1E04 ; GetPortLocInfo
             ; WRITE IT TO WINBLK

SAVEW PushLong #WINRECT ; POINTER TO WINDOW RECTANGLE
ToolCall $2004 ; GetPortRect
             ; WRITE IT TO WINRECT
PushLong #$0000 ; SPACE FOR RESULT
ToolCall $C904 ; GetVisHandle
PullLong WREGION ; SAVE HANDLE TO WINDOW VIS REGION

SETOFF PushLong PREGION ; PICTURE VIS REGION
ToolCall $C804 ; SetVisHandle
             ; GET VIS REGION FOR ENTIRE PICTURE
PushLong #PICBLK
ToolCall $1D04 ; SetPortLoc
             ; SET PORT LOC TO PICTURE
PushLong #PICRECT
ToolCall $1F04 ; SetPortRect
             ; PORT = ENTIRE PICTURE

DRAWOFF PushWord OLDX ; X POSN OF MOUSE
PushWord OLDY ; Y POSN OF MOUSE
ToolCall $3A04 ; MoveTo
PushWord XPOS ; X POSN OF MOUSE
PushWord YPOS ; Y POSN OF MOUSE
ToolCall $3C04 ; LineTo

SETWIN PushLong WREGION ; GET WINDOW VIS REGION HANDLE
ToolCall $C804 ; SetVisHandle
             ; MAKE VIS REGION WINDOW AGAIN
PushLong #WINBLK
ToolCall $1D04 ; SetPortLoc
             ; SET PORT LOC BACK TO WINDOW
PushLong #WINRECT
ToolCall $1F04 ; SetPortRect
             ; PORT = ACTIVE WINDOW

LDA XPOS ; MAKE OLD X = CURRENT X
STA OLDX
LDA YPOS ; OLD Y = CURRENT Y
STA OLDY
```

468
This revised routine changes things by adding another step to the drawing process. As each line segment is drawn from OLDX, OLDY to XPOS, YPOS in the main window, a repeat performance is made after switching the GrafPort LocInfo pointer to our offscreen document. Let's look at the actual program steps to see how this is done.

The offscreen portion begins at the label OFFSCRN, where the current GrafPort LocInfo data is written to our data block WINBLK. This stores the current window's address information (this will make a lot more sense if you review the listing while you read each reference). SAVEW then uses GetPortRect to get the local coordinates of the window into WINRECT and the VissRgn clipping region handle into WREGION. These are saved because these are precisely the GrafPort attributes we are about to change to get the next drawing command to draw directly into our document.

SETOFF initiates this change-over by setting the VissRgn for the GrafPort to the rectangular region created for our document when it was first created. That region definition was stored in the handle PREGION. It then uses SetPortLoc to switch the drawing area to the part of memory designated by the LocInfo structure PICBLK. Finally, the PortRect is set to the entire document size of 200 X 320 pixels. This guarantees there will be no clipping as long as the drawing action is still within our document boundaries.

The actual offscreen drawing takes place starting at DRAWOFF, and only takes a few instructions.

SETWIN then starts the process of restoring the GrafPort back to the window on the display screen. The VissRgn is restored using the stored handle in WREGION; the PortLocInfo parameters are restored with the data stored in WINBLK, and finally the PortRect is restored to that of the open window using the values stored in WINRECT.

Because the drawing takes place in both the window and the offscreen document almost simultaneously, the UP routine no longer needs to do any
pixel copying, and so only needs to restore the origin and the cursor before returning to the event loop.

This is not the only approach to drawing offscreen, and probably not even the best one. The QuickDraw call OpenPort is designed specifically for opening many GrafPorts, none of which are required to be visible on the screen. The techniques shown here were chosen to give you a hands-on feel for what actually happens at the lower levels with various regions and data structures when you use commands like OpenPort and other Apple II GS tools.

**Conclusion**

Well, here we are at last. Even though there is a lot to learn on the Apple II GS, I hope you've found that in the end, everything really can be reduced to very simple elements.

The information presented here is only the beginning of the process of discovering the many facets of the Apple II GS, and assembly language programming in general.

The real challenge begins now. In the end, a teacher only shows students what they were on the verge of discovering for themselves. As a facilitator to your own path of discovery, I hope you'll continue the process you've begun in this book and use each demonstration program as a starting point for many other explorations and experiments. Not every answer has to come out of a book, and now is as good a time as any to begin to discover your own secrets to the Apple II GS.

It is the very fact that a computer has no defined purpose that makes it the ideal vehicle for the human imagination. You now have the ability and tools to make your Apple II GS do what you want it to do. You're no longer limited to programs "off the shelf," someone else's preconceived notion of what the computer can do.

Remember that the most important person to please in writing programs is yourself, and it is your own quest for challenges and accomplishments that determine the value of the computer in your life. I hope this book has helped influence that process and helped nurture the seeds that will grow into the future reality of computers for everyone.
Appendices
Appendix A

65816 Instructions

This section will provide you with a quick reference for various 65816 instructions.

Although the terms native and emulation are used for the 16- and 8-bit modes of the 65816, it probably would have been better if the e bit had stood for enable bit, in regard to enabling the options of the 16-bit modes. The term emulation implies that the 65816 is somehow limited when in the emulation mode to a subset of the 65816 that emulates the 6502, and that many of the 65816 instructions are not available when in the emulation mode. This is not true, and you should not forget that even in the emulation mode, you have the full complement of 65816 instructions available to you. The main limitations in the emulation mode are simply in register size and the stack size and location.

Most instructions operate on either a byte or word (2 bytes) depending on the status of the e, m, and x bits.

For Accumulator and Memory operations:
- e = 1 (emulation): byte operation (m and x not applicable)
- e = 0 (native), m = 0: word operation
- e = 0 (native), m = 1: byte operation

For Index Register operations:
- e = 1 (emulation): byte operation (m and x not applicable)
- e = 0 (native), x = 0: word operation
- e = 0 (native), x = 1: byte operation

Because the relevance of the m and x bits are directly dependent on the status of the e bit, individual instructions may be commented with a phrase like “How many bytes are transferred is dependent on the condition of the m bit.” Obviously the e bit is also a factor, but because the m bit can only be conditioned if e = 0, the mention of it would be redundant. There are a few instructions, notably the transfer commands like TAX, that involve both the m and x bits, and where some question may exist about the effect of the e bit, so appropriate discussion of all the status bits is included.
Appendix A

Although many of the example code segments use an 8-bit mode in the interest of clarity, it should be easy for you to anticipate the 16-bit version. Most actions with softswitches, hardware registers, and ASCII data are only done in the 8-bit mode, and many of the best illustrations of the uses of instructions like CMP and BIT are in reference to these operations. Remember that the 8-bit mode for a particular instruction doesn’t require that the microprocessor be in the emulation mode (e = 1), only that the m or x bit be set for that particular instruction as appropriate.

For each instruction, the addressing modes are indicated. Note that under Common Syntax, lowercase letters are used within each example, like this:

```
Absolute Long
ADC $00FFff 6F ff FF 00
```

These are used only to clarify which byte of a word is allocated at a given position in the final assembly. You would not normally use lowercase letters in the operand of an instruction, although nothing specifically prohibits you from doing so.

For some examples, binary numbers are indicated in the form

```
%01100100
```

This is supported in the Merlin assembler, but may not be in the particular assembler you’re using. This form of the operand is not required for any particular instruction and is used only in the interest of clearly showing the binary operation in any particular example.

Summary of Addressing Modes

Although the various addressing modes available on the 65816 are covered in greater detail in the main body of this book, here is an exampled summary of some of the modes available.

```
Absolute
LDA $1234
```

Loads one or two bytes from locations $1234 (low byte), $1235 (high byte) in the current data bank.

```
Absolute Long
LDA $FF1234
```

Loads one or two bytes from locations $1234 (low byte), $1235 (high byte) in bank $FF, regardless of the current data bank setting.
Summary of Addressing Modes

Direct Page

**LDA $12**

Loads one or two bytes from relative positions $12 (low byte) and $13 (high byte) on the direct page. Remember the base address of the direct page depends on the setting of the direct page register, but this need not be considered within the context of any particular instruction.

Direct Page Indirect

**LDA ($12)**

A 16-bit address is determined by examining the contents of bytes $12 and $13 (low byte, high byte) on the direct page. The resulting address in the current data bank is then used as the source of the data to be loaded into the Accumulator. It's only at that point that the condition of the \( m \) bit is taken into account and will affect how many bytes are loaded into the Accumulator.

Direct Page Indirect Long

**LDA [$12]**

A 24-bit address is determined by examining the contents of bytes $12, $13 and $14 (low byte, high byte, bank byte) on the direct page. The resulting address is then used as the source of the data to be loaded into the Accumulator. It's only at that point that the condition of the \( m \) bit is taken into account and will affect how many bytes are loaded into the Accumulator.

Immediate

**LDA #$12**

The value specified, in this example, $12, is loaded into the Accumulator. If \( m = 0 \) (16 bits), two bytes will be loaded ($0012).

Absolute Indexed,X

**LDA $1234,X**

Absolute Indexed,Y

**LDA $1234,Y**

A 16-bit address is determined by adding the contents of the X or Y Register to the base address specified in the current data bank. The resulting address is then used as the source of the data to be loaded into the Accumulator. If the calculated address exceeds the end of the current data bank, data will be accessed in the next bank of memory. This allows data blocks to cross bank boundaries. One or two bytes will then be loaded into the Accumulator depending on the condition of the \( m \) bit. If the X or Y Registers held the value $05, then the examples above would load the data from $1239, $123A.
Appendix A

**Absolute Long Indexed, X**  
**LDA $FF1234,X**

A 16-bit address is determined by adding the contents of the X Register to the base address specified. The resulting address is then used as the source of the data to be loaded into the Accumulator. If the calculated address exceeds the end of the base address bank, data will be accessed in the next bank of memory. This allows data blocks to cross bank boundaries. One or two bytes will then be loaded into the Accumulator depending on the condition of the $m$ bit. If the X Register held the value $05$, then the example above would load the data from $FF1239, FF123A$.

**Direct Page Indexed, X**  
**LDA $12,X**

Loads one or two bytes from position $12$ plus the contents of the X Register on the direct page. Thus, if the X Register contained $05$, the bytes would be loaded from locations $17, 18$. Remember the base address of the direct page depends on the setting of the direct page register, but this need not be considered within the context of any particular instruction.

**Direct Page Indexed Indirect, X**  
**LDA ($12,X)**

A 16-bit address in bank zero is determined by adding the contents of the X Register to the specified direct-page address. One or two bytes are then loaded from the resulting relative address (low byte, high byte) on the direct page. Remember that the base address of the direct page depends on the setting of the direct-page register, but this need not be considered within the context of any particular instruction. If the X Register held the value $05$, then the example above would first use the contents of locations $17, 18$ to determine the target address. If $17, 18$ pointed to location $1000$, the data would then be loaded from $1000, 1001$.

**Direct Page Indirect Indexed, Y**  
**LDA ($12,Y)**

A 16-bit address is determined by examining the contents of bytes $12$ and $13$ (low byte, high byte) on the direct page. The contents of the Y Register are then added to the address in $12, 13$, and the resulting address is then used as the source of the data to be loaded into the Accumulator. If the calculated address exceeds the end of the current data bank, data will be accessed in the next bank of memory. This allows data blocks to cross bank boundaries. It is only at that point that the condition of the $m$ bit is taken into account and will affect how many bytes are loaded into the Accumulator. If the Y Register contained $05$, and locations $12, 13$ pointed to location $1000$, the bytes would be loaded from locations $1005, 1006$. 

476
Summary of Addressing Modes

**Direct Page Indirect Long Indexed, Y**  \[ \text{LDA } [\$12],Y \]

A 24-bit address is determined by examining the contents of bytes $12, $13 and $14 (low byte, high byte, bank byte) on the direct page. The resulting address is then added to the contents of the Y Register, and the result is used as the source of the data to be loaded into the Accumulator. If the calculated address exceeds the end of the base address bank, data will be accessed in the next bank of memory. This allows data blocks to cross bank boundaries. It is only at that point that the condition of the \( m \) bit is taken into account and will affect how many bytes are loaded into the Accumulator. If $12, $13, and $14 pointed to $FF1000, and the Y Register held $05, the data would be loaded from locations $FF1005, $FF1006.

**Stack Relative**  \[ \text{LDA } \$12,S \]

This instruction adds the value in the operand ($12) to the current Stack Pointer value and loads the Accumulator with one or two bytes from the resulting address in the current data bank. Because the Stack Pointer always points to the next available position on the stack, which is empty, the first meaningful byte of data is always found at 1,S.

**Stack Relative Indirect Indexed, Y**  \[ \text{LDA } ([12,S]),Y \]

This is similar to the Stack Relative addressing mode, but instead uses the contents of ($12,S) as a two-byte pointer to an address in the current data bank. The contents of the Y Register are then added to this address to determine the actual locations from which one or two bytes will be loaded into the Accumulator. If the calculated address exceeds the end of the current data bank, data will be accessed in the next bank of memory. This allows data blocks to cross bank boundaries.

**Absolute Indexed Indirect**  \[ \text{JSR } (1234,X) \]

This instruction uses the contents of locations $1234, $1235 (low byte, high byte) as a base address to which the contents of the X Register are added. The resulting address in the current program bank is used as the target for the JSR.
ADC

ADC: Add with Carry

Description
This instruction adds the contents of a memory location or immediate value to the contents of the Accumulator, plus the carry bit, if it was set. The result is put back in the Accumulator. ADC works for both the binary and BCD modes.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>ADC $FFff</td>
<td>6D ff FF</td>
</tr>
<tr>
<td>Absolute Long</td>
<td>ADC $00FFff</td>
<td>6F ff FF 00</td>
</tr>
<tr>
<td>Direct Page</td>
<td>ADC $FF</td>
<td>65 FF</td>
</tr>
<tr>
<td>Direct Page Indirect</td>
<td>ADC ($FF)</td>
<td>72 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Long</td>
<td>ADC [$FF]</td>
<td>67 FF</td>
</tr>
<tr>
<td>Immediate</td>
<td>ADC #$FF</td>
<td>69 FF</td>
</tr>
<tr>
<td>Absolute Indexed,X</td>
<td>ADC $FFff,X</td>
<td>7D ff FF</td>
</tr>
<tr>
<td>Absolute Long Indexed,X</td>
<td>ADC $00FFff,X</td>
<td>7F ff FF 00</td>
</tr>
<tr>
<td>Absolute Indexed,Y</td>
<td>ADC $FFff,Y</td>
<td>79 ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed,X</td>
<td>ADC $FF,X</td>
<td>75 FF</td>
</tr>
<tr>
<td>Direct Page Indexed Indirect,X</td>
<td>ADC ($FF,X)</td>
<td>61 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Indexed,Y</td>
<td>ADC ($FF),Y</td>
<td>71 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Long Indexed,Y</td>
<td>ADC [$FF],Y</td>
<td>77 FF</td>
</tr>
<tr>
<td>Stack Relative</td>
<td>ADC $FF,S</td>
<td>63 FF</td>
</tr>
<tr>
<td>Stack Relative Indirect Indexed,Y</td>
<td>ADC ($FF,S),Y</td>
<td>73 FF</td>
</tr>
</tbody>
</table>

Uses
Peculiarly enough, ADC is most often used to add numbers together. Here are some common examples:

Adding a constant to the Accumulator or memory location:

```
CLC ; GET READY TO ADD
LDA MEM ; GET FIRST VALUE (BYTE OR WORD)
ADC #$80 ; ADD #$80
STA RSLT ; STORE RESULT

(RSLT = MEM + #$80)
```
Adding a constant (such as an offset) to a 2-byte (or 2-word) memory value:

<table>
<thead>
<tr>
<th>2-Byte Form</th>
<th>2-Word Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLC</td>
<td>CLC</td>
</tr>
<tr>
<td>LDA MEM</td>
<td>LDA MEM</td>
</tr>
<tr>
<td>ADC #$80</td>
<td>ADC #$80</td>
</tr>
<tr>
<td>STA MEM</td>
<td>STA MEM</td>
</tr>
<tr>
<td>LDA MEM+1</td>
<td>LDA MEM+2</td>
</tr>
<tr>
<td>ADC #$00</td>
<td>ADC #$00</td>
</tr>
<tr>
<td>STA MEM+1</td>
<td>STA MEM+2</td>
</tr>
</tbody>
</table>

\[(MEM, MEM+1(2) = MEM, MEM+1(2) + #$80)\]

Adding two 2-byte (or word) values together:

<table>
<thead>
<tr>
<th>2-Byte Form</th>
<th>2-Word Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLC</td>
<td>CLC</td>
</tr>
<tr>
<td>LDA MEM</td>
<td>LDA MEM</td>
</tr>
<tr>
<td>ADC MEM2</td>
<td>ADC MEM2</td>
</tr>
<tr>
<td>STA MEM</td>
<td>STA MEM</td>
</tr>
<tr>
<td>LDA MEM+1</td>
<td>LDA MEM+2</td>
</tr>
<tr>
<td>ADC MEM2</td>
<td>ADC MEM2</td>
</tr>
<tr>
<td>STA MEM+1</td>
<td>STA MEM+2</td>
</tr>
</tbody>
</table>

\[(MEM, MEM+1(2) = MEM, MEM+1(2) + #$80)\]
**AND: Logical AND**

**Description**
This instruction takes each bit of the Accumulator and performs a logical AND with each corresponding bit of the specified memory location or immediate value. The result is put back in the Accumulator. The memory location specified is unaffected. (See ORA also.)

The truth table used is:

<table>
<thead>
<tr>
<th>AND</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Example:**

| Accumulator: | 0 0 1 0 0 1 1 |
| Memory:      | 0 1 0 1 0 1 0 1 |
| Result:      | 0 0 0 1 0 0 0 1 |

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>AND $FFff</td>
<td>2D ff FF</td>
</tr>
<tr>
<td>Absolute Long</td>
<td>AND $00FFff</td>
<td>2F ff FF 00</td>
</tr>
<tr>
<td>Direct Page</td>
<td>AND $FF</td>
<td>25 FF</td>
</tr>
<tr>
<td>Direct Page Indirect</td>
<td>AND ($FF)</td>
<td>32 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Long</td>
<td>AND [$FF]</td>
<td>27 FF</td>
</tr>
<tr>
<td>Immediate</td>
<td>AND #$FF</td>
<td>29 FF</td>
</tr>
<tr>
<td>Absolute Indexed,X</td>
<td>AND $FFff,X</td>
<td>3D ff FF</td>
</tr>
<tr>
<td>Absolute Long Indexed,X</td>
<td>AND $00FFff,X</td>
<td>3F ff FF 00</td>
</tr>
<tr>
<td>Absolute Indexed,Y</td>
<td>AND $FFff,Y</td>
<td>39 ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed,X</td>
<td>AND $FF,X</td>
<td>35 FF</td>
</tr>
<tr>
<td>Direct Page Indexed Indirect</td>
<td>AND ($FF,X)</td>
<td>21 FF</td>
</tr>
<tr>
<td>Direct Page Indexed Indirect,Y</td>
<td>AND ($FF),Y</td>
<td>31 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Long Indexed</td>
<td>AND [$FF],Y</td>
<td>37 FF</td>
</tr>
<tr>
<td>Stack Relative</td>
<td>AND $FF,S</td>
<td>23 FF</td>
</tr>
<tr>
<td>Stack Relative Indexed,Y</td>
<td>AND ($FF,S),Y</td>
<td>33 FF</td>
</tr>
</tbody>
</table>

**Uses**

AND is used primarily as a mask, that is, to let only certain bit patterns through a section of a program. The mask is created by putting 1's in each bit position where data is to be allowed through, and 0's where data is to be suppressed.
For example, it’s frequently desirable to mask out the high-order bit of ASCII data, such as would come from the keyboard or another input device (like a disk file). The routine shown assures that no matter what value is gotten from DEVICE, the high-order bit of the value put in MEM will always be clear:

<table>
<thead>
<tr>
<th>Routine:</th>
<th>Sample Input:</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA DEVICE</td>
<td>Accumulator: 01010111 or 11010111</td>
</tr>
<tr>
<td>AND #$7F</td>
<td>#$7F</td>
</tr>
<tr>
<td>STA MEM</td>
<td>01111111 01111111</td>
</tr>
</tbody>
</table>

As an example, when reading the keyboard directly, without the benefit of the Event Manager (8-bit mode, usually in Applesoft BASIC or ProDOS 8), it is necessary to clear the high-order bit (bit 7). Here is the way this is usually done:

```
WATCH  LDA  KYBD     ; $C000
      BPL  WATCH     ; AGAIN IF <$80
      BIT  STROBE    ; CLEAR STROBE: $C010
      AND  #$7F      ; CLR HIGH BIT
      STA  MEM
```

Furthermore, because the only difference between upper- and lowercase characters is that lowercase characters have bit 5 set, AND can be used to normalize input to all uppercase:

```
LDA  CHAR      ; INPUT CHARACTER
AND  #$DF      ; %1101 1111 (LC -> UC)
STA  CHAR      ; CONVERTS LOWERCASE TO UPPER.
```

Another way of looking at this same effect is to say that AND can be used to force a zero in any desired position in the bit pattern of a byte or word. (See ORA to force 1’s). A zero is put in the mask value at the positions to be forced to 0, and all remaining positions are set to 1. Whenever a data byte is ANDed with this mask, a zero will be forced at each position marked with a zero in the mask, while all other positions will be unaffected, remaining 0’s or 1’s as was their original condition.

In graphics operations, AND has the effect of creating the result of the intersection of two other images. For example:

```
00011000 00000000 00000000
00011000 00000000 00000000
00011000 00000000 00000000
00011000 AND 11111111 = 00011000
00011000 11111111 00011000
00011000 00000000 00000000
00011000 00000000 00000000
00011000 00000000 00000000
```
AND

This is the underlying logical function behind the mask used in the cursor mask or a clipping region.

There are also rather obscure uses for the AND instruction. The first of these is to do the equivalent of a MOD function, involving a piece of data and a power of two. You'll recall that the MOD function produces the remainder in a division operation. For example: 12 MOD 4 = 0; 14 MOD 4 = 2; 18 MOD 4 = 2; 17 MOD 2 = 1; and so on. The general formula is:

\[
\text{Acc. MOD } 2^n = \text{RESULT}
\]

The actual operation is carried out by using a value of \((2^n - 1)\) as the mask value. The theory of operation is that only the last \(n\) bits of the data byte are let through, thus producing the result corresponding to a MOD function.

Example:

\[
\begin{align*}
\text{LDA} & \quad \text{MEM} \\
\text{AND} & \quad \$07 \\
\text{STA} & \quad \text{MEM}
\end{align*}
\]

This technique provides one of several ways of testing for the odd/even attribute of a number:

\[
\begin{align*}
\text{LDA} & \quad \text{MEM} \\
\text{AND} & \quad \$01 \\
\text{BEQ} & \quad \text{EVEN} \\
\text{BNE} & \quad \text{ODD}
\end{align*}
\]

Note: $\text{BIT}$ (described later) can be used to test for one or more matches, but the AND technique described here confirms that all the bits of interest match.
ASL: Arithmetic Shift Left

Description
This instruction moves each bit of the Accumulator or memory location specified one position to the left. A zero is forced at the bit 0 position, and the high-order bit of the byte or word (bit 7 for a byte, bit 15 for a word) falls into the carry. The result is left in the Accumulator or memory location.

(See also ROL, also LSR and ROR.)

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ASL
(Arithmetic Shift Left)

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulator</td>
<td>ASL</td>
<td>0A</td>
</tr>
<tr>
<td>Absolute</td>
<td>ASL $FF$ff</td>
<td>0E ff FF</td>
</tr>
<tr>
<td>Direct Page</td>
<td>ASL $FF$</td>
<td>06 FF</td>
</tr>
<tr>
<td>Absolute Indexed,X</td>
<td>ASL $FFFF$,$X$</td>
<td>1E ff FF</td>
</tr>
<tr>
<td>Direct Page,X</td>
<td>ASL $FF$,$X$</td>
<td>16 FF</td>
</tr>
</tbody>
</table>

Uses
The most common use of ASL is for multiplying by a power of two. You are already familiar with the effect in base ten: 123 * 10 = 1230 (shift left). For example:

```
LDA MEM
ASL
ASL ; TIMES 2
ASL ; TIMES 2 AGAIN 2
STA MEM ; MEM = MEM * 4 (4 = 2 )
```

ASL can also be used in a loop to successively test each bit of a byte or word.
ASL

LDX #$07 ; 7 FOR A BYTE, 15 FOR A WORD
PHA ; SAVE VALUE

LOOP PLA ; RETRIEVE CURRENT VALUE
ASL ; SHIFT HIGH-ORDER BIT INTO CARRY
PHA ; SAVE SHIFTED VALUE
BCC CLEAR ; BIT X CLEAR
BCS SET ; BIT X SET

CLEAR NOP ; YOUR HANDLER HERE
SET NOP ; YOUR HANDLER HERE
DEX ; X = X - 1 FOR NEXT BIT
BPL LOOP ; ANOTHER CYCLE
BCC: Branch Carry Clear

Description
Executes a branch if the carry flag is clear. Ignored if carry is set. Many assemblers have an equivalent mnemonic, BLT (Branch Less Than, not to be confused with the sandwich), since BCC is often used immediately following a comparison to see if the Accumulator held a value less than the specified value. BCC is limited to a relative branch distance of $-128$ to $+127$ bytes. (See also BCS.)

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative Only</td>
<td>BCC LABEL</td>
<td>6D ff</td>
</tr>
<tr>
<td>or:</td>
<td>BCC $FFFF</td>
<td>6D ff</td>
</tr>
</tbody>
</table>

Uses
As mentioned, BCC is used to detect when the Accumulator is less than a specified value. The usual appearance of the code is listed below. Note that in a two-byte or two-word comparison, the high-order bytes/words are checked first. (See also BCS.)

One-Byte (Word) Comparison

<table>
<thead>
<tr>
<th>ENTRY</th>
<th>LDA MEM</th>
<th>CMP MEM2</th>
<th>BCC LESS</th>
<th>BCS EQ/GRTR</th>
</tr>
</thead>
</table>

Two-Byte (Word) Comparison

<table>
<thead>
<tr>
<th>ENTRY</th>
<th>LDA MEM+1(2)</th>
<th>CMP MEM2+1(2)</th>
<th>BCC LESS</th>
<th>BEQ CHK2</th>
<th>BCS GRTR</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>CHK2</th>
<th>LDA MEM</th>
<th>CMP MEM2</th>
<th>BCC LESS</th>
<th>BCS EQ/GRTR</th>
</tr>
</thead>
</table>

Goes to LESS if MEM < MEM2)
Goes to LESS only if MEM,MEM+1(2) < MEM2,MEM2+1(2)
BCS: Branch Carry Set

Description
Executes the branch only if the carry flag is set. Some assemblers support the mnemonic BGT (for Branch Greater Than), since this command is used to test for the Accumulator equal to or greater than the specified value. BCS is limited to a relative branch distance of $-128$ to $+127$ bytes.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

Addressing Modes Available

Mode
Relative Only

Common Syntax
BCS LABEL

Hex Coding
B0 ff

Uses
Used to detect Accumulator equal to or greater than a specified value. Can be combined with BEQ to detect a greater-than relationship. Note that in the 2-byte or 2-word comparison, the high-order bytes/words are checked first.

One-Byte (Word) Comparison

ENTRY LDA MEM
CMP MEM2
BCC LESS
BEQ EQUAL
BCS GREATER

Goes to GREATER if
MEM > MEM2, or EQUAL
if MEM = MEM2

Two-Byte (Word) Comparison

ENTRY LDA MEM + 1(2)
CMP MEM2 + 1(2)
BCC LESS
BEQ CHK2
BCS GREATER

CHK2 LDA MEM
CMP MEM2
BCC LESS
BEQ EQUAL
BCS GREATER

Goes to GREATER only if
MEM, MEM + 1(2) > MEM2, MEM2 + 1(2), or
to EQUAL if MEM, MEM + 1 = MEM2, MEM2 + 1
**BEQ: Branch if Equal**

**Description**
Executes a branch if the Z flag (zero flag) is set, indicating that the result of a previous operation was zero. See BCS to see how a comparison for the Accumulator equal to a value is done. BEQ is limited to a relative branch distance of −128 to +127 bytes.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

**Addressing Modes Available**

- **Mode**: Relative Only
- **Common Syntax**: BEQ LABEL
- **Hex Coding**: F0 ff
  or: BEQ $FFFF F0 ff

**Uses**
In addition to being used in conjunction with compare operations, BEQ is used to test whether the result of a variety of other operations has resulted in a value of zero. The common classes of these operations are increment/decrement, logical operators, shifts, and register loads. Even easier to remember is the general principle that whenever you've done something that results in zero, chances are good the Z flag has been set. Likewise, any nonzero result of an operation is likely to clear the Z flag.

BEQ is used to check for the end of a string by using zero as a delimiter.

```assembly
LDX #$00 ; BEGINNING OF DATA
LOOP LDA DATA,X ; GET A CHARACTER (8 BIT)
BEQ DONE ; CHAR = 0 = END OF STRING
WORK NOP ; YOUR PROGRAM HERE
INX ; NEXT CHARACTER IN STRING
JMP LOOP
DONE RTS
```

BEQ can be used to terminate a loop by waiting for the counter to reach zero. (See BNE also.) When BEQ follows the decrement instruction at the end of the loop, the counter is initialized with the value for the number of times for the loop to execute.

```assembly
LDX #25 ; LOOP 25 TIMES
LOOP JSR ROUTINE ; DO SOMETHING
DEX ; COUNT DOWN
BEQ DONE ; X = 0 = FINISHED
JMP LOOP ; BACK FOR MORE
```
BIT: Compare Accumulator Bits with Contents of Memory

Description
Performs a logical AND on the bits of the Accumulator and the contents of the memory location. The result of the AND will be zero or nonzero, and the Z flag is conditioned accordingly. What this means is that if any bits set in the Accumulator happen to match any set in the value specified, the Z flag will be cleared. If no match is found, it will be set. BNE is used to detect a match, BEQ detects a no-match condition.

Fully understanding the function and various applications of this instruction is a sign of having arrived as an assembly language programmer, and suggests you are probably the hit of parties, thrilling your friends by doing hex arithmetic in your head and reciting ASCII codes on command.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

m7, m6 for byte operations,
m15, m14 for word operations.

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>BIT $FFff</td>
<td>2C ff FF</td>
</tr>
<tr>
<td>Direct Page</td>
<td>BIT $FF</td>
<td>24 FF</td>
</tr>
<tr>
<td>Immediate</td>
<td>BIT #$FF</td>
<td>89 FF</td>
</tr>
<tr>
<td>Absolute Indexed, X</td>
<td>BIT $FFff,X</td>
<td>3C ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed, X</td>
<td>BIT $FF,X</td>
<td>34 FF</td>
</tr>
</tbody>
</table>

Uses
BIT provides a means of testing whether a given bit is on in a byte of data. Important: BIT will only indicate that at least one of the bits in question match. It does not indicate how many actually do match. See the AND instruction on how to do a check for all matching. The mask is created by setting 1's in the bit positions you are interested in, and leaving all remaining positions set to 0. Thus, if you are only interested in testing a specific bit, your mask should only have that bit set. BIT is almost always used in conjunction with branch instructions including BMI, BPL, BEQ and BNE. Be sure to look over the descriptions of these instructions as well.
Examples:

Showing the results of the BIT operation:

<table>
<thead>
<tr>
<th>Acc: 1 0 0 1 1 0 1 1</th>
<th>Mem: 0 1 0 1 0 1 0 1</th>
<th>Z flag effect:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result: 0 0 0 1 0 0 0 1 + 1</td>
<td>clr BNE works</td>
<td>BEQ not taken</td>
</tr>
</tbody>
</table>

Status Register

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

m7 m6

Acc: 1 0 0 1 1 0 1 1
Mem: 0 1 0 0 0 1 0 0
Z flag effect:

Result: 0 0 0 0 0 0 0 0 - 0 set BEQ works
        BNE not taken

Status Register

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

m7 m6

Testing a given bit:

Test Acc. for bit 4 on
ENT LDA DEVICE
    BIT #$10 ; %00010000
    BNE MATCH
    BEQ NOMATCH

Test memory for bit 4 on
ENT LDA #$10 ; %00010000
    BIT MEM
    BNE MATCH
    BEQ NOMATCH

Also important is the fact that a BIT instruction does not change the contents of the Accumulator. Here is an example of an 8-bit mode program that waits for a keypress while maintaining a unique value in the Accumulator.

LDA #$41 ; "A"
LOOP LDA #$41 ; "A"
    BIT KYBD ; $C000
    BPL LOOP ; VAL < 128 = NO PRESS
    BIT STROBE ; $C010
    JSR COUT ; PRINTS LETTER "A"
DONE RTS

489
BIT

Notice that in this example, no data is actually retrieved from the keyboard. Only a wait is done until the keypress.

The BIT STROBE step in the previous example also provides an illustration of another application of BIT, which is to access a hardware location (often called a softswitch) without changing the contents of the Accumulator.

**Important:** Many ProDOS 8 and Applesoft BASIC routines set softswitches using the BIT instruction. For example, the instruction BIT $C050 enables the graphics display, BIT $C051 switches to text. However, this only works in the 8-bit mode. This is because in the 16-bit mode, the instruction BIT $C050 accesses both $C050 and $C051 (remember, it is then a 2-byte operation). The result is that $C051 is accessed last, and the final state is the text display, which is probably not what you intended. BIT instructions of softswitches should only be done in the 8-bit mode.

BIT also sets the N and V flags, and thus provides a very fast way of testing bits 6 and 7 of a byte or bits 14 and 15 of a word. When examining hardware registers, BIT can be combined with BMI to test for a high bit set. For example, in the Apple IIgs, location $C036 uses the high-order bit to indicate whether the system speed is configured to the normal or fast modes. BIT and BMI can be used to easily test this (in the 8-bit mode, of course).

```assembly
TEST BIT $C036 ; CONFIGURATION REGISTER
   BMI FAST ; BIT 7 = 1 = FAST MODE
   BPL SLOW ; BIT 6 = 0 = SLOW MODE
```

Some applications use BIT to facilitate using a single byte or word as a flag to hold several status bits. For example, suppose our application wanted to set up a flag for whether the screen was black-and-white or color, and whether the printer was an ImageWriter or LaserWriter. Bit 7 could hold the color flag, bit 6 the printer code, and one BIT instruction could test for everything.

Bit 7: 0 = black-and-white, 1 = color
Bit 6: 0 = ImageWriter, 1 = LaserWriter

```assembly
TEST BIT FLAG
   BPL COLOR ; BIT 7 = 0
   BMI BLACKWH ; BIT 7 = 1
   BVC IMAGE ; BIT 6 = 0
   BVS LASER ; BIT 6 = 1
```

This routine works because the BIT instruction automatically transfers bits 6 and 7 to the Status Register. In the 16-bit mode, bits 14 and 15 would have to store the flags for transfer to the N and V bits of the Status Register.
**BMI: Branch on Minus**

**Description**
Executes branch only if the N flag (sign flag) is set. N flag is set by any operation producing a result in the range of $80 to $FF for byte operations, and $8000 to $FFFF for word operations (for example, high bit set). BMI is limited to a relative branch distance of $-128$ to $+127$ bytes.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

- **Mode** Relative Only
- **Common Syntax**
  - BMI LABEL
  - BMI $FFFF
- **Hex Coding**
  - 30 ff
  - 30 ff

**Uses**
BMI is used to detect negative numbers when signed binary math is used. However, because the high-order bit is frequently used to indicate a wide variety of special conditions, BMI, along with BPL, is used to test the high-order bit, irrespective of its negative number connotations. See also the BIT instruction.

The high-order bit of many hardware registers is used to indicate significant conditions. BMI is usually used to test these. Because the registers use only a single byte, this testing is usually done in the 8-bit mode. For example, here's a typical way of watching the keyboard hardware location, $C000$, for a keypress. (See BIT.)

```assembly
LOOP   LDA  KYBD ; 8-BIT MODE
     BMI  PRESS ; DATA > $7F
     BPL  LOOP ; DATA < $80
```

Since bit 7 is the high-order bit and has no effect on the ASCII character associated with a byte, this can be quite handy. For example, some word processing applications store formatting characters (such as flags for underlining, bold, and so forth) among the text of a document. The high-order bit of each byte is used to indicate whether that byte is a text byte (high bit clear) or a formatting byte (high bit set).

```assembly
READ   LDA  CHAR ; GET CHARACTER FROM TEXT
     BMI  FORMAT ; SPECIAL FORMAT ENCODING
     CONT  NOP ; CONTINUE WITH TEXT READER HERE
```
BMI

BMI is also useful for terminating a loop that you want to reach 0, and where the loop will otherwise stay out of the $80 to $FF range. Notice that a loop terminated by a decrement and a BMI test will loop n + 1 times, where n is the starting value of the counter.

ENTRY LDX #$20 ; TO LOOP 33 TIMES
LOOP DEX
    BMI DONE ; WHEN X = $FF
    BPL LOOP ; WHILE X > $FF
DONE RTS
**BNE: Branch Not Equal**

**Description**
Executes the branch if the Z flag (zero flag) is clear, that is to say, if the result of an operation was a nonzero value. BNE is limited to a relative branch distance of −128 to +127 bytes.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative Only</td>
<td>BNE Label</td>
<td>D0 FF</td>
</tr>
<tr>
<td></td>
<td>or:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BNE $FFFF</td>
<td>D0 ff</td>
</tr>
</tbody>
</table>

**Uses**

Often used in loops to branch until the counter reaches zero. Also used in data input loops to verify the nonzero nature of the last byte loaded, as when checking for the end of a string.

BNE can be used to terminate a loop by waiting for the counter to reach zero. (See BEQ also.) When BNE follows the decrement instruction at the end of the loop, the counter is initialized with the value for the number of times for the loop to execute.

```
LDX  #25
LOOP: JSR ROUTINE
      DEX
      BNE LOOP
DONE: RTS
```

BNE can be used in a way similar to that for BEQ to check for the end of a string by using zero as a delimiter.

**Example:**

```
LDX  #$00
LOOP: LDA DATA,X
      BNE WORK
      JMP DONE
WORK: NOP
      INX
      JMP LOOP
DONE: RTS
```

493
BPL

**BPL: Branch on Plus**

**Description**
Executes branch only if the N flag (Sign Flag) is clear, as would be the case when the result of an operation is in the range of $00$ to $7F$ (high bit clear) for byte operations, and $0000$ to $7FFF$ for word operations. BPL is limited to a relative branch distance of $-128$ to $+127$ bytes. (See also BMI.)

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative Only</td>
<td>BPL LABEL</td>
<td>10 ff</td>
</tr>
<tr>
<td></td>
<td>BPL $FFFF$</td>
<td>10 ff</td>
</tr>
</tbody>
</table>

**Uses**

BPL is an easy way of staying in a loop until the high bit is set. It is also used in general to detect the status of the high bit.

Here's our familiar keypress check using BPL:

ENTRY LDA KYBD ; $C000, 8-BIT MODE
BPL ENTRY ; LOOP UNTIL DATA > $7F
BIT STROBE ; CLR $C010
STA MEM ; SAVE VALUE
DONE RTS

Also used for short loops (counters less than 128) that you want to reach zero.

ENTRY LDX #$20 ; WILL LOOP 33 TIMES
LOOP DEX ; X = X - 1
BPL LOOP ; UNTIL X = $FF
DONE RTS
BRA: Branch Always

Description
Always executes a branch. BRA is limited to a relative branch distance of −128 to +127 bytes. (See BRL.)

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative Only</td>
<td>BRA LABEL</td>
<td>80 FF</td>
</tr>
<tr>
<td></td>
<td>or:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BRA $FFFF</td>
<td>80 FF</td>
</tr>
</tbody>
</table>

Uses
BRA is used mainly as a substitute for a JMP instruction in code that must be position independent without the benefit of a relocating loader like the System.Loader. In situations where the routine must be as small as possible, BRA also takes fewer bytes (2 vs. 3) than a JMP instruction.

Here's a loop that uses a BRA instead of a JMP:

```
LDX #25 ; LOOP 25 TIMES
LOOP  JSR ROUTINE ; DO SOMETHING
     DEX ; COUNT DOWN
     BEQ DONE ; X = 0 = FINISHED
     BRA LOOP ; BACK FOR MORE
```

BRK: Break (Software Interrupt)

Description
When a BRK is encountered in a program, program execution halts, and the user generally sees something like the following:

```
00/0308: 00 00        BRK 00
A=00A0 X=0000 Y=0001 S=0137 D=0000 P=D9
B=00 K=00 M=0C Q=80 L=1 m=1 x=1 e=1
```
BRK

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

Emulation mode only (e = 1)

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>BRK 00</td>
<td>00 00</td>
</tr>
</tbody>
</table>

What happens is that the program counter plus two is saved on the stack. What happens next depends on which mode—emulation or native—the 65816 is in when the BRK occurs.

Emulation mode. The status register, where the BRK bit has been set, is pushed on the stack. The processor jumps to the address pointed to by the bytes at $00/FFFE,FFFF. This currently points to $00/C074, which jumps again to the Apple IIgs break handler at $E1/0010. For ProDOS 8 applications, control ultimately passes to a vector at $3F0,3F1.

Native mode. There's not a BRK bit in the status register, so, instead of setting a bit, the processor jumps to a vector pointed to by locations $00/FFE6,FFE7—the native mode BRK handler.

Uses

BRK can be very useful in debugging ML programs. Insert a BRK into the code at strategic points in the routine. When the program halts, examine the status of various memory locations and registers to see if all is as it should be. This process can be formalized, and considerably improved, by using a software utility called a debugger, which allows you to step through a program one instruction at a time.

Because BRK advances the program counter two bytes, the Monitor disassembles it as a two-byte instruction. Although Merlin lets you enter a single-byte BRK instruction in a source listing, your programs will disassemble better if you create each one in the BRK $00 form. If conserving memory is an issue, once your program is debugged, go back and replace the BRK $00 instructions with a simple one-byte BRK.
**BRL: Branch Always Long**

**Description**
Always executes a branch. BRL is different from BRA in that it supports a two-byte displacement and is limited only to a target address within the current program bank.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative Only</td>
<td>BRL LABEL</td>
<td>80 FF</td>
</tr>
<tr>
<td></td>
<td>or:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BRL $FFFF</td>
<td>80 FF</td>
</tr>
</tbody>
</table>

**Uses**
BRL is used mainly as a substitute for a JMP instruction in code that must be position independent without the benefit of a relocating loader like the System.Loader.

Here's a loop that uses a BRL instead of a JMP:

```assembly
LDX #25 ; LOOP 25 TIMES
LOOP   JSR ROUTINE ; DO SOMETHING
        DEX ; COUNT DOWN
        BEQ DONE ; X = 0 = FINISHED
        BRL LOOP ; BACK FOR MORE
```

497
BVC

**BVC: Branch on Overflow Clear**

**Description**
Executes a branch only if the V flag (overflow flag) is clear. The overflow flag is cleared whenever the result of an operation did not entail the carry of a bit from position 6 to position 7 (or bit 14 to 15 for words). The overflow flag can also be cleared with a CLV command. BVC is limited to a relative branch distance of −128 to +127 bytes.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative Only</td>
<td>BVC LABEL</td>
<td>50 ff</td>
</tr>
<tr>
<td></td>
<td>or:</td>
<td>50 ff</td>
</tr>
</tbody>
</table>

The overflow bit is only conditioned by the instructions ADC, SBC, BIT, PLP, REP and SEP. That is to say, just loading a value into the Accumulator or a register doesn't automatically condition the V flag, as is the case with the N flag.

**Uses**
BVC is used primarily in detecting a possible overflow from the data portion of the byte into the sign bit when using signed binary numbers. For example:

```assembly
ENTRY  CLC
  LDA  #$64 ; %01100100 = +100
  ADC  #$40 ; %01000000 = +64
  BVC  STORE ; NOT TAKEN HERE
ERR   RTS  ; RESULT = +164 = %10100100 > $7F
STORE  STA  MEM
```

BVC can also be used as a forced branch when writing position-independent code. The advantage is that the carry remains unaffected, thus allowing it to be tested later in the conventional manner.

```assembly
CLV ; CLEAR V FLAG
BVC  LABEL ; (ALWAYS)
```
BVS: Branch Overflow Set

Description
Executes the branch only when the V flag (overflow flag) is set. The overflow flag is set only when the result of an operation causes a carry of a bit from position 6 to position 7 (or bit 14 to 15 for words). Note that there is not a command to specifically set the overflow flag (as would correspond to a SEC command for the carry.) BVS is limited to a relative branch distance of -128 to +127 bytes.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative Only</td>
<td>BVS LABEL</td>
<td>70 ff</td>
</tr>
<tr>
<td>or:</td>
<td>BVS $FFFF</td>
<td>70 ff</td>
</tr>
</tbody>
</table>

The overflow bit is only conditioned by the instructions ADC, SBC, BIT, PLP, REP and SEP. That is to say, just loading a value into the Accumulator or a register does not automatically condition the V flag as is the case with the N flag.

Uses
BVS is used primarily in detecting a possible overflow from the data portion of the byte into the sign bit, when using signed binary numbers. For example:

ENTRY CLC
LDA #$64 ; %01100100 = +100
ADC #$40 ; %01000000 = +64
BVS ERR ; RESULT = +164 = %10100100 > $7F
STORE STA MEM
DONE RTS
ERR JSR BELL ; ALERT TO OVERFLOW

The overflow bit can be explicitly set by using the BIT instruction on an Accumulator or memory location value with the appropriate bit (bit 6 or bit 14) set. (See also BVC.) For example:

BIT $FF58 ; ASSUMES 8-BIT MODE...
BVS SET ; GUARANTEED TO BE #$60
BVS SET ; BRANCH ON OVERFLOW SET
CLC

CLC: Clear Carry
Description
Clears the carry bit of the status register.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>CLC</td>
<td>18</td>
</tr>
</tbody>
</table>

Uses
CLC is usually required before the first ADC instruction of an addition operation, to make sure the carry hasn’t inadvertently been set somewhere else in the program, and thus incorrectly added to the values used in the routine itself.

```
CLC ; PREPARE TO ADD
LDA MEM ; GET FIRST VALUE
ADC MEM2 ; ADD 2ND VALUE
STA RSLT ; SAVE RESULT
```

CLC is also prior to an XCE instruction to set the 65816 microprocessor to the native mode.

```
CLC
XCE ; e = 0 = NATIVE MODE
```

A CLC can also be used to force a branch when writing position-independent code, such as:

```
CLC
BCC LABEL ; (ALWAYS)
```
CLD: Clear Decimal Mode

Description
CLD is used to enter the binary mode (which the Apple is usually in by default), so as to properly use the ADC and SBC instructions. (See SED for setting decimal mode.)

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>CLD</td>
<td>D8</td>
</tr>
</tbody>
</table>

Uses
The arithmetic mode of the 65816 is an important point to keep in mind when using the ADC and SBC instructions. If you are in the wrong mode from what you might assume, rather unpredictable results can occur. See the SED instruction entry for more details on the other mode.
CLI

CLI: Clear Interrupt Mask

Description
This instruction enables interrupts by clearing bit 2 of the Status Register.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>clr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

- **Mode**: Implied Only
- **Common Syntax**: CLI
- **Hex Coding**: 58

Uses
CLI tells the 65816 to recognize incoming IRQ (Interrupt ReQuest) signals. The Apple's default is to have interrupts enabled, but after the first interrupt, all succeeding interrupts are disabled by the 65816 until a CLI is re-issued, which usually happens automatically at the end of the given interrupt routine. Because timing-dependent operations such as writing to a disk drive cannot be interrupted without damaging the data being written, interrupts are usually disabled during a disk access. That's why you can't always get to the Classic Desk Accessory menu with Control-Open Apple-Escape when the disk drive is on. Certain operating systems like Pascal and ProDOS 8 version 1.1.1 disable interrupts when they first start up. If you wish to enable the Desk Accessory menu in these operating systems, you must execute a CLI instruction. (See also SEI.)
**CLV: Clear Overflow Flag**

**Description**
This clears the overflow flag by setting the V bit of the status register to 0.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>clr</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>CLV</td>
<td>B8</td>
</tr>
</tbody>
</table>

**Uses**
Because the overflow flag is automatically cleared by a non-overflow result of an ADC instruction, it is not usually necessary to clear the flag prior to an addition operation. It is, however, occasionally used as a way of forcing a branch instruction when writing position-independent code.

```
CLV        ; CLEAR OVERFLOW FLAG
BVC ROUTINE ; BRANCH TO ROUTINE
```

This technique has the advantage of leaving the carry flag undisturbed should your program wish to test the carry flag after the forced branch.
CMP

**CMP: Compare to Accumulator**

**Description**

CMP compares the Accumulator to a specified value or memory location. The N (sign), Z (zero), and C (carry) flags are conditioned. A conditional branch is usually then done to determine whether the Accumulator was less than, equal to, or greater than the data.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>CMP $FFff</td>
<td>CD ff FF</td>
</tr>
<tr>
<td>Absolute Long</td>
<td>CMP $00FFff</td>
<td>CF ff FF 00</td>
</tr>
<tr>
<td>Direct Page</td>
<td>CMP $FF</td>
<td>C5 FF</td>
</tr>
<tr>
<td>Direct Page Indirect</td>
<td>CMP ($FF)</td>
<td>D2 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Long</td>
<td>CMP [$FF]</td>
<td>C7 FF</td>
</tr>
<tr>
<td>Immediate</td>
<td>CMP #$FF</td>
<td>C9 FF</td>
</tr>
<tr>
<td>Absolute Indexed,X</td>
<td>CMP $FFff,X</td>
<td>DD ff FF</td>
</tr>
<tr>
<td>Absolute Long Indexed,X</td>
<td>CMP $00FFff,X</td>
<td>DF ff FF 00</td>
</tr>
<tr>
<td>Absolute Indexed,Y</td>
<td>CMP $FFff,Y</td>
<td>D9 ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed,X</td>
<td>CMP $FF,X</td>
<td>D5 FF</td>
</tr>
<tr>
<td>Direct Page Indexed Indirect,X</td>
<td>CMP ($FF,X)</td>
<td>C1 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Indexed,Y</td>
<td>CMP ($FF),Y</td>
<td>D1 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Long Indexed,Y</td>
<td>CMP [$FF],Y</td>
<td>D7 FF</td>
</tr>
<tr>
<td>Stack Relative</td>
<td>CMP $FF,S</td>
<td>C3 FF</td>
</tr>
<tr>
<td>Stack Relative Indirect Indexed,Y</td>
<td>CMP ($FF,S),Y</td>
<td>D3 FF</td>
</tr>
</tbody>
</table>

**Uses**

CMP is used to check the value of a byte against certain values such as would be done in loops, or in data processing routines. The routine typically decides whether the result is less than, equal to, or greater than a critical value. The usual pattern is:

- **BCC**: Acc. < value
- **BCS**: Acc. >= value
- **BEQ, BCS**: Acc. > value

See the sections on BCC through BCS for specific examples.
**Important:** A CMP #$00 followed by a BCC or BCS as the test to the end of a loop should never be done. Consider this example:

```
ENTRY    LDY    #$FF
LOOP     DEY
         CPY    #$00
         BCS    LOOP   ; (ALWAYS TAKEN!)
         BCC    DONE
DONE     RTS
```

Because $01 through $FF is larger than zero, the branch will be taken while the Y register is in this range. Since $0 = $0, when Y reaches 0, the branch will still be taken. Therefore, the example creates an endless loop which will never terminate.

Similarly, if a BCC is done first, it will never be taken, since there is no value less than zero to trigger it.

This bug is most likely to show up when you design a loop that uses a memory location to store, for example, the length of a string.

```
ENTRY    LDY    #$FF
LOOP     DEY
         CPY    LEN    ; LEN HELDS LENGTH OF STRING
         BCS    LOOP   ; ALWAYS TAKEN IF (LEN) = 0
DONE     RTS
```

As long as the length of the string is nonzero, everything works fine. However, a zero-length string will cause the routine to loop forever, making the system appear to hang up.
COP

**COP: Co-Processor Enable**

**Description**
This is a special instruction similar to the BRK instruction, but which is designed to allow access to a secondary microprocessor to the 65816. For example, the Apple II GS contains additional microprocessors for the keyboard and sound generation operations.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>clr</td>
<td>set</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>COP 00</td>
<td>02 00</td>
</tr>
</tbody>
</table>

Like the BRK instruction, a COP instruction causes the program counter plus two to be saved on the stack. The 65816 then jumps to the address indicated by the vector stored in locations $00/FFF4,FFF5. Unlike the BRK instruction, the byte following the COP instruction is presumed to have a specific meaning, and is used to pass an instruction to the other microprocessor. Operands in the range of $80 to $FF have been reserved by Western Design Center, the designer of the 65816. Operands in the range of $00 to $7F are available.
**CPX: Compare data to the X Register.**

**Description**
CPX compares the contents of the X Register against a specified value or memory location. (See also CMP.)

**Flags & Registers Affected**

```
N V - B D I Z C Acc X Y Mem
• • • • • • •
```

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>CPX $FFff</td>
<td>EC ff FF</td>
</tr>
<tr>
<td>Direct Page</td>
<td>CPX $FF</td>
<td>E4 FF</td>
</tr>
<tr>
<td>Immediate</td>
<td>CPX #$FF</td>
<td>E0 FF</td>
</tr>
</tbody>
</table>

**Uses**
CPX is primarily used in loops which read data tables, with the X-Register being used as the offset in the Absolute,X addressing mode. The X Register is usually loaded with zero and then is incremented until it reaches the length of the data stream to be read. For example:

```
ENTRY LDX #$00 ; INITIALIZE OFFSET
LOOP LDA DATA,X ; GET A CHARACTER
    JSR PRINT ; PRINT IT
    INX ; X = X + 1
    CPX #$05 ; DONE WITH THE LOOP?
    BCC LOOP ; NOPE (LESS THAN 5)
DONE RTS ; ALL DONE
DATA ASC "TEST!" ; STRING DATA TO PRINT
```

For the same reasons discussed under CMP, a CPX #$00 or CPX MEM followed by BCC or BCS should not be used. (See CMP for details.)
**CPY: Compare Data to the Y Register**

**Description**
CPY compares the contents of the Y Register against a specified value or memory location. (See also CMP.)

**Flags & Registers Affected**
```
<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>CPY $FFff</td>
<td>CCff FF</td>
</tr>
<tr>
<td>Direct Page</td>
<td>CPY $FF</td>
<td>C4 FF</td>
</tr>
<tr>
<td>Immediate</td>
<td>CPY #$FF</td>
<td>C0 FF</td>
</tr>
</tbody>
</table>

**Uses**
The Y Register is used when reading a stream of data indirectly from a direct-page pointer. CPY allows for checking the current value of the Y Register against a critical value. In this example, the Y Register is used to retrieve and print the first five bytes of a string.

```
ENTRY  LDY  #$00  ; INITIALIZE OFFSET
       LDA  #$00  ; LOW BYTE OF DATA
       STA  PTR
       LDA  #$00  ; (PTR) = DATA
       STA  PTR+1

LOOP  LDA  (PTR),x ; GET A CHARACTER
       JSR  PRINT ; PRINT IT
       INY  ; X = X + 1
       CPY  #$05  ; DONE WITH THE LOOP?
       BCC  LOOP  ; NOPE (LESS THAN 5)
       DOTE  RTS  ; ALL DONE
       DATA  ASC  "TEST!" ; STRING DATA TO PRINT
```

For the same reasons discussed under CMP, a CPY #$00 or CPY MEM followed by BCC or BCS should not be used. (See CMP for details.)
DEC: Decrement a Memory Location

Description
The contents of the specified memory location (byte or word) are decremented by one. If the original contents were equal to #$00, then the result would wrap around, giving a result of #$FF (or #$FFFF for a word).

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied (Accumulator)</td>
<td>DEC</td>
<td>3A</td>
</tr>
<tr>
<td>Absolute</td>
<td>DEC $FFff</td>
<td>CE ff FF</td>
</tr>
<tr>
<td>Direct Page</td>
<td>DEC $FF</td>
<td>C6 FF</td>
</tr>
<tr>
<td>Absolute Indexed,X</td>
<td>DEC $FFff,X</td>
<td>DE ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed,X</td>
<td>DEC $FF,X</td>
<td>D6 FF</td>
</tr>
</tbody>
</table>

Uses
DEC is usually used when decrementing a one-byte memory value (such as a counter), or a two-byte memory pointer (word). Here are the common examples:

One-Byte or Word Value

<table>
<thead>
<tr>
<th>ENTRY DEC MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>DONE RTS</td>
</tr>
</tbody>
</table>

Two-Byte Pointer

<table>
<thead>
<tr>
<th>ENTRY DEC MEM</th>
<th>ENTRY DEC MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LDA MEM</td>
</tr>
<tr>
<td></td>
<td>CMP #$FF</td>
</tr>
<tr>
<td></td>
<td>BNE DONE</td>
</tr>
<tr>
<td></td>
<td>DEC MEM+1</td>
</tr>
<tr>
<td>DONE RTS</td>
<td></td>
</tr>
</tbody>
</table>

Two-Word Pointer:

<table>
<thead>
<tr>
<th>ENTRY DEC MEM</th>
<th>; 8-BIT MODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA MEM</td>
<td>; GET LOW BYTE</td>
</tr>
<tr>
<td>CMP #$FFFF</td>
<td>; WRAP AROUND?</td>
</tr>
<tr>
<td>BNE DONE</td>
<td>; NO</td>
</tr>
<tr>
<td>DEC MEM+1</td>
<td>; YES: DEC MEM+1</td>
</tr>
<tr>
<td>DONE RTS</td>
<td></td>
</tr>
</tbody>
</table>

After the DEC operation, the N and Z flags are often checked to see if the result was negative or a zero/nonzero value, respectively.

The technique shown for the two-byte decrement operation is not necessarily the most efficient for a two-word (or two-byte in the 8-bit mode) decrement. See the SBC entry for an alternative method.
DEX

**DEX: Decrement the X Register**

**Description**
The X Register is decremented by 1. When the original value was #$00, the result will wrap around to give a result of #$FF—or of #$FFFF, for a word. (See also DEC.)

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

- **Common Syntax**
  - DEX
- **Hex Coding**
  - CA

**Uses**
DEX is often used in reading a data block via indexed addressing, for example, LDA $1234,X. Here is a simple example:

```assembly
ENTRY LDX #$05 ; START COUNTER = 5
LOOP LDA DATA-1,X ; 8-BIT ACCUMULATOR
    JSR PRINT ; PRINT THE CHARACTER
    DEX ; X = X - 1
    BNE LOOP ; NOT DONE YET
DONE RTS
DATA ASC "!TSET"
```

**Note:** There are several points of interest in this example. Besides the general use of the X Register in the indexed addressing mode, notice that the loop runs backward from #$05 to #$01. The loop is terminated when the X Register reaches zero. Because the loop runs from high memory down, the ASCII string is put in memory in reverse order, as evidenced in the listing. Also note that the base address of the loop is DATA−1. This allows the use of the #$05 to #$01 values of the X Register.
DEY: Decrement the Y Register

Description
The Y Register is decremented by 1. When the original value was #$00, the result will wrap around to give a result of #$FF, or a result of #$FFFF, for a word. (See also DEC.)

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>^</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>DEY</td>
<td>88</td>
</tr>
</tbody>
</table>

Uses
DEY is usually used when decrementing a reverse scan of a data block, using a direct-page pointer via indirect indexed addressing (such as LDA ($FF),Y). Reverse scans are often used because it's so easy to use a BEQ instruction to detect when you're done. DEY is also used when making a counter for a small number of cycles. Here's a routine which outputs a variable number of carriage returns, as indicated by the contents of MEM.

ENTRY LDY MEM ; GET COUNTER VALUE
BEQ DONE ; PROTECT AGAINST (MEM) = 0
LOOP LDA #$8D ; <RETURN>
JSR COUT ; PRINT IT
DEY
BNE LOOP ; TILL Y=0
DONE RTS

511
EOR: Exclusive Or with Accumulator

Description
The contents of the Accumulator are exclusive ORed with the specified data. The N (sign) and Z (zero) flags are also conditioned depending on the result. The result is put back in the Accumulator. The memory location (if specified) is unaffected.

The truth table used is:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Example:

AND:  

<table>
<thead>
<tr>
<th>Accumulator:</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory:</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Result:</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>EOR $FFff</td>
<td>4D ff FF</td>
</tr>
<tr>
<td>Absolute Long</td>
<td>EOR $00FFff</td>
<td>4F ff FF 00</td>
</tr>
<tr>
<td>Direct Page</td>
<td>EOR $FF</td>
<td>45 FF</td>
</tr>
<tr>
<td>Direct Page Indirect</td>
<td>EOR ($FF)</td>
<td>52 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Long</td>
<td>EOR [$FF]</td>
<td>47 FF</td>
</tr>
<tr>
<td>Immediate</td>
<td>EOR #$FF</td>
<td>49 FF</td>
</tr>
<tr>
<td>Absolute Indexed,X</td>
<td>EOR $FFff,X</td>
<td>5D ff FF</td>
</tr>
<tr>
<td>Absolute Long Indexed,X</td>
<td>EOR $00FFff,X</td>
<td>5F ff FF 00</td>
</tr>
<tr>
<td>Absolute Indexed,Y</td>
<td>EOR $FFff,Y</td>
<td>59 ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed,X</td>
<td>EOR $FF,X</td>
<td>55 FF</td>
</tr>
<tr>
<td>Direct Page Indexed Indirect,X</td>
<td>EOR ($FF,X)</td>
<td>41 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Indexed,Y</td>
<td>EOR ($FF),Y</td>
<td>51 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Long Indexed,Y</td>
<td>EOR [$FF],Y</td>
<td>57 FF</td>
</tr>
<tr>
<td>Stack Relative</td>
<td>EOR $FF,S</td>
<td>43 FF</td>
</tr>
<tr>
<td>Stack Relative Indirect Indexed,Y</td>
<td>EOR ($FF,S),Y</td>
<td>53 FF</td>
</tr>
</tbody>
</table>

Uses
EOR has a wide variety of uses. One is to encode data by doing an EOR with an arbitrary one-byte key. The data may then be decoded later by again doing an EOR of each data byte with the same key again.
Another application of EOR is to reverse a given bit of a data byte. The mask is created by putting 1 in the positions that you wish to have reversed. A zero is put in all remaining positions. When the EOR with the mask is done, bits in the specified positions will reverse; for example, ones will become zeroes, and vice versa. See the truth table for this entry to verify this effect.

The Z (zero) flag will be set if either the Accumulator or memory or both equal zero:

```
ENTRY LDA MEM
EOR MEM2
BEQ ZERO ; MEM = 0 and/or MEM2 = 0
BNE NOTZ ; NEITHER MEM NOR MEM2 = 0
```

EOR is useful in producing the twos complement of a number for use in signed binary arithmetic.

```
ENTRY LDA #$34 ; %00110100 = +52
 ; TO BE CVRTD TO -52
EOR #$FF ; %11111111 = $FF
 ; RSLT = %11001111
CLC
ADC #$01 ; RSLT = RSLT + 1
 ; = %11011100 = $CC
STA MEM ; STORE RSLT
DONE RTS
```
EOR

And to convert signed negative numbers back:

ENTRY LDA #$CC ; %11001100 = $CC = -52
    ; TO BE CVRTD BACK
SEC
SBC #$01 ; ACC = ACC - 1
    ; = %11001011 = $CB
EOR #$FF ; REVERSE ALL BITS
    ; RSLT = %00110100 = $34 = +52
STA MEM ; STORE RESULT
DONE RTS

In graphics, doing an EOR using #$FF (or #$FFFF for words) will reverse the visible image:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>100</td>
</tr>
<tr>
<td>010</td>
<td>010</td>
<td>100</td>
<td>001</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>001</td>
<td>100</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>000</td>
<td>000</td>
<td>100</td>
<td>100</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>000</td>
<td>000</td>
<td>010</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>

EOR 1111111 = 11101111
00010000 11111111 01111111
00001000 11111111 11110111
00000100 11111111 11111011
00000010 11111111 11111101
00000001 11111111 11111110
INC: Increment Memory

Description
The contents of a specified memory location are incremented by 1. If the original value was #$FF (or #$FFFF for a word), then incrementing will result in a wrap around, giving a result of #$00. The N (sign) and Z (zero) flags are conditioned depending on the result.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied (Accumulator)</td>
<td>INC</td>
<td>1A</td>
</tr>
<tr>
<td>Absolute</td>
<td>INC $FFff</td>
<td>EE ff FF</td>
</tr>
<tr>
<td>Direct Page</td>
<td>INC $FF</td>
<td>E6 FF</td>
</tr>
<tr>
<td>Absolute Indexed,X</td>
<td>INC $FFff,X</td>
<td>FE ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed,X</td>
<td>INC $FF,X</td>
<td>F6 FF</td>
</tr>
</tbody>
</table>

Uses
INC is most often used for incrementing a one-byte (or word) value (such as a counter) or a two-byte or two-word pointer. Here are the most common forms:

8-bit Accumulator/Memory (e = 1):

**One-Byte Value**

- ENTRY INC MEM
- RTS

**Two-Byte Pointer**

- ENTRY INC MEM
- BNE DONE
- INC MEM+1
- DONE RTS

16-bit Accumulator/Memory (e = 0, m = 0):

**One-Word Value**

- ENTRY INC MEM
- RTS

**Two-Word Pointer**

- ENTRY INC MEM
- BNE DONE
- INC MEM+2
- DONE RTS

After the INC operation, the N and/or Z flags are often checked to see if the result was negative or a zero/nonzero value, respectively.
INX

INX: Increment the X Register

Description
The contents of the X Register are incremented by one. If the original value was #$FF (or #$FFFF for a word), then incrementing will result in a wrap around, giving a result of #$00. The N (sign) and Z (zero) flags are conditioned depending on the result.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only:</td>
<td>INX</td>
<td>E8</td>
</tr>
</tbody>
</table>

Uses
INX is used in forward scanning loops which digest a DATA stream as shown here:

ENTRY LDX #$00 ; INITIALIZE INDEX
LOOP LDA DATA,X ; GET A CHARACTER
BEQ DONE ; DELIMITER?
JSR PRINT
BRA LOOP ; NEXT CHAR
DONE RTS
DATA ASC "TEST!"
HEX 00 ; END OF DATA

INX can also be used as a general purpose counter for miscellaneous routines.

ENTRY LDX #$00
LDA #$8D ; <RETURN>
LOOP JSR PRINT
INX
CPX #$05 ; TILL X = 5
BCC LOOP ; PRINTS 5 CR'S
DONE RTS

Note that in forward scanning loops that check a length or limit value, the base address can be DATA itself. (See DEX for another situation.)
INY: Increment the Y Register

Description
The contents of the Y Register are incremented by one. If the original value was #$FF (or #$FFFF for a word), then incrementing will result in a wrap around giving a result of #$00. The N (sign) and Z (zero) flags are conditioned depending on the result.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>INY</td>
<td>C8</td>
</tr>
</tbody>
</table>

Uses
INY is used in forward scanning loops that use the indirect indexed addressing mode (for example, LDA ($FF),Y). This is quite common in routines which process strings for certain characters, such as search routines. Here is a routine that scans an input buffer for the first carriage return:

ENTRY LDA #$<BUF> ; LOW BYTE OF BUFFER ADDRESS
STA PTR ; LOW BYTE OF POINTER
LDA #$>BUF ; HIGH BYTE OF BUFFER ADDRESS
STA PTR ; (PTR) = BUFFER
LDY #$00 ; INITIALIZE INDEX

LOOP LDA (PTR),Y ; GET A CHARACTER
CMP #$8D ; CHR = <CR>?
BEQ FOUND ; LEAVE LOOP
INY ; Y = Y + 1
BNE LOOP ; 'TILL Y = $00 AGAIN

DONE RTS

FOUND STY MEM ; SAVE POSITION OF FOUND CHAR.
BRA DONE ; (ALWAYS)
JML/JMP: Jump to Address

Description
Causes program execution to jump to the address specified. Except for the long address form of JMP (JML), a JMP instruction always goes to the operand address within the current program bank. A JML automatically changes the Program Bank Register to the bank of the target address.

Flags & Registers Affected (none)

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>JMP $FFff</td>
<td>4C ff FF</td>
</tr>
<tr>
<td>Absolute Long</td>
<td>JML $00FFff</td>
<td>5C ff FF 00</td>
</tr>
<tr>
<td>Absolute Indirect</td>
<td>JMP ($FFff)</td>
<td>6C ff FF</td>
</tr>
<tr>
<td>Absolute Indexed Indirect</td>
<td>JMP ($FFff,X)</td>
<td>7C ff FF</td>
</tr>
<tr>
<td>Absolute Indirect Long</td>
<td>JML [$FFff]</td>
<td>DC ff FF</td>
</tr>
</tbody>
</table>

Uses
Besides the obvious application of the usual absolute addressed JMP instruction, the indirect JMP is used when creating vectored jumps. The Apple IIGS makes extensive use of these vectors for handling everything from the RESET vector to the patches to the tool set calls.

By creating a table of vectors, your application can use an input value and the Absolute Indexed Indirect JMP to go to a particular routine.

```
LDA CMD ; GET COMMAND VALUE $0 TO $2
ASL ; MULTIPLY BY 2 = 0,2,4
JMP (CMDTBL,X) ; GO TO VECTORED ROUTINE
CMDTBL DA ROUTINE1 ; ADDRESS OF ROUTINE1 (2 BYTES)
DA ROUTINE2 ; ADDRESS OF ROUTINE2 (2 BYTES)
DA ROUTINE3 ; ADDRESS OF ROUTINE3 (2 BYTES)
```
JSL/JSR: Jump to Subroutine

Description
The address of the last byte of the JSR instruction is pushed onto the stack. This is equivalent to the return address minus one. The address of the operand for the JSR is then jumped to. When an RTS in the called subroutine is encountered, a return to the location on the stack plus one (the address of the next instruction after the JSR) is done. This is analogous to a GOSUB in BASIC. Except for the long address form of JSR (JSL), a JSR always goes to the operand address within the current program bank.

Flags & Registers Affected (none)

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>JSR $FFff</td>
<td>20 ff FF</td>
</tr>
<tr>
<td>Absolute Indexed Indirect</td>
<td>JSR ($FFff,X)</td>
<td>FC ff FF</td>
</tr>
<tr>
<td>Absolute Long</td>
<td>JSL $00FFff</td>
<td>22 ff FF 00</td>
</tr>
</tbody>
</table>

Uses
JSR is one of the most commonly used instructions, being used to call often needed subroutines. The disadvantage of the instruction is that if the JSR's reference addresses within the code (as opposed to routines external to the program, such as in the ROM), the code can only be executed at the location for which the code was originally assembled. This problem can be avoided by using the System Loader and creating ProDOS 16 files in a relocatable format, or by writing the program so as to be position independent (no internal address references).

Because the calling address is saved on the stack, a JSR to a known RTS can be done, and the data can be retrieved to determine where in memory the routine is currently being executed. (See PER.)

The Absolute Indexed Indirect form of the JSR is often used in Apple IIGS programs using the Event Manager and Menu Manager because of the easy vectoring to an appropriate subroutine. The addresses for each subroutine to be called are defined in a common table, and the command value times 2 is used as the offset for the JSR.
JSL/JSR

LDA CMD ; GET COMMAND VALUE $0 TO $2
ASL ; MULTIPLY BY 2 = 0,2,4
JSR (CMDTBL,X) ; GO TO VECTORED ROUTINE
JMP MAIN ; BACK TO MAIN ROUTINE

CMDTBL DA ROUTINE1 ; ADDRESS OF ROUTINE1 (2 BYTES)
DA ROUTINE2 ; ADDRESS OF ROUTINE2 (2 BYTES)
DA ROUTINE3 ; ADDRESS OF ROUTINE3 (2 BYTES)
**LDA: Load Accumulator**

**Description**
Loads the Accumulator with either a specified value or the contents of the designated memory location. One byte is loaded in the 8-bit mode, 2 bytes (a word) in the 16-bit mode. The N (Sign) and Z (Zero) flags are conditioned when a value with either the high bit set or a zero value are loaded, respectively.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

- **Mode**
  - **Common Syntax**
    - Absolute: LDA $FFff
    - Absolute Long: LDA $00FFff
    - Direct Page: LDA $FF
    - Direct Page Indirect: LDA ($FF)
    - Direct Page Indirect Long: LDA [$FF]
    - Immediate: LDA #$FF
    - Absolute Indexed, X: LDA $FFff,X
    - Absolute Long Indexed, X: LDA $00FFff,X
    - Absolute Indexed, Y: LDA $FFff,Y
    - Direct Page Indexed, X: LDA $FF,X
    - Direct Page Indexed Indirect, X: LDA ($FF,X)
    - Direct Page Indirect Indexed, Y: LDA ($FF),Y
    - Direct Page Indirect Long Indexed, Y: LDA [$FF],Y
    - Stack Relative: LDA $FF,S
    - Stack Relative Indirect Indexed, Y: LDA ($FF,S),Y

- **Hex Coding**
  - Absolute: ADff FF
  - Absolute Long: AF ff FF 00
  - Direct Page: A5 FF
  - Direct Page Indirect: B2 FF
  - Direct Page Indirect Long: A7 FF
  - Immediate: A9 FF
  - Absolute Indexed, X: BD ff FF
  - Absolute Long Indexed, X: BF ff FF 00
  - Absolute Indexed, Y: B9 ff FF
  - Direct Page Indexed, X: B5 FF
  - Direct Page Indexed Indirect, X: A1 FF
  - Direct Page Indirect Indexed, Y: B1 FF
  - Direct Page Indirect Long Indexed, Y: B7 FF
  - Stack Relative: A3 FF
  - Stack Relative Indirect Indexed, Y: B3 FF

**Uses**
LDA is probably the most used of any instruction. The vast majority of operations center around the Accumulator, and this instruction is used to get data into the important register.
LDX

**LDX: Load the X Register**

**Description**
Loads the X Register with either a specified value, or the contents of the designated memory location. The N (sign) and Z (zero) flags are conditioned when a value with either the high bit set or a zero value are loaded, respectively. One byte is loaded when the x bit is 1, a word (2 bytes) is loaded when x is 0.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>LDX $FFff</td>
<td>AE ff FF</td>
</tr>
<tr>
<td>Direct Page</td>
<td>LDX $FF</td>
<td>A6 FF</td>
</tr>
<tr>
<td>Immediate</td>
<td>LDX #$FF</td>
<td>A2 FF</td>
</tr>
<tr>
<td>Absolute Indexed,Y</td>
<td>LDX $FFff,Y</td>
<td>BE ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed,Y</td>
<td>LDX $FF,Y</td>
<td>B6 FF</td>
</tr>
</tbody>
</table>

**Uses**
This is the primary way in which data is placed into the X Register.
LDY: Load the Y Register

Description
Loads the Y Register with either a specified value or the contents of the designated memory location. The N (sign) flag is conditioned when the high bit is set, and the Z (zero) flag is conditioned when a zero value is loaded. One byte is loaded when the \( x = 1 \), a word (2 bytes) when \( x = 0 \).

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>LDY $FFff</td>
<td>AC ff FF</td>
</tr>
<tr>
<td>Direct Page</td>
<td>LDY $FF</td>
<td>A4 FF</td>
</tr>
<tr>
<td>Immediate</td>
<td>LDY #$FF</td>
<td>A0 FF</td>
</tr>
<tr>
<td>Absolute Indexed,X</td>
<td>LDY $FFff,X</td>
<td>BC ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed,X</td>
<td>LDY $FF,X</td>
<td>B4 FF</td>
</tr>
</tbody>
</table>

Uses
This is the primary way in which data is placed into the Y Register.
LSR

**LSR: Logical Shift Right**

**Description**
This instruction moves each bit of the Accumulator or memory location specified one position to the right. A zero is forced at the bit 7 position (the high-order bit) for a byte, and bit 15 for a word. Bit zero falls into the carry. The result is left in the Accumulator or memory location. (See also ROL, ASL, and ROR.)

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>clr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 —> 7 6 5 4 3 2 1 0

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulator</td>
<td>LSR</td>
<td>4A</td>
</tr>
<tr>
<td>Absolute</td>
<td>LSR $FFff</td>
<td>4E ff FF</td>
</tr>
<tr>
<td>Direct Page</td>
<td>LSR $FF</td>
<td>46 FF</td>
</tr>
<tr>
<td>Absolute Indexed,X</td>
<td>LSR $FFff,X</td>
<td>5E ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed,X</td>
<td>LSR $FF,X</td>
<td>56 FF</td>
</tr>
</tbody>
</table>

**Uses**
ASL provides an easy way of dividing by a power of two. The corresponding effect in decimal arithmetic is well known: $123/10 = 12.3$ (shift right). As an example:

```
ENTRY LDA MEM ; GET ORIGINAL VALUE
  LSR ; DIV BY 2
  LSR ; DIV BY 2 AGAIN 2
  STA MEM ; MEM = MEM/4 (4 = 2)
```
ASL also provides a way of detecting whether a number is odd or even:

ENTRY LDA MEM
LSR
BCS EVEN
BCC ODD

Since bit 0 determines the odd/even nature of a number, this is easily transferred to the carry via the LSR, and then checked via the BCS/BCC instructions.

In double hi-res graphics and 80-column text, the bytes that make up the screen display are interleaved. For example, in 80-column text, each even position character (0, 2, 4, and so on) is located in bank 1 (AuxMem). Each odd position character is located in bank 0 (MainMem). Thus, the actual byte offset from the base address for the left edge of the screen for each line is equal to the horizontal position divided by two. Which bank is used is determined by whether the given horizontal position is an even or an odd number. These characteristics are ideal for the use of LSR in an 80-column print routine:

PRINT LDA CV ; VERTICAL POSITION
JSR VTAB ; MONITOR ROUTINE TO CALCULATE BASE ADDRESS
LDA CH ; HORIZONTAL POSITION
LSR ; DIVIDE BY TWO, CARRY SET IF ODD
TAY ; PUT RESULT IN Y REGISTER
BCS MAIN ; MAIN MEMORY IF ODD

AUX BIT PAGE2 ; SOFTSWITCH FOR BANK 1
STA BASL,Y ; WRITE CHARACTER
BRA CONT ; ALWAYS BRANCH TO CONTINUE

MAIN BIT PAGE1 ; SOFTSWITCH FOR BANK 0
STA BASL,Y ; WRITE CHARACTER

CONT NOP ; PROGRAM CONTINUES HERE
MVN/MVP: Block Move Next/Previous

Description
Moves an entire block of memory from one location to another. The source and destination blocks must not cross bank boundaries, but the destination block may be in a different bank than the source.

The next in MVN refers to the source block being after the destination block, but within the same bank. MVN (vs. MVP) is specifically required when the destination block is lower in memory, but within the same bank, as the source block. Otherwise either MVN or MVP may be used.

The previous in MVP refers to the source block being ahead of, or previous to, the destination block, but within the same bank. MVP is specifically required in that case.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

(Requires use of A, X and Y Registers.)

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>MVN $00,$FF</td>
<td>54 00 FF</td>
</tr>
<tr>
<td></td>
<td>MVP $00,$FF</td>
<td>54 00 FF</td>
</tr>
</tbody>
</table>

Uses
In processors previous to the 65816, such as the 65C02 in the Apple IIe and IIc, moving data was accomplished by calling a routine. For small amounts of memory, this was perfectly adequate. However, for a machine with the large amounts of memory like the Apple IIIGS, conventional methods of moving memory would just be too slow. Fortunately, the 65816 includes two move instructions, MVN and MVP which automate the moving of memory to enable large blocks of memory to be moved much faster.

In practice, it’s usually easier to use the Memory Manager calls PtrToHand, PtrToPtr, HandToHand, and BlockMove to move blocks of memory than it is to use the MVN or MVP instructions. That’s because the operands that describe the source and destination banks for MVN and MVP must be hard coded into the instruction. This means that you must either limit your moves to within predefined banks (fine for ProDOS 8 and Applesoft BASIC), or you must use self-modifying code to achieve flexibility for inter-bank transfers. In addition, should the source or destination blocks cross a bank boundary, additional code must be included to handle the separate blocks of memory. The
built-in routines of the Memory Manager already take all these considerations into account and use the MVN and MVP instructions to their best advantage while the machinations remain transparent to the programmer.

You have already seen from the demonstration program in Chapter 17 that the Memory Manager can move large blocks of memory quite quickly with the standard tool calls. However, in the event you should ever wish to use the MVN or MVP instructions explicitly, here is an appropriate code segment:

ENTRY LDX START ; STARTING ADDRESS
        LDY DEST ; DESTINATION ADDRESS
        LDA LENGTH ; NUMBER OF BYTES TO MOVE - 1
        MVN SBNK, DBNK ; SOURCE BANK, DESTINATION BANK
                        ; OR MVP IF NECESSARY

Note that normal use presumes that both $m$ and $x$ are equal to 0 (full 16-bit mode), but that MVN and MVP can be used with 8-bit $m$ or $x$ bits—or even in emulation mode. In any particular case, if the X or Y register has a high-order byte equal to zero (as would be the case with $e = 1$ or $x = 1$), then the source and destination will be limited to page zero ($00$ to $FF$), irrespective of the direct-page register. The complete Accumulator (C) is used for the length, regardless of the setting of the $e$ or $m$ bits.
NOP

NOP: No Operation

Description
Does nothing for one instruction.

Flags & Registers Affected (none)

Addressing Modes Available

Uses
NOP is used primarily to disable portions of code written by other programmers that you have decided you can live without. Additionally, NOPs may be used during debugging to disable certain steps.
ORA: Inclusive OR with the Accumulator

Description
This instruction takes each bit of the Accumulator and performs a logical OR with each corresponding bit of the specified memory location or immediate value. The result is put back in the Accumulator. The memory location, if specified, is unaffected. Conditions the N (Sign) and Z (Zero) flags depending on the result. One byte is operated on when the \( m \) bit = 1, a word (2 bytes) is operated on when \( m = 0 \). (See ORA and AND also.)

Inclusive OR means if \( \text{either or both} \) bits are 1 then the result is 1. Only when both bits are 0 is the result 0.

The truth table used is:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Example:

<table>
<thead>
<tr>
<th>ORA:</th>
<th>Accumulator: 0 0 1 1 0 0 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Memory: 0 1 0 1 0 1 0 1</td>
</tr>
<tr>
<td></td>
<td>Result 0 1 1 1 0 1 1 1</td>
</tr>
</tbody>
</table>

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>ORA $FFff</td>
<td>0D ff FF</td>
</tr>
<tr>
<td>Absolute Long</td>
<td>ORA $00FFff</td>
<td>0F ff FF 00</td>
</tr>
<tr>
<td>Direct Page</td>
<td>ORA $FF</td>
<td>05 FF</td>
</tr>
<tr>
<td>Direct Page Indirect</td>
<td>ORA ($FF)</td>
<td>12 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Long</td>
<td>ORA [$FF]</td>
<td>07 FF</td>
</tr>
<tr>
<td>Immediate</td>
<td>ORA #$FF</td>
<td>09 FF</td>
</tr>
<tr>
<td>Absolute Indexed,X</td>
<td>ORA $FFff,X</td>
<td>1D ff FF</td>
</tr>
<tr>
<td>Absolute Long Indexed,X</td>
<td>ORA $00FFff,X</td>
<td>1F ff FF 00</td>
</tr>
<tr>
<td>Absolute Indexed,Y</td>
<td>ORA $FFff,Y</td>
<td>19 ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed,X</td>
<td>ORA $FF,X</td>
<td>15 FF</td>
</tr>
<tr>
<td>Direct Page Indexed Indirect,X</td>
<td>ORA ($FF,X)</td>
<td>01 FF</td>
</tr>
<tr>
<td>Direct Page Indexed Indirect,Y</td>
<td>ORA ($FF),Y</td>
<td>11 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Long Indexed,Y</td>
<td>ORA [$FF],Y</td>
<td>17 FF</td>
</tr>
<tr>
<td>Stack Relative</td>
<td>ORA $FF,S</td>
<td>03 FF</td>
</tr>
<tr>
<td>Stack Relative Indirect Indexed,Y</td>
<td>ORA ($FF,S),Y</td>
<td>13 FF</td>
</tr>
</tbody>
</table>

529
ORA

Uses
ORA is used primarily as a mask to force 1's in specified bit positions. (See AND to force 0's.) To create the mask, 1 is put in each bit position which is to be forced to one. All other positions are set to 0. For example, here is a routine which will set the high bit on any ASCII data going out through a print routine in your application:

ENTRY LDA DATA,X ; GET CHARACTER TO PRINT
ORA #$80 ; %10000000
     ; SET HIGH BIT
JSR PRINT
RTS

In graphics, ORA has the effect of blending two images on a common background:

```
00011000 00000000 00011000
00011000 00000000 00011000
00011000 00000000 00011000
00011000 ORA 11111111 = 11111111
00011000 11111111 11111111
00011000 00000000 00011000
00011000 00000000 00011000
00011000 00000000 00011000
```

ORA can also be used when checking for a pointer equal to zero that is stored in two bytes in the 8-bit mode, or in four bytes, as would be the case for a handle in the 16-bit mode:

8-Bit Mode, Two Bytes
LDA MEM1 ; LOW-ORDER BYTE
ORA MEM1+1 ; HIGH-ORDER BYTE
BEQ ZERO ; BEQ TAKEN IF BOTH = ZERO

16-Bit Mode, Two Words
LDA MEM1 ; LOW-ORDER WORD
ORA MEM1+2 ; HIGH-ORDER WORD
BEQ ZERO ; BEQ TAKEN IF BOTH = ZERO
PEA: Push Effective Address

Description
Pushes 16-bit address indicated by operand onto stack, regardless of the e, m, or x bits.

Flags & Registers Affected (none)

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>PEA</td>
<td>F4</td>
</tr>
</tbody>
</table>

Uses
PEA is most often used to push an immediate value onto the stack in preparation for a call to an Apple IIgs tool set routine. This value may either be an immediate value, for example the X coordinate of a point, or an address, such as that of a label:

```
PEA $0002 ; PUSH #$2 ON STACK
PEA LABEL ; PUSH LOW WORD OF LABEL ON STACK
PEA `LABEL ; PUSH HIGH WORD OF LABEL ON STACK
```

Note that even though the immediate mode character (#) is not used, the value of LABEL, not its contents, will be pushed on the stack.

PEA could also be used to push an artificial return address on the stack for a subroutine. By using a JMP to access the routine which then ended in an RTS, program execution would resume at the byte immediately after the address pushed on the stack:

```
PEA LABEL-1 ; RETURN ADDRESS-1
JMP SUBR ; ROUTINE TO 'JSR' TO
BRK $00 ; SHOULD NEVER GET HERE...
LABEL NOP ; PROGRAM RESUMES HERE
SUBR NOP ; SOME SORT OF SUBROUTINE
RTS ; RETURNS TO 'LABEL'
```
PEA

As long as the subroutine knows that an alternate return address is on the stack, you can also use PEA with a JSR for two or more possible return addresses:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEA</td>
<td>RTRN2-1</td>
</tr>
<tr>
<td>JSR</td>
<td>SUBR</td>
</tr>
<tr>
<td>RTRN1</td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>PLA</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td>RTRN2</td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td>SUBR</td>
<td>NOP</td>
</tr>
<tr>
<td>BEQ</td>
<td>NORML</td>
</tr>
<tr>
<td></td>
<td>PLA</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>NORML</td>
<td>RTS</td>
</tr>
</tbody>
</table>

532
PEI: Push Effective Indirect Address

Description
Pushes 16-bit address found at the direct-page address (low byte, high byte) in the operand onto the stack, regardless of the e, m, or x bits. The high byte of the address is pushed first, followed by the low byte.

Flags & Registers Affected (none)

Addressing Modes Available

Uses
This instruction allows the program to push a vector or other 16-bit address stored on the direct page onto the stack. For example, suppose the direct page register is set to $8000, and you use the instruction

PEI ($25) ; PUSH EFFECTIVE INDIRECT

The microprocessor will go to locations $8025, $8026 (low-byte, high-byte) for the address held there. Assuming the contents of $8025, $8026 = $1234, the value $1234 would then be pushed on the stack.

Like PEA, PEI can also be used to push an artificial return address on the stack for a subroutine, the only difference being that the source of the return address would be a direct-page pointer. See PEA for examples of the general technique.
**PER**

**PER: Push Effective Relative Address**

**Description**
Pushes a 16-bit address relative to the current program counter, regardless of the e, m or x bits. This is more or less equivalent in function to doing a PEA with a LABEL of an address (as opposed to a constant), except that no absolute address is used in the PER instruction, and it can thus be used to create position-independent code.

**Flags & Registers Affected (none)**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>V</td>
<td>-</td>
<td>B</td>
<td>D</td>
<td>I</td>
<td>Z</td>
<td>C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>PER</td>
<td>62</td>
</tr>
</tbody>
</table>

**USE**
This instruction allows the program to push a 16-bit address determined by the relative offset generated as the operand by the assembler label. For example, consider the following code segment located at $8000:

```
8000: 62 05 00 PER LABEL
8003: 20 08 80 JSR SUBR ; JSR TO A SUBROUTINE
8006: EA LABEL NOP ; MORE OF YOUR PROGRAM
8007: 60 RTS ; THIS PART IS DONE
8008: EA SUBR NOP ; SOME SUBROUTINE
8009: 68 PLA ; RETRIEVE VALUE OF LABEL
800A: EA NOP ; SOME MORE PROGRAM
800B: 60 RTS ; RETURN
```

The microprocessor will take the relative offset of 5 (the operand of the PER instruction), add this to the program counter for the next instruction ($8003), and push the result ($8008) on the stack. The result is the actual address of the assembler operand, LABEL. This value is then available to SUBR. With ProDOS 16 and the System Loader, it is just as easy to use a PEA LABEL to achieve the same result, but you may find this variation useful.

A more likely scenario would be in writing position-independent code that for whatever reason did not have the benefit of the ProDOS 16 System Loader (for example a ProDOS 8 application). In such a case, the PER can be used to generate the current address that the program is running at for an indirect accessing of data.

534
PER DATA ; PUSH WHERE DATA IS ON THE STACK
PLA ; RETRIEVE IT
STA PTR ; WRITE DIRECT-PAGE PTR (2 BYTES)
LDX #$00 ; INITIALIZE INDEX
LOOP
LDA (PTR),X ; GET A CHARACTER
BEQ DONE ; 0 = END OF STRING
NOP ; DO SOMETHING WITH IT
INX ; NEXT CHARACTER
BNE LOOP ; UNTIL X REACHES 0 AGAIN
RTS ; WE'RE DONE HERE

DATA ASC "TEST",00 ; SAMPLE DATA

This code segment can read the characters at DATA, and yet it uses no absolute addressing that would prevent it from running anywhere in memory.

Like PEA, PER can also be used to push an artificial return address on the stack for a subroutine. See PEA for examples of the general technique.

PER can also be used to simulate a JSR to an internal address in position-independent code:

ENTRY PER RTRN-1 ; RETURN ADDRESS-1
PER SUBR-1 ; SUBROUTINE TO "JSR" TO
RTS ; DO EQUIV. OF JSR TO SUBR
RTRN NOP ; YOUR PROGRAM CONTINUES HERE
SUBR NOP ; SOME SUBROUTINE
RTS ; RETURNS TO "RTRN"
PHA

**PHA: Push Accumulator**

**Description**
This pushes the contents of the Accumulator onto the stack. The Accumulator and status register are unaffected. When two bytes are pushed (a word, \( m = 0 \)), the high byte is pushed first; then the low byte is pushed. (See also PLA.)

**Flags & Registers Affected (none)**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>PHA</td>
<td>48</td>
</tr>
</tbody>
</table>

**Uses**
This is one of the most common ways of temporarily storing a byte or two. It is combined with PLA to retrieve the data. Generally speaking, each PHA must be matched by a PLA later in the routine. Otherwise the final RTS of your routine will deliver you, not back to the calling BASIC program or immediate mode, but rather to some unpredictable location.

This and other stack operations are used extensively in sending data to the various tool set routines by pushing the input data onto the stack and then doing a JSL $E10000.

A more obscure use of PHA is to set up an artificial JMP by executing an RTS for which a JSR was never done. Providing two PHAs have been done prior to the RTS, the pseudo-jump will be executed. This is similar to the approach used to adjust the return address of a pending RTS as was discussed in Chapter 11, and the fourth example in the description of PER.
**PHB: Push Data Bank Register**

**Description**
Pushes the single byte value of the data bank register on the stack, regardless of the condition of the e, m, and x bits.

**Flags & Registers Affected (none)**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>PHB</td>
<td>8B</td>
</tr>
</tbody>
</table>

**Uses**
This is done to save the current data bank value, in anticipation of changing and then later restoring it. This could be because your application wishes to access data in another bank or because your subroutine has been called by another program elsewhere. An example of this is the UPDATE routine used in a program that uses the Window Manager.

In general, the procedure for saving and restoring the data bank register would look like this:

```assembly
PHB          ; SAVE CURRENT DATA BANK
PHK
PLB          ; SET DATA BANK = OUR PROGRAM BANK
NOP           ; MORE PROGRAM HERE...
PLB          ; RESTORE DATA BANK TO ORIGINAL
RTL          ; BACK TO WHEREVER
```
**PHD: Push Direct-Page Register**

**Description**
Pushes the 16-bit value of the data bank register on the stack, regardless of the condition of the e, m, or x bits.

**Flags & Registers Affected (none)**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>PHD</td>
<td>0B</td>
</tr>
</tbody>
</table>

**Uses**
This is done to save the current direct-page value, in anticipation of changing and then later restoring it. This could be because a subroutine in your application wishes to use a different direct page than the rest of the application or because your subroutine has been called by another program with its own direct page elsewhere. An example of this is the UPDATE routine used in a program that uses the Window Manager.

In general, the procedure for saving and restoring the direct page register would look like this:

```
PHD            ; SAVE CURRENT DIRECT PAGE
LDA MYDP       ; DP ADDRESS FOR OUR ROUTINE
TCD            ; SET DP = MYDP
NOP            ; MORE PROGRAM HERE . . .
PLD            ; RESTORE DIRECT PAGE TO ORIGINAL
RTS            ; BACK TO WHEREVER
```
PHK: Push Program Bank Register

Description
Pushes the single-byte value of the program bank register on the stack.

Flags & Registers Affected (none)

Addressing Modes Available

Uses
This is usually done to determine the current program bank value, so as to be able to set the data bank equal to the bank the program was running in. This is required for the proper operation of all JSRs and JMPs (non-long address) and internal references to data such as LDA LABEL.

In general, the procedure for using the program bank to set the data bank would look like this:

```
PHB ; SAVE CURRENT DATA BANK
PHK ; PUSH OUR PROGRAM BANK
PLB ; SET DATA BANK = OUR PROGRAM BANK
NOP ; MORE PROGRAM HERE . . .
PLB ; RESTORE DATA BANK TO ORIGINAL
RTL ; BACK TO WHEREVER
```
**PHP: Push Processor Status**

**Description**
This pushes the status register onto the stack for later retrieval. The status register itself is unchanged, and none of the registers are effected. PHP *always* pushes only one byte onto the stack, regardless of the condition of the e, m, and x bits.

**Flags & Registers Affected (none)**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>PHP</td>
<td>08</td>
</tr>
</tbody>
</table>

**Uses**

PHP is done to preserve the status register for later testing for a specific condition. This is handy if you don't want to test a flag right then, but the next instruction would ruin what you want to test for. By putting the status register on the stack, and then later retrieving it, you can test things like the sign flag or carry when it's most convenient.

```
ENTRY CLC ; CLR CARRY
PHP ; SAVE REG
SEC ; SET CARRY
PLP ; RETRIEVE REG
BCC DONE ; (ALWAYS!)
BRK ; (NEVER)

DONE RTS

ENTRY LDA #$00 ; SET Z FLAG
PHP ; SAVE REG
LDA #$FF ; DESTROY
PLP ; RETRIEVE
BEQ DONE ; (ALWAYS!)
BRK ; (NEVER)

DONE RTS
```

If it's ever necessary for your application to disable interrupts, it should use the PHP and PLP instructions to properly save and restore the interrupt status external to your routine. This is done by first saving the Status Register with a PHP instruction, and then later restoring it with PLP. This has the effect
of re-enabling interrupts (CLI) if they, in fact, had been previously enabled, and does not enable them if the interrupt disable bit (bit 2 in the Status Register) had been set (no interrupts).

```
START PHP ; SAVE INTERRUPT STATUS
   SEI ; SET INTERRUPT DISABLE
   NOP ; DO YOUR STUFF HERE...
   PLP ; RESTORE INTERRUPT STATUS
DONE RTS ; BACK TO WHEREVER
```

**Important:** Like the PHA instruction, PHP should always be accompanied by an equal number of PLP instructions to keep the Apple happy. A common cause of problems in programs is to use a PHP to push the Status Register on the stack (1 byte), and to then use a PLA in the 16-bit mode to examine the value later, resulting in pulling a byte of your return address off with the byte put there by the PHP.
PHX

**PHX: Push X Register**

**Description**
This pushes the contents of the X Register onto the stack. The X Register and status register are unaffected. When two bytes are pushed (a word, \( x = 0 \)), the high byte is pushed first, then the low byte. (See also PLX.)

**Flags & Registers Affected (none)**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

**Addressing Modes Available**

- **Mode**
  - Implied Only

- **Syntax**
  - Common: PHX
  - Hex Coding: DA

**Uses**
This is one of the most common ways of temporarily storing the contents of the X Register. It is combined with PLX to retrieve the data.
**PHY: Push Y Register**

**Description**
This pushes the contents of the Y Register onto the stack. The Y Register and status register are unaffected. When two bytes are pushed (a word, \( x = 0 \)), the high byte is pushed first, then the low byte. (See also PLY.)

**Flags & Registers Affected (none)**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>PHY</td>
<td>5A</td>
</tr>
</tbody>
</table>

**Uses**
This is one of the most common ways of temporarily storing the contents of the Y Register. It is combined with PLY to retrieve the data.
**PLA: Pull Accumulator**

**Description**
This is the converse of the PHA instruction. PLA retrieves one byte (8-bit mode, $m = 1$) or one word (16-bit mode, $m = 0$) from the stack and places it in the Accumulator. This accordingly conditions the N (sign) and Z (zero) flags, just as though an LDA instruction had been done.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>PLA</td>
<td>68</td>
</tr>
</tbody>
</table>

**Uses**
This is combined with PHA to retrieve data from the stack. See PHA for an example of this.

**Description**
Pulls a single byte from the stack, and sets the data bank register to that value, regardless of the condition of the $e$, $m$, and $x$ bits.
PLB: Pull Data Bank Register

Description
Pulls a single byte from the stack, and sets the data bank register to that value, regardless of the condition of the e, m and x bits.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>PLB</td>
<td>AB</td>
</tr>
</tbody>
</table>

Uses
This is used to set the current data bank value. This could be because your application wishes to access data in another bank or because your application or subroutine has been called by another program elsewhere. This should always be done as one of the first instructions in a ProDOS 16 application that is running in an indeterminant bank of memory.

In general, the procedure for saving and restoring the data bank register looks like this:

```
PHB ; SAVE CURRENT DATA BANK
PHK ; PUSH OUR PROGRAM BANK
PLB ; SET DATA BANK = OUR PROGRAM BANK
NOP ; MORE PROGRAM HERE . . .
PLB ; RESTORE DATA BANK TO ORIGINAL
RTL ; BACK TO WHEREVER
```

Changing the data bank register does not change direct page references, for example LDA ($25),Y, within the program because by definition these are restricted to bank zero. However, the indirect address indicated on the direct page in bank zero will be interpreted to reside in the bank indicated by the current bank register, unless Indirect Indexed Long addressing is used.
PLD

PLD: Pull Direct-Page Register

Description
Pulls a 16-bit value from the stack and sets the direct page register to that value, regardless of the condition of the e, m, or x bits. If e = 1 (emulation mode), this instruction pulls two bytes from the stack and changes the direct-page register accordingly.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>PLD</td>
<td>2B</td>
</tr>
</tbody>
</table>

Uses
This is used to set the current direct page value from a value stored on the stack. This could be because a subroutine in your application wishes to use a different direct page than the rest of the application or because your subroutine has been called by another program with its own direct page elsewhere. An example of this is the UPDATE routine used in a program that uses the Window Manager.

In general, the procedure for saving and restoring the direct page register would look like this:

```
PHD          ; SAVE CURRENT DIRECT PAGE
LDA MYDP     ; DP ADDRESS FOR OUR ROUTINE
TCD          ; SET DP = MYDP
NOP          ; MORE PROGRAM HERE ...
PLD          ; RESTORE DIRECT PAGE TO ORIGINAL
RTS          ; BACK TO WHEREVER
```
PLP: Pull Processor Status

Description
This is used after a PHP to retrieve the status register data from the stack. The byte is put in the status register, and all the flags are conditioned corresponding to the status of each bit in the byte placed there. The Accumulator and other registers are unaffected. This always pulls a single byte, regardless of the condition of the e, m, and x bits. (See PHP.)

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>PLP</td>
<td>28</td>
</tr>
</tbody>
</table>

Uses
PLP is used to retrieve the status register after a PHP has stored the flags at an earlier time. See PHP for examples.
As with the PHA/PLA set, PLPs should always be matched with a corresponding number of PHP instructions in a one-to-one relationship. Failure to observe this requirement can result in some very strange results.
PLX

PLX: Pull X Register

Description
This pulls one \((x = 1)\) or two \((x = 0)\) bytes off the stack and sets the X Register equal to that value. When two bytes are pulled (a word), the low byte is pulled first, then the high byte is pulled. (See also PHX.)

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>·</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>·</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>PLX</td>
<td>FA</td>
</tr>
</tbody>
</table>

If \(x = 1\) (8-bit mode), the high-order byte of the X Register will always be zero.

Uses
This is one of the most common ways of temporarily storing and retrieving the contents of the X Register. It is combined with PHX to store the data.
PLY: Pull Y Register

Description
This pulls one or two bytes off the stack, depending on the condition of the e and x bits, and it sets the Y Register equal to that value. When two bytes are pulled (a word), the low byte is pulled first, then the high byte is pulled. (See PHY also.)

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

Mode         | Common Syntax | Hex Coding |
-------------|---------------|------------|
Implied Only | PLY           | 7A         |

If x = 1 (8-bit mode), the high-order byte of the Y Register will always be zero.

Uses
This is one of the most common ways of temporarily storing and retrieving the contents of the Y Register. It is combined with PHY to store the data.
REP

REP: Reset (Clear) Bits in Status Register

Description
This clears bits in the Status Register according to which bits are set in the operand. If the $e$ bit is 1 (emulation), bit 4 (break flag) and bit 5 (unused) are ignored. If $e$ is 0 (native), then bits 4 and 5 respectively condition the $m$ and $x$ bits.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$m$ \(x\) only if in native mode ($e = 0$)

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>REP $FF$</td>
<td>C2 FF</td>
</tr>
</tbody>
</table>

Uses
Although this can be used to condition any of the bits in the Status Register, REP is used most often in the native mode to clear the $m$ and/or $x$ bits to enable 16-bit operations for the Accumulator and memory locations and/or the index registers ($X$ and $Y$).

- REP $30$ ; $M$ AND $X = 0$ (16 BITS BOTH)
- REP $20$ ; $M = 0$, $X$ UNAFFECTED
- REP $10$ ; $X = 0$, $M$ UNAFFECTED
- REP $00$ ; NOTHING IS AFFECTED

In the Merlin assembler, the assembler will automatically adjust its internal register size assumptions for future instructions, such as LDA and LDX, after each particular REP or SEP instruction. However, if the $m$ and $x$ bit status within a code segment is ambiguous, or not consistent with the current internal settings, the directive MX may be used.

MX %11 ; $M = 1$, $X = 1$
MX %01 ; $M = 0$, $X = 1$

In the APW assembler, each occurrence of REP or SEP must be accompanied with the appropriate LONGA ON/OFF and/or LONGI ON/OFF directive.
**ROL: Rotate Left**

**Description**
This instruction moves each bit of the Accumulator or memory location specified one position to the left. The carry bit is pushed into position 0 and is replaced by bit 7 (the high-order bit). The N (sign) and Z (zero) flags are also conditioned depending on the result of the shift. ROL shifts either a byte \( m = 1 \) or a word \( m = 0 \). (See also ROR, ASL, and LSR.)

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of bit rotation](image)

ROL
(Rotate One Bit Left)

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulator</td>
<td>ROL</td>
<td>2A</td>
</tr>
<tr>
<td>Absolute</td>
<td>ROL $FFff</td>
<td>2E ff FF</td>
</tr>
<tr>
<td>Direct Page</td>
<td>ROL $FF</td>
<td>26 FF</td>
</tr>
<tr>
<td>Absolute Indexed,X</td>
<td>ROL $FFff,X</td>
<td>3E ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed,X</td>
<td>ROL $FF,X</td>
<td>36 FF</td>
</tr>
</tbody>
</table>

**Uses**
ROL is used to shift multiple byte or word groups as a unit:

```
SHIFT       ASL  MEM       ; SHIFT LOW BYTE (WORD)
            ROL  MEM+1(2)  ; COMPLETE SHIFT ON HIGH-ORDER BYTE (WORD)
            ; MEM, MEM+1(2) TIMES 2
```
**ROR**

**ROR: Rotate Right**

**Description**

This instruction moves each bit of the Accumulator or memory location specified one position to the right. The carry bit is pushed into position 7 (the high order bit) in a byte operation (bit 15 for a word) and is replaced by bit 0. The N (sign) and Z (zero) flags are also conditioned depending on the result of the shift. (See also ROL, ASL, and LSR.)

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulator</td>
<td>ROR</td>
<td>6A</td>
</tr>
<tr>
<td>Absolute</td>
<td>ROR $FFff</td>
<td>6E ff FF</td>
</tr>
<tr>
<td>Direct Page</td>
<td>ROR $FF</td>
<td>66 FF</td>
</tr>
<tr>
<td>Absolute Indexed,X</td>
<td>ROR $FFff,X</td>
<td>7E ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed,X</td>
<td>ROR $FF,X</td>
<td>76 FF</td>
</tr>
</tbody>
</table>

**SHIFT**  ROR MEM+1(2)  ; SHIFT HIGH-ORDER BYTE (WORD)
LSR MEM    ; COMPLETE ON SHIFT LOW BYTE (WORD)
            ; MEM, MEM+1(2) DIVIDED BY 2
RTI: Return from Interrupt

Description
This restores both the program counter and the status register in preparation to resuming the routine being executed at the time of the interrupt. All flags of the status register are reset to the original values. As an RTS is to a JSR, so RTI is the return instruction for an interrupt routine.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>RTI</td>
<td>40</td>
</tr>
</tbody>
</table>

Uses
RTI is used in much the same way that an RTS would be used in returning from a JSR. After an interrupt has been handled and the background operation has been performed, the return is done via the RTI command. Usually the program will want to restore the A, X, and Y registers prior to returning. Because the number of bytes pulled off the stack for the return address depends on whether the processor is in the native or emulation mode, it is essential that the RTI be done with the processor status the same as it was when the interrupt occurred.
RTL/RTS: Return from (Long) Subroutine

**Description**
This restores the program counter to the address stored on the stack plus one, usually the address of the next instruction after the JSR that called the routine. Analogous to a RETURN to a GOSUB in BASIC. (See also JSL/JSR.) RTS only returns to an address within the current program bank; RTL changes the program bank register, and can return to an address anywhere in addressable memory.

**Flags & Registers Affected (none)**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>RTS</td>
<td>60</td>
</tr>
<tr>
<td>Implied Only</td>
<td>RTL</td>
<td>6B</td>
</tr>
</tbody>
</table>

**Uses**
RTS is, surprisingly enough, most often used to return from subroutines. It can on occasion be used to simulate a JMP instruction, by using PHA or PER instructions to put a false return address on the stack and then executing the RTS. See the sections on PHA and PER and Chapter 11 for more details.

An RTS can be POPed one level by the execution of two PLA instructions if in the 8-bit mode (m = 1), or one PLA if in the 16-bit mode (m = 0).

RTL is the return instruction for a JSL (Jump Subroutine Long); it pulls three bytes off the stack (Address Low Byte, Address High Byte, then Bank Byte).
SBC: Subtract with Carry

Description
Subtracts the contents of the memory location or a specified value from the Accumulator. The opposite of the carry is also subtracted, and in this instance, the carry is called a borrow. The N (sign), V (overflow), Z (zero), and C (carry) flags are all conditioned by this operation, and are often used to detect the nature of the result of the subtraction. The result of the subtraction is put back in the Accumulator. The memory location, if specified, is unchanged. SBC works for both the binary and BCD arithmetic modes. The operation involves one byte if \( m = 1 \), two bytes (a word) if \( m = 0 \).

Important: A SEC should always be done prior to the first SBC operation. This is equivalent to clearing the borrow, and is analogous to the CLC done prior to an ADC instruction.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>SBC $FFff</td>
<td>ED ff FF</td>
</tr>
<tr>
<td>Absolute Long</td>
<td>SBC $00FFff</td>
<td>EF ff FF 00</td>
</tr>
<tr>
<td>Direct Page</td>
<td>SBC $FF</td>
<td>E5 FF</td>
</tr>
<tr>
<td>Direct Page Indirect</td>
<td>SBC ($FF)</td>
<td>F2 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Long</td>
<td>SBC [$FF]</td>
<td>E7 FF</td>
</tr>
<tr>
<td>Immediate</td>
<td>SBC #$FF</td>
<td>E9 FF</td>
</tr>
<tr>
<td>Absolute Indexed,X</td>
<td>SBC $FFff,X</td>
<td>FD ff FF</td>
</tr>
<tr>
<td>Absolute Long Indexed,X</td>
<td>SBC $00FFff,X</td>
<td>FF ff FF 00</td>
</tr>
<tr>
<td>Absolute Indexed,Y</td>
<td>SBC $FFff,Y</td>
<td>F9 ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed,X</td>
<td>SBC $FF,X</td>
<td>F5 FF</td>
</tr>
<tr>
<td>Direct Page Indexed Indirect,X</td>
<td>SBC ($FF,X)</td>
<td>E1 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Indexed,Y</td>
<td>SBC ($FF),Y</td>
<td>F1 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Long Indexed,Y</td>
<td>SBC [$FF],Y</td>
<td>F7 FF</td>
</tr>
<tr>
<td>Stack Relative</td>
<td>SBC $FF,S</td>
<td>E3 FF</td>
</tr>
<tr>
<td>Stack Relative Indirect Indexed,Y</td>
<td>SBC ($FF,S),Y</td>
<td>F3 FF</td>
</tr>
</tbody>
</table>

Uses
SBC is used for subtracting a constant or memory value from the contents of the Accumulator.
SBC

One-Byte (Word) Subtraction:
ENTRYSEC; PREPARE FOR SUBTRACTION
LDA MEM; GET 1ST VALUE
SBC #$80; SUBTRACT #$80
STA RSLT; STORE RESULT
DONE RTS

Two-Byte (Word) Subtraction:
ENTRY SEC; PREPARE FOR SUBTRACTION
LDA MEM; GET LOW-ORDER BYTE (WORD)
SBC #$80; SUBTRACT #$80
STA RSLT; SAVE LOW-ORDER BYTE (WORD)
LDA MEM + 1(2); GET HIGH-ORDER BYTE (WORD)
SBC #$00; SUBTRACT HIGH-ORDER BYTE (WORD) OF #$80
STA RSLT + 1(2); SAVE HIGH-ORDER RESULT
DONE RTS
**SEC: Set Carry**

**Description**
This sets the carry flag of the status register.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>SEC</td>
<td>38</td>
</tr>
</tbody>
</table>

**Uses**
SEC is usually used just prior to a SBC operation. (See SBC.)

The carry is set prior to using the instruction XCE to set the 65816 microprocessor to the emulation mode.

```
SEC
XCE ; EMULATION MODE
```

The carry is also used very often to indicate an error when returning from a call to a routine, as is done by ProDOS and the Apple II GS tool sets. In these instances, the carry is set to indicate an error. This would be detected by the calling program upon return from the routine.

```
JSR PRODOS
BCS ERROR
```

SEC is also sometimes used to force a branch. For example:

```
SEC
BCS ADDRESS ; (ALWAYS)
```
**SED: Set Decimal Mode**

**Description**
SED sets the 65816 to the Binary Coded Decimal (BCD) mode, in preparation for an ADC or SBC operation.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>SED</td>
<td>F8</td>
</tr>
</tbody>
</table>

**Uses**
BCD math is used to encode decimal data in a flexible number of bytes. In this mode, each four bits (nibble) of a byte represent one digit of a base ten number. Although not discussed in this book, here is a brief example of a BCD addition operation:

```
ENTRY    SED ; SET DEC MODE
          CLC ; PREPARE FOR ADDITION
LDA      #$25 ; %00101001 = #25
ADC      #$18 ; %00011000 = #18
STA      RSLT ; RSLT = %01000011 = #43
          CLD ; CLR DEC MODE
DONE     RTS
```
SEI: Set Interrupt Disable

Description
SEI is used to disable the interrupt response to an IRQ (a maskable interrupt). This does not disable the response to an NMI (Non-Maskable Interrupt) or RESET.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>SEI</td>
<td>78</td>
</tr>
</tbody>
</table>

Uses
SEI is automatically set whenever an interrupt occurs so that no further interrupts can disturb the system while it is going through the $FFFE,FFFF vector path. ProDOS typically does a SEI/CLI operation when entering and exiting from the routines that read and write data on the disk, so interrupts do not interfere with the highly timing-dependent disk read/write routines. (See PHP also.)
SEP

SEP: Set Bits in Status Register

Description
This sets bits in the Status Register according to which bits are set in the operand. If the e bit is 1 (emulation), bit 4 (break flag) and bit 5 (unused) are ignored. If e = 0 (native), then bits 4 and 5 condition the m and x bits, respectively.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

m \( \times \) only if in native mode, e = 0

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>SEP $FF</td>
<td>E2 FF</td>
</tr>
</tbody>
</table>

Uses
Although this can be used to condition any of the bits in the Status Register, SEP is used most often in the native mode to set the m and/or x bits to enable 8-bit operations for the Accumulator and memory locations and/or the index registers (X and Y).

SEP $30 ; M AND X = 1 (8 BITS BOTH)
SEP $20 ; M = 1, X UNAFFECTED
SEP $10 ; X = 1, M UNAFFECTED
SEP $00 ; NOTHING IS AFFECTED

In the Merlin assembler, the assembler will automatically adjust its internal register size assumptions for future instructions (like LDA, LDX, and so on) that follow a particular REP or SEP instruction. However, if the m- and x-bit status within a code segment is ambiguous or inconsistent with the current internal settings, the directive MX may be used.

MX %11 ; M = 1, X = 1
MX %01 ; M = 0, X = 1

In the APW assembler, each occurrence of REP or SEP must be accompanied with the appropriate LONGA ON/OFF and/or LONGI ON/OFF directive.
STA: Store Accumulator

Description
Stores the contents of the Accumulator into the specified memory location. The contents of the Accumulator are not changed, nor are any of the status register flags. One byte is stored if \( m = 1 \) (8-bit mode); two bytes are stored if \( m = 0 \) (16-bit mode).

Flags & Registers Affected (none)

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>STA $Fff</td>
<td>8D ff FF</td>
</tr>
<tr>
<td>Absolute Long</td>
<td>STA $00Ffff</td>
<td>8F ff FF 00</td>
</tr>
<tr>
<td>Direct Page</td>
<td>STA $FF</td>
<td>85 FF</td>
</tr>
<tr>
<td>Direct Page Indirect</td>
<td>STA ($FF)</td>
<td>92 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Long</td>
<td>STA [$FF]</td>
<td>87 FF</td>
</tr>
<tr>
<td>Absolute Indexed,X</td>
<td>STA $Ffff,X</td>
<td>9D ff FF</td>
</tr>
<tr>
<td>Absolute Long Indexed,X</td>
<td>STA $00Ffff,X</td>
<td>9F ff FF 00</td>
</tr>
<tr>
<td>Absolute Indexed,Y</td>
<td>STA $Ffff,Y</td>
<td>99 ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed,X</td>
<td>STA $FF,X</td>
<td>95 FF</td>
</tr>
<tr>
<td>Direct Page Indexed Indirect,X</td>
<td>STA ($FF,X)</td>
<td>81 FF</td>
</tr>
<tr>
<td>Direct Page Indexed Indirect,Y</td>
<td>STA ($FF),Y</td>
<td>91 FF</td>
</tr>
<tr>
<td>Direct Page Indirect Long Indexed,Y</td>
<td>STA [$FF],Y</td>
<td>97 FF</td>
</tr>
<tr>
<td>Stack Relative</td>
<td>STA $FF,S</td>
<td>83 FF</td>
</tr>
<tr>
<td>Stack Relative Indirect Indexed,Y</td>
<td>STA ($FF,S),Y</td>
<td>93 FF</td>
</tr>
</tbody>
</table>

Uses
STA is another highly used instruction, being used at the end of many operations to put the final result into a memory location.

In general, the LDA/STA combination is used to transfer bytes from one location to another.
STP

STP: Stop Processor

Description
This brings the whole system to a nearly permanent halt by stopping the microprocessor's clock input. RESET is the only way to resume operation.

Flags & Registers Affected (none)

```
N V - B D I Z C           Acc X Y Mem
```

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>STP</td>
<td>DB</td>
</tr>
</tbody>
</table>

Uses
STP is really designed for other hardware devices that may use the 65816 microprocessor. When the clock input is stopped, the power consumption of the 65816 drops almost to zero. In the case of the Apple IIGS, this really doesn't save much because of all those other peripheral cards, the video circuitry, the power supply itself, the monitor sitting on top of the computer, and so on.
**STX: Store the X Register**

**Description**
Stores the contents of the X Register in the specified memory location. The X Register is unchanged and none of the status register flags are affected. One byte is stored if \( x = 1 \) (8-bit register); two bytes (a word) are stored if \( x = 0 \) (16-bit registers).

**Flags & Registers Affected (none)**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>STX $FFff</td>
<td>8E ff FF</td>
</tr>
<tr>
<td>Direct Page</td>
<td>STX $FF</td>
<td>86 FF</td>
</tr>
<tr>
<td>Direct Page Indexed,Y</td>
<td>STX $FF,Y</td>
<td>96 FF</td>
</tr>
</tbody>
</table>

**Uses**
STX is another alternative to using LDA/STA to transfer data. In some instances, the Accumulator will already hold a value you wish to preserve while you transfer another byte or word. The X Register is also used by the Monitor GETLN routine to return the length of the input string.
STY

**STY: Store the Y Register**

**Description**
STY stores the contents of the Y Register in the specified memory location. The Y Register is unchanged, and none of the status register flags are affected. One byte is stored if \( x = 1 \) (8-bit register), two bytes (a word) if \( x = 0 \) (16-bit registers).

**Flags & Registers Affected** (none)

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>STY $FFff</td>
<td>8C ff FF</td>
</tr>
<tr>
<td>Direct Page</td>
<td>STY $FF</td>
<td>84 FF</td>
</tr>
<tr>
<td>Direct Page Indexed,X</td>
<td>STY $FF,X</td>
<td>94 FF</td>
</tr>
</tbody>
</table>

**Uses**
STY is used to store the value of the Y Register, usually from within string or data scanning loops. For example, here is a routine which returns the position of the first control character in a block of data.

```
ENTRY LDY #$00 ; ZERO COUNTER
LOOP  LDA DATA,Y ; GET CHARACTER
      BEQ NOTF ; CHAR = 0 = END OF STRING
      CMP #$20 ; 'SPC'
      BCS NXT ; CHR > CTRL'S
      FOUND STY POS ; SAVE Y-REG
      DONE RTS

NXT  INY ; Y = Y + 1
      BNE LOOP ; TILL Y=0 AGAIN.
      BRA DONE ; BRANCH ALWAYS
      NOTF LDY #$FF ; FLAG NOTFOUND
      BRA FOUND ; BRANCH ALWAYS
```
STZ: Store Zero in Memory

Description
STZ stores a one-byte ($m = 1$) or two-byte ($m = 0$) zero value in the memory location indicated by the operand.

Flags & Registers Affected (none)

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>STZ $FFff</td>
<td>9C ff FF</td>
</tr>
<tr>
<td>Direct Page</td>
<td>STZ $FF</td>
<td>64 FF</td>
</tr>
<tr>
<td>Absolute Indexed,X</td>
<td>STZ $FFff,X</td>
<td>9E ff FF</td>
</tr>
<tr>
<td>Direct Page Indexed,X</td>
<td>STZ $FF,X</td>
<td>74 FF</td>
</tr>
</tbody>
</table>

Uses
STZ is a convenient alternative to using LDA/STA to zero a memory location. In some instances, the Accumulator will already hold a value you wish to preserve while you zero out another byte or word.

LDA #$00
STA MEM

is replaced by

STZ MEM
TAX

**TAX: Transfer Accumulator to X Register**

**Description**

Puts contents of Accumulator into the X Register. Doesn’t affect the Accumulator.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>TAX</td>
<td>AA</td>
</tr>
</tbody>
</table>

The actual data transferred depends significantly on the condition of the e, m, and x bits at the moment of transfer:

- **e = 1 = Emulation**
  - m x Accumulator X Register
  - 1 (8) 1 (8) $FFff \rightarrow $00ff

  In emulation mode, both the Accumulator and the X Register are one byte in size. Although the Accumulator has a hidden B portion (the high-order byte), this is not transferred to the X Register because, in the 8-bit mode, the high-order byte of the X Register is always forced to zero. Thus, the result in the X Register is always $00ff, regardless of the high-order byte of the Accumulator.

- **e = 0 = Native**
  - m x Accumulator X Register
  - 0 (16) 0 (16) $FFff \rightarrow $FFff

  Transfer as expected.

- m x Accumulator X Register
  - 0 (16) 1 (8) $FFff \rightarrow $00ff

  The high-order byte of the X Register is always zero in the 8-bit mode.

- m x Accumulator X Register
  - 1 (8) 0 (16) $FFff \rightarrow $FFff
Although the Accumulator is in the 8-bit mode, both bytes are transferred (A and B portions) if the X Register is in the 16-bit mode. This is a common source of program bugs. If the Accumulator is in the 8-bit mode, it is possible that the Accumulator is still carrying along some value (not necessarily zero) from millions of previous instructions in the hidden B register. You should never assume the B register in the Accumulator is zero unless you have explicitly set it that way.

\[
\begin{array}{ccc}
m & x & \text{Accumulator} & X \text{ Register} \\
1 (8) & 1 (8) & $\text{FF}ff$ & \rightarrow & $\text{00}ff$
\end{array}
\]

This is equivalent to the e = 1 (emulation mode) transfer.

Uses
Most simply, TAX is used for transferring data in the manner which it implies.
TAY: Transfer Accumulator to Y Register

Description
Puts contents of Accumulator into the Y Register. Does not effect the Accumulator. See TAX for a discussion of the impact of the condition of the e, m, and x bits.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>TAY</td>
<td>A8</td>
</tr>
</tbody>
</table>

Uses
TAY is used for transferring data from the Accumulator to the Y Register.
**TCD: Transfer Accumulator to Direct-Page Register**

**Description**
Transfers a 16-bit value from the Accumulator and sets the direct-page register to that value, regardless of the condition of the e, m, or x bits.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>TCD</td>
<td>5B</td>
</tr>
</tbody>
</table>

**Uses**
This is used to set the current direct-page value from a value stored in the Accumulator. This could be because a subroutine in your application wishes to use a different direct page then the rest of the application, or because your subroutine has been called by another program with its own direct page elsewhere. An example of this is the UPDATE routine used in a program that uses the Window Manager.

In general, the procedure for saving and restoring the direct-page register would look like this:

```
PHD             ; SAVE CURRENT DIRECT PAGE
LDA MYDP        ; DP ADDRESS FOR OUR ROUTINE
TCD             ; SET DP = MYDP
NOP             ; MORE PROGRAM HERE ...
PLD             ; RESTORE DIRECT PAGE TO ORIGINAL
RTS             ; BACK TO WHEREVER
```
**TCS: Transfer Accumulator to Stack**

**Description**
This puts the contents of the Accumulator into the stack pointer. None of the status register flags are affected, nor is the Accumulator itself changed. How many bytes are transferred depends on the condition of the $e$ and $x$ bits.

**Flags & Registers Affected (none)**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>TCS</td>
<td>1B</td>
</tr>
</tbody>
</table>

If the processor is in the native mode ($e = 0$), then 16 bits are transferred from the Accumulator to the Stack Pointer regardless of the condition of the $m$ bit. If the $e = 1$ (emulation mode), then only the lower byte (A) of the Accumulator is transferred to the Stack Pointer, because the high-order byte of the Stack Pointer is locked to $01$ (page $100$).

**Uses**
This provides a way of directly setting the stack pointer.
TDC: Transfer Direct Page Register to Accumulator

Description
Transfers the 16-bit value in the direct page register to the Accumulator, regardless of the condition of the e, m, or x bits.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>TDC</td>
<td>7B</td>
</tr>
</tbody>
</table>

Uses
This is used to determine, and optionally to save, the current direct-page value. This could be because a subroutine in your application wishes to use a different direct page then the rest of the application, or because your subroutine has been called by another program with its own direct page elsewhere. An example of this is the UPDATE routine used in a program that uses the Window Manager.

In general, the procedure for saving and restoring the direct-page register using TDC would look like this:

```
TDC          ; PUT CURRENT DIRECT PAGE IN ACC.
PHA          ; SAVE CURRENT DIRECT PAGE
LDA MYDP     ; DP ADDRESS FOR OUR ROUTINE
TCD          ; SET DP = MYDP
NOP          ; MORE PROGRAM HERE . . .
PLD          ; RESTORE DIRECT PAGE TO ORIGINAL
RTS          ; BACK TO WHEREVER
```
TRB

**TRB: Test and Reset (Clear) Memory Bits**

**Description**
Clears bits in memory corresponding to each bit set in the Accumulator. Also conditions the Z flag depending on the result in the same way as the BIT instruction. That is, Z will be clear (BNE will work) if any of the tested bits were set before being cleared. Whether the operation involves a byte or a word depends on the status of the e and m bits.

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

**Addressing Modes Available**

**Mode** | **Common Syntax** | **Hex Coding**
---|---|---
Absolute | TRB $Fff | 1C ff FF
Direct Page | TRB $FF | 14 FF

**Uses**
TRB is an alternative to using the AND instruction to force zeros in given bit positions. It has the advantage of combining the function of the BIT instruction in telling you whether any of the bits you just cleared were set beforehand. Use an operand with only one bit set for cases where you want to test and clear a specific bit. (See also TSB.) Operates on one byte if m = 1, two bytes if m = 0.

```
LDA #$80 ; %10000000
TRB MEM ; CLEAR BIT 7
BNE SET ; BIT SET BEFOREHAND
```
TSB: Test and Set Memory Bits

Description
Sets bits in memory corresponding to each bit set in the Accumulator. Also conditions the Z flag depending on the result in the same way as the BIT instruction. That is, Z will be set (BEQ will work) only if all of the tested bits were clear before being set. Operates on one byte if m = 1, on two bytes if m = 0.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>TSB $FFff</td>
<td>0C ff FF</td>
</tr>
<tr>
<td>Direct Page</td>
<td>TSB $FF</td>
<td>04 FF</td>
</tr>
</tbody>
</table>

Uses
TSB is an alternative to using the ORA instruction to force 1's in given bit positions. It has the advantage of combining the function of the BIT instruction in telling you whether any of the bits you just set were clear beforehand. Use an operand with only one bit set for cases where you want to test and set a specific bit. (See also TRB.)

```
LDA #$80 ; %10000000
TSB MEM  ; SET BIT 7
BEQ CLR  ; BIT CLEAR BEFOREHAND
```
TSC

TSC: Transfer Stack Pointer to Accumulator

Description
This puts the 16-bit contents of the Stack Pointer into the Accumulator, regardless of the condition of the e or m bits. The N (sign) and Z (zero) flags are conditioned. The stack pointer is unchanged. When e = 1 (emulation), the high-order byte of the Accumulator will always be set to $01, since that is the high byte of the Stack Pointer in the emulation mode (page $100).

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>TSC</td>
<td>3B</td>
</tr>
</tbody>
</table>

Uses
TSC is used in examining the value of the stack pointer at a particular moment. (See also TSX.)
TSX: Transfer Stack Pointer to X Register

Description
This puts the contents of the Stack Pointer into the X Register. The N (sign) and Z (zero) flags are conditioned. The stack pointer is unchanged. If x = 0, two bytes are transferred, otherwise only the low-order byte of the Stack Pointer will be used.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>TSX</td>
<td>BA</td>
</tr>
</tbody>
</table>

Uses
The most obvious use of TSX is in preserving the value of the stack pointer at a particular moment.

Another use for TSX is in retrieving data from the stack without having to do a PLA instruction. Although stack relative addressing can be used to access data on the stack, TSX can be used to retrieve information that is officially lost at that point. This lets you retrieve data that is lower in memory than the current stack pointer, and which would be overwritten by the next PHA instruction.

One example of this is in using a JSR to a known RTS in the Monitor for no other purpose than to be able to immediately retrieve the otherwise lost return address. This is done so that position-independent code has a way of finding out where it’s currently located. Although there is really not room here for an in-depth explanation, here’s the routine that uses a JSR to a known RTS, and then examines the stack to determine where in memory it is currently executing.

ENTRY PHP ; SAVE INTERRUPT STATUS
SEI ; SET INTERRUPT DISABLE
JSR RETURN ; $FF58
TSX ; GET STACK POINTER
LDA STACK,X ; $100,X
STA PTR+1 ; SAVE HIGH BYTE OF RETURN ADDRESS
DEX ; MOVE TO NEXT POSITION
LDA STACK,X ; GET LOW BYTE OF RETURN ADDRESS
STA PTR ; (PTR) = ENTRY + 2.
PLP ; RESTORE INTERRUPT STATUS
DONE RTS

TSX
This technique was originally developed for the 6502 microprocessor, and code similar to this is used on peripheral cards that must determine which slot they're assigned to. On the Apple II GS with the 65816 microprocessor, this is not really needed because the PER instruction will put an address on the stack without the worry of illegal stack use, or conflict with interrupts. (See PER.)

**Caution:** Most Step and Trace utilities will not properly trace code like this because of the somewhat illegal use of the stack. Strictly speaking, good programming principles dictate that once data is officially off the stack, it is counted as being effectively lost. This is especially true in the case of interrupts, where an interrupt in the middle of the dummy JSR, RTS, and retrieval process could produce a completely invalid result in PTR,PTR+1. That is why our routine temporarily disables interrupts while it examines the dead area of the stack.
TXA: Transfer X Register to Accumulator

Description
This puts the contents of the X Register into the Accumulator, and thus conditions the status register just as if a LDA instruction had been executed. The X Register is unaffected by the operation. (See also TAX.)

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>TXA</td>
<td>8A</td>
</tr>
</tbody>
</table>

The actual data transferred depends significantly on the condition of the e, m, and x bits at the moment of transfer:

\[ e = 1 = \text{Emulation} \]

\[ m \times \text{X Register} \rightarrow \text{Accumulator} \]

1 (8) 1 (8) $00ff \rightarrow $FFff

In emulation mode, both the Accumulator and the X Register are one byte in size. However, the Accumulator has a hidden B portion (the high-order byte), that is not overwritten by the X Register because it is in the 8-bit mode. Thus, the result in the Accumulator still contains the high-order byte, regardless of the high-order byte of zero in the X Register.

\[ e = 0 = \text{Native} \]

\[ m \times \text{X Register} \rightarrow \text{Accumulator} \]

0 (16) 0 (16) $FFff \rightarrow $FFff

Transfer as expected.

\[ m \times \text{X Register} \rightarrow \text{Accumulator} \]

0 (16) 1 (8) $xxff \rightarrow $FFff

The high-order byte of Accumulator (B) is not overwritten by the high-order byte of the X Register.

\[ m \times \text{X Register} \rightarrow \text{Accumulator} \]

1 (8) 0 (16) $00ff \rightarrow $00ff
TXA

Because the Accumulator is in the 16-bit mode, both bytes of the X Register are transferred even though the X Register is in the 8-bit mode. This is a common source of program bugs. Although you may have had some value in the B portion of the Accumulator, it will be overwritten by TXA even if \( x = 1 \) (8-bit mode).

\[
\begin{array}{c|c|c|c}
 m & x & X \text{ Register} & Accumulator \\
 1 \ (8) & 1 \ (8) & \$00ff & \rightarrow \$FFff \\
\end{array}
\]

This is equivalent to the \( e = 1 \) (emulation mode) transfer. The B portion of the Accumulator retains its value.

Uses
TXA provides a way of retrieving the value in the X Register for appropriate processing by the program. In the case of string-related routines, this is often the length of the string just entered or scanned. The Accumulator can then go about the things it does so well in terms of putting the value into the most useful part of memory. Notice that there are more addressing modes available to the STA command, not to mention the overall powers granted the Accumulator in terms of logical operators and so forth.
TXS: Transfer X Register to Stack Pointer

Description
This puts the contents of the X Register into the Stack Pointer. None of the status register flags are affected, nor is the X-Register itself changed. If \( e = 1 \) (emulation), then only the low-order byte of the X Register is transferred, since the stack is fixed to page 1 ($100-$1FF). If \( e = 0 \) (native) and \( x = 0 \) (16-bit registers), then both bytes of the X Register are used. If \( e = 0 \) and \( x = 1 \) (8-bit register), then the low-order byte of the X Register is transferred, and the high-order byte is forced to zero. This puts the stack in page zero of bank zero.

Flags & Registers Affected (none)

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>TXS</td>
<td>9A</td>
</tr>
</tbody>
</table>

Uses
This provides a way of directly setting the stack pointer.
TXY

TXY: Transfer X Register to Y Register

Description
This puts the contents of the X Register into the Y Register, and conditions the status register just as if a LDY instruction had been executed. The X Register is unaffected by the operation. One byte is transferred if x = 1; two bytes are transferred if x = 0.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>TXY</td>
<td>9B</td>
</tr>
</tbody>
</table>

Uses
TXY provides a way of transferring the contents of the X Register to the Y Register. This could either be because you want to temporarily store the contents of the X Register while you use it for something else, or because you want to use the index in the X Register in an addressing mode only supported by the Y Register. For example:

```
LDX #$00          ; INITIALIZE COUNTER
LOOP           LDA BUFF,X ; GET CHARACTER FROM ABSOLUTE ADDR.
                TXY         ; SET Y = X
                STA (PTR),Y ; STORE USING INDIRECT ADDRESSING
                INX
                CPX LEN       ; LENGTH OF STRING YET?
                BCC LOOP      ; NOPE
DONE           RTS
```
TYA: Transfer Y Register to Accumulator

Description
This puts the contents of the Y Register into the Accumulator, thus conditioning the status register as if an LDA instruction had been executed. The Y Register is unaffected by the operation. The actual data transferred depends on the condition of the e, m, and x bits. See the discussion of TAX for detailed examples.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>TYA</td>
<td>98</td>
</tr>
</tbody>
</table>

Uses
TYA provides a way of retrieving the value in the Y Register for appropriate processing by the program. This comes in handy when scanning a data block, and information as to certain locations is to be processed. As mentioned under TXA, the Accumulator has far greater flexibility than the Y Register in terms of addressing modes, logical operators available, and so on.
TYX

TYX: Transfer Y Register to X Register

Description
This puts the contents of the Y Register into the X Register, and conditions the status register just as if a LDX instruction had been executed. The Y Register is unaffected by the operation. One byte is transferred if \( x = 1 \), and two bytes are transferred if \( x = 0 \).

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>TYX</td>
<td>BB</td>
</tr>
</tbody>
</table>

Uses
TYX provides a way of transferring the contents of the Y Register to the X Register. This could either be because you want to temporarily store the contents of the Y Register while you use it for something else, or because you want to use the index in the Y Register in an addressing mode only supported by the X Register. For example:

```
LDY #$00 ; INITIALIZE COUNTER
LOOP LDA (PTR),Y ; GET CHARACTER USING INDIRECT
    TYX ; SET X = Y
    STA BUFF,X ; STORE USING ABSOLUTE WITH INDEX
    INY
CPY LEN ; LENGTH OF STRING YET?
    BCC LOOP ; NOPE
DONE RTS
```

582
**WAI: Wait for Interrupt**

**Description**
This brings current program execution to a stop until the next interrupt or RE-SET occurs.

**Flags & Registers Affected (none)**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>WAI</td>
<td>CB</td>
</tr>
</tbody>
</table>

**Uses**
WAI is designed for use with other hardware devices that may use the 65816 microprocessor, or for peripheral cards or specialized devices on the Apple II GS that require synchronized interrupts. While the processor is waiting, the power consumption of the 65816 is reduced (though not as low as for the STP instruction). The WAI instruction can also be used to cause the microprocessor to respond nearly instantly to an interrupt event. Ordinarily, if an interrupt occurs while an instruction is executed, several cycles of the microprocessor may elapse while that instruction is completed before the interrupt is processed. The WAI instruction allows hardware designers to insure that the microprocessor will respond instantly to an interrupt.
WDM

WDM: William D. Mensch, Jr.

Description
This instruction is included as a bridge to a possible, though not yet implemented, expanded instruction set. The characters WDM are the initials of the designer of the 65816, William D. Mensch, Jr., of the Western Design Center, Inc. WDM is a minimum two-byte instruction. Bytes following the initial opcode are subject to future definition, and may involve one or more additional bytes.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only:</td>
<td>WDM ??</td>
<td>42 ??</td>
</tr>
</tbody>
</table>
XBA: Exchange B and A Accumulators

Description
This swaps the A and B portions of the Accumulator, regardless of the condition of the e or m bits. B is the high-order byte; A, the low-order. The resulting condition of the N and Z flags depends on the contents of the A portion of the Accumulator result. N is set (BMI) if bit 7 is set, Z is set (BEQ) if the A Accumulator is zero.

Flags & Registers Affected

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addressing Modes Available

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>XBA</td>
<td>EB</td>
</tr>
</tbody>
</table>

Uses
XBA can be used to swap the hidden B Accumulator in the 8-bit mode with the A Accumulator. It can also be used to swap the low- and high-order bytes in the 16-bit mode (same operation, different interpretation).

Although it’s not particularly more efficient than just adding $100 with the ADC instruction, here’s a way to increment the high-order byte of an address in the Accumulator:

```
LDA MEM ; GET VALUE FROM MEMORY
XBA    ; PUT HIGH BYTE IN "A"
INC    ; ADD 1
XBA    ; NET: ACC = ACC + $100
STA MEM ; SAVE RESULT
```

This is equivalent to

```
CLC    ; PREPARE FOR ADDITION
LDA MEM ; GET VALUE FROM MEMORY
ADC #$100 ; ADD $100
STA MEM ; SAVE RESULT
```
**XCE: Exchange Carry and Emulation Bits**

**Description**
This swaps the contents of the Carry flag with the emulation bit. If the Carry is clear, the result is $e = 0$, and the microprocessor is put into the native mode (16 bits enabled, but not yet selected). If the Carry is set, the result is $e = 1$, and the microprocessor is put in the emulation mode (8 bits only for Accumulator, index registers, and memory operations).

**Flags & Registers Affected**

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>-</th>
<th>B</th>
<th>D</th>
<th>I</th>
<th>Z</th>
<th>C</th>
<th>Acc</th>
<th>X</th>
<th>Y</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$m \ x$

If $e$ is cleared (native), then $m$ and $x$ are set to 1 (8 bit). If $e$ is set (emulation), $m$ and $x$ disappear in function, and are replaced by the Break flag for bit 4 and by an unused bit for bit 5.

**Addressing Modes Available**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>XCE</td>
<td>FB</td>
</tr>
</tbody>
</table>

**Uses**
XCE is the only way to switch the microprocessor from the native to emulation mode, and vice versa. Here's a review of the action in each direction.

**Native to Emulation**

- SEC
- XCE ; $e = 1$ = EMULATION

**Other Effects:**

- **m = 1**
  - Accumulator/Memory operations locked to 8 bits. Accumulator continues to hold both A and B portions intact.

- **x = 1**
  - Index Register size locked to 8 bits. The high-order bytes of $X$ and $Y$ are set to zero.

- **Stack**
  - The stack is set to page one ($$1xx$$), bank zero, and is limited to one page in size ($$100$$ to $$1FF$$).
Direct Page, Program Bank, Data Bank

The Direct Page, Program Bank, and Data Bank registers do not explicitly change. Although ProDOS 8 and other Apple II software may expect the Direct Page, Program Bank, and Data Bank values to be $0000, it is up to the application to make sure this is the case if any of these have been reset by the application, and a switch is made to emulation for the purpose of executing a Monitor routine, Applesoft BASIC program, and so forth.

Emulation to Native

CLC
XCE

; e = 0 = NATIVE

Other Effects:

m = 1

Accumulator/Memory operations default to 8 bits. Accumulator continues to hold both A and B portions intact. Data size may then be changed with REP or SEP instructions at will.

x = 1

Index Register size defaults to 8 bits. The high-order bytes of X and Y are set to zero. Register size may then be changed with REP or SEP instructions at will.

Stack

The stack is left set to page one ($1xx), bank zero, and but is no longer limited to one page in size. Program may then use TCS or any other relevant instruction to change the stack location as desired to anywhere in bank 0.

Direct Page, Program Bank, Data Bank

The Direct Page, Program Bank and Data Bank Registers do not explicitly change. Although ProDOS 8 and other Apple II software may have set the Direct Page, Program Bank, and Data Bank values to be $0000, the application may change these as desired. ProDOS 16 automatically sets the Program Bank Register when an application is loaded and run, and assigns $400 bytes somewhere in bank 0 for the Direct Page ($100 bytes) and the Stack ($300 bytes). Under ProDOS 16, the program must set its own Data Bank Register as needed.
Appendix B

The Apple II GS Monitor

Early chapters in this book introduced the Apple II GS Monitor and some of the commands that make life easier for the assembly language programmer. This appendix provides a quick reference guide for some of the additional and useful commands in the Monitor not discussed in the main body of the book.

The Monitor is entered by typing CALL-151 from Applesoft BASIC. The mini-assembler is started by typing the exclamation mark (!) and is described in Chapter 3. You can exit the mini-assembler by typing Return alone on a line. You can return to Applesoft BASIC from the monitor by typing Control-C or by pressing Q (for Quit).

The following sections organize the Monitor commands by function. Assume that each sample instruction is terminated by pressing Return.

### Memory Operations

<table>
<thead>
<tr>
<th>Sample Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>02/1234</td>
<td>Set current examination bank to $02.</td>
</tr>
<tr>
<td>1234</td>
<td>Display contents of location $1234.</td>
</tr>
<tr>
<td>1234.5678</td>
<td>Display hex and ASCII values from memory in the range.</td>
</tr>
<tr>
<td>$1234–$5678</td>
<td>Sets current examination bank to $02, and dumps contents of $1234 through $5678 in bank $02.</td>
</tr>
<tr>
<td>{Return}</td>
<td>Return alone displays the next 8 or 16 bytes starting at the current address depending on whether the screen is in 40 or 80 columns.</td>
</tr>
<tr>
<td>02/1234.5678</td>
<td>Sets current examination bank to $02, and dumps contents of $1234 through $5678 in bank $02.</td>
</tr>
<tr>
<td>1234: 56</td>
<td>Stores $56 in location $1234 in the current examination bank.</td>
</tr>
<tr>
<td>02/1234: 56</td>
<td>Stores $56 in location $1234 in bank $02. Note: The bank byte may be included in any of the following memory instructions as you wish.</td>
</tr>
<tr>
<td>Control-X (No C/R)</td>
<td>Terminates a memory dump in progress.</td>
</tr>
<tr>
<td>1234: “A”</td>
<td>Stores the ASCII value ($41 or $C1) for the letter A in location $1234. Whether the high bit is set or not depends on the status of the filter value.</td>
</tr>
<tr>
<td>FF = F</td>
<td>Change filter value for ASCII characters to $FF for high bit on.</td>
</tr>
<tr>
<td>7F = F</td>
<td>Change filter value for ASCII characters to $7F for high bit off.</td>
</tr>
</tbody>
</table>
Appendix B

1234: 01 02 03
Store the values $01, $02, $03 in locations $1234, $1235, and $1236.

1234: "TEST"
Store the ASCII values for the characters T-E-S-T in locations $1234, $1235, $1236, and $1237, with the high bit conditioned by the filter value.

1234: 'ABCD'
Store the ASCII values for the characters in the reverse order that they appear on the line. Thus the characters D-C-B-A would be stored starting at location $1234. This is called flip ASCII.

0<1234.5679Z
Fills memory locations $1234 through $5678 with 0.

800<1234.5678M
Moves memory from $1234 to $5678 to location $800+.

800<1234.5678V
Compares each byte in the range of $1234 to $5678 to the corresponding byte starting at $800. Any bytes that do not match are reported.

"A"<1234.5678P
Search for the ASCII characters from $1234 to $5678.

25<1234.5678P
Search for the value $25 in memory in the range of $1234 to $5678.

"ABCD"<1234.5678P
Search for the pattern of ASCII characters ABCD in the memory in the range of $1234 to $5678.

25 7F 3E<1234.5678P
Search for the pattern of hex bytes 25 7F 3E in memory in the range of $1234 to $5678.

Number Conversions

<table>
<thead>
<tr>
<th>Sample Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234=</td>
<td>Convert $1234 to decimal.</td>
</tr>
<tr>
<td>=1234</td>
<td>Convert 1234 (decimal) to hex.</td>
</tr>
</tbody>
</table>

Program Execution and Register Display

<table>
<thead>
<tr>
<th>Sample Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control-E</td>
<td>Display contents of registers. These registers are set when a BRK is encountered, or they may be set by the user.</td>
</tr>
<tr>
<td>Control-N</td>
<td>Returns $e$, $m$, and $x$ bits to Native mode (0).</td>
</tr>
<tr>
<td>Control-R</td>
<td>Returns registers and flags to default Monitor configuration.</td>
</tr>
<tr>
<td>300G</td>
<td>Run a program in bank 0 (only). All registers are set using the stored values in the register display before the Go is executed. The routine should end with an RTS for control to return to the Monitor.</td>
</tr>
<tr>
<td>02/0300X</td>
<td>Execute a program in any bank. All registers are set using the stored values in the register display before the Go is executed. The routine should end with an RTL for control to return to the Monitor.</td>
</tr>
</tbody>
</table>
The following commands change the registers and flags in the stored register and flag display:

1234=A Set Accumulator to $1234.
1234=X Set X Register to $1234.
1234=Y Set Y Register to $1234.
1234=D Set Direct-Page Register to $1234.
12=B Set Data Bank to bank $12.
12=K Set Program Bank to bank $12.
1234=S Set Stack Pointer to $1234.
12=P Set Status Register to $12.
12=M Set machine state for the next Go or eXecute command to $12.
12=Q Set Quagmire state for next Go or eXecute command to $12. Quagmire bits are defined as follows:

Bit 7 = 1 High speed mode.
Bit 6 = 1 Stop Language Card, I/O shadowing.
Bit 5 = 1 Must always be zero.
Bit 4 = 1 Stop bank 1 hi-res shadowing.
Bit 3 = 1 Stop super hi-res shadowing.
Bit 2 = 1 Stop hi-res page 2 shadowing.
Bit 1 = 1 Stop hi-res page 1 shadowing.
Bit 0 = 1 Stop text page 1 shadowing.

0 = m Set m bit to 0 (or 1).
0 = e Set e bit to 0 (or 1).
0 = x Set x bit to 0 (or 1).
(See Control-N and Control-R.)

0 = L Change language card RAM to first bank.
1 = L Change language card RAM to second bank.
Appendix C

ProDOS 16 File Dump Utility

This program, Program C-1, demonstrates how a ProDOS 16 application opens, reads, and closes a file. In addition, the routine handles ProDOS errors by printing an error code and gives the user a chance to try again.

Support of text input and output on the Apple II GS is very limited. Apple really doesn’t want people to write programs that use the text display, so very little effort went into the text tools.

In particular, although there is a ReadString command in the text tools that will input a line of text from the keyboard, it does not support any editing functions such as the backspace or delete key, so is nearly useless. This sample program demonstrates how a very simple editing routine is built by handling the left- and right-arrow keys and Delete. ReadChar could have been used for getting a character from the keyboard, but there is a dilemma in how to handle control characters vs. normal text. You want to echo normal characters to the screen, but not control characters. ReadChar doesn’t distinguish between these in its echo parameter. In addition, a pause function is included that will start and stop the text output with a keypress.

For these reasons, the program deals directly with the keyboard register, $C000 for input, but it uses various text tools for output. Since the primary purpose of this program is to illustrate ProDOS 16 file techniques, I will leave it as an exercise for you to improve upon the input routine.

Program C-1. P16 File Dump

```
*******************************
1  * P16 FILE DUMP DEMO PROGRAM *
2  * MERLIN ASSEMBLER *
*******************************

593

5 MX %00 ; TELL MERLIN WE'RE IN 16 BITS
6 REL
7 DSK FDUMP.SYS16.L
8
9 LST OFF ; DON'T PRINT MACRO LISTING
10 USE UTIL.MACS ; USE MACRO LIBRARY
11 LST ON ; LISTING BACK "ON"
```

593
Appendix C

13 EXP OFF

14

= E100A8

15 PRODOS EQU $E100A8

16 KYBD EQU $00C000

17 STROBE EQU $00C010

18

008000: 4B 19 STARTUP PHK

008001: AB 20 PLB

008002: E2 21

008003: 30 22

008004: A9 23 SETRES SEP $30

008005: 5C 24 LDA #$5C ; JML (JMP WNG)

008006: 8F F8 03 00 25 STAL $3F8 ; CTRL-Y VECTOR

008007: C2 30 26 REP $30 ; 16-BIT MODE

008008: A9 A8 83 27 LDA #RESUME

008009: 8F F9 03 00 28 STAL $3F9 ; $3F9,3FA

00800A: A9 00 00 29 LDA #ARESUME

00800B: 8F FB 03 00 30 STAL $3FB ; $3FB,3FC

00800C: 30 31 PROMPT PushLong #MSSG1

00800D: 3C ToolCall $200C ; POINTER TO STRING TO PRINT

00800E: 8D 7E 83 32 LDA #$8D ; GET PATHNAME

00800F: STA INBUF

008010: 9C 7C 82 33 STZ LEN

008011: 00 INPUT PushLong #INBUF

008012: ToolCall $200C ; PRINT EXISTING PATHNAME

008013: 8D 7C 82 34 LDA LEN

008014: 88 35 CMP #$88 ; BACKSPACE?

008015: DO 34 BNE INPUT ; SPACE FOR RESULT

008016: AD 7C 82 36 EDIT LDA LEN

008017: BEQ EDIT ; GET LEN CHAR

008018: F0 C1 =8034 37 BEQ INPUT ; IGNORE

008019: 807F: AA 38 TAX ; KEEP OLD LEN IN X-REG

00801A: 3A 39 DEC ; SUBTRACT 1

00801B: 8D 7C 82 40 STA LEN ; LEN = LEN 1

594
008078: A9 00 00       65    LDA  #$00 ; PUT ZERO AT END
00807B: 9D 7E 83       66    STA  INBUF,X ; (DELETES OLD NTH CHAR).
00807E: 4C 34 80       68    JMP  INPUT ; GO GET SOME MORE
69
008081: 48              70    ECHO  PHA ; SAVE CHARACTER
008082: EE 7C 82       71    INC  LEN ; ADD 1 TO LENGTH
008085: AE 7C 82       72    LDX  LEN ; PUT IN XREG
008088: 68              73    PLA ; RETRIEVE CHARACTER
008089: 9D 7E 83       74    STA  INBUF,X ; ADD TO STRING
75 ; AUTO 'O' AT END! (HIGH BYTE)
00808C: 4C 34 80       76    JMP  INPUT ; GO GET SOME MORE
77
00808F: AE 7C 82       78    GOTPATH  LDX  LEN ; GET LENGTH OF INPUT
79
008092: E0 04 00       80    CHK1  CPX  #$04 ; 4 = LEN "QUIT"
008095: DO 13          81    BNE  FIX ; IT’S NOT "QUIT"
82
008097: AD 7F 83       83    CHK2  LDA  INBUF+1 ; 1ST & 2ND CHARS OF INPUT
00809A: CD 23 82       84    CMP  WORD ; "QUIT"?
00809D: D0 0B =80AA    85    BNE  FIX ; NOPE
00809E: AD 81 83       86    LDA  INBUF+3 ; 3RD & 4TH CHARS
0080A0: CD 25 82       87    CMP  WORD+2
0080A3: D0 03 =80AA    88    BNE  FIX ; NOPE
89
0080A6: 4C 79 81       90    JMP  QUIT ; STR$ = "QUIT"
91
0080A7: 4C 79 81       92    FIX  LDA  LEN ; 8-BIT MODE
93
0080AA: AD 7C 82       94    STA  INBUF ; CHANGE TO PRODOS STR$ ; 8-BIT MODE
95
0080AD: E2 00          96    REP  $30 ; BACK TO 16 BITS
0080AF: 8D 7E 83       97    CLRSCRN  PushWord #$008C ; HOME & CLEAR SCREEN
98
99
0080B2: C2 30          100  OPEN  JSLL  PRODOS ; OPEN COMMAND
0080C0: 10 00          101    DA  $10 ; OPEN CMD TABLE ; NO ERROR
0080C3: EE 81 00 00     102    ADRL  OPENBLK
0080CC: 90 03 =80D1     103    BCC  OPEN2 ; PRODOS ERROR MESSAGE
104
0080CE: 4C 85 81       105    JMP  ERROR ; PRODOS ERROR MESSAGE
106
0080D1: AD EE 81       107  OPEN2  LDA  OPENBLK ; GET REFERENCE NUMBER
108
0080D4: 8D F8 81       109    STA  READBLK ; STORE REF NUMBER
110
0080D7: 22 A8 00 E1    110    READ  JSLL  PRODOS ; READ COMMAND
111
0080DB: 12 00          112    DA  $12 ; READ CMD TABLE ; NO ERROR ...
0080DD: F8 81 00 00     113    ADRL  READBLK
0080E1: 90 08 =80EB     114    BCC  PRINT
115
0080E3: C9 4C 00       116    CMP  #$4C ; ERROR = END OF FILE?
117
0080E6: F0 5C =8144     118    BEQ  CLOSE ; YEP!
119

595
Appendix C

0080E8: 4C 85 81 117 JMP ERROR : PRODOS ERROR MSG
          118         ; 1ST CHAR OF BUFFER
0080EB: A2 01 00 119 PRINT LDX #$01         ; CHECK FOR A PAUSE KEY
          120         ; CLEAR HIGH BYTE
0080EE: AF 00 C0 00 121 PRLOOP LDAL KYBD ; KEYPRESS?
            AND #$00FF ; NOPE
0080F2: C9 80 00 122 CMP #$80 ; CLEAR KEYBOARD
0080F5: 90 14 810E 123 BCC PRCHAR ; WAIT FOR KEYPRESS
0080F8: 8F 10 C0 00 124 STAL STROBE
0080FA: AF 00 C0 00 125 PRINT LOX #$01 ; 1ST CHAR OF BUFFER
0080FC: AF 00 C0 00 126 PRWOP LOAL KYBD ; CHECK FOR A PAUSE KEY
0080F8: 29 FF 00 127 PAUSE AND #$00FF
008102: C9 80 00 128 CMP #$80 ; KEYPRESS?
008105: 90 14 8007 129 BCS READ ; NOPE
008108: 8F 10 C0 00 130 BCC PRWOP ; CLEAR KEYBOARD
00810A: SF 10 CO 00 131 STAL STROBE
00810E: BO 7E 82 132 PRCHAR LDA BUFFER,X
008110: AF 00 CO 00 133 AND #$00FF ; SAVE OUR XPOSITION
008114: DA 135 PHX ; SAVE THE CHAR FOR LATER
008115: 48 136 PHA ; PUSH THE CHARACTER
008116: 48 137 ToolCall $180C ; WriteChar
008118: 22 AS 00 E1 138 :2 PLA ; GET CHAR JUST PRINTED
00811A: EC 02 82 139 AND #$007F ; CLEAR HIGH BIT
00811C: 90 AE 80FE 140 CMP #$000D ; WAS IT A RETURN?
00811E: F0 AC 800E 141 BNE :2 ; NO, CONTINUE PRINTING
008122: 29 7F 00 142 PushWord #$008A ; LINE FEED
008124: 14 00 143 ToolCall $180C ; WriteChar
008126: C9 0D 00 144 PLA ; RETRIEVE X VALUE
008128: 90 0E 8139 145 INX ; NEXT CHAR IN BUFFER
00812A: EC 02 82 146 CPX NUMREAD ; ALL CHARS PRINTED YET?
00812C: 90 AE 80EE 147 BCC PRLOOP ; NOPE
00812E: F0 AC 80EE 148 BEQ PRLOOP ; THIS WILL BE THE LAST ONE
008130: BO 93 80D7 149 BCS READ ; GET ANOTHER BATCH
008132: 68 22 A8 00 E1 14A CLOSE JSR PRODOS ; CLOSE COMMAND
008134: 68 14 00 14B DA $14 ; SAME TABLE AS 'READ'
008136: F8 81 00 00 14C ADRL READBLK ; NO ERRORS
008138: 30 00 3153 14D BCC DONE ; PRODOS ERROR MSG
00813A: 04 C8 81 14E JMP ERROR ; END OF FILE MSG.
00813C: 161 DONE PushLong #MSSG3 ; SPACE FOR RESULT
00813E: 162 ToolCall $200C ; ECHO FLAG = NO ECHO
008140: 163 ToolCall $220C ; ReadChar
008142: 164 RDKEY2 PushWord #$0000 ; RETRIEVE CHARACTER
008144: 165 PushWord #$0000 ; SPACE FOR RESULT
008146: 166 ToolCall $220C ; ECHO FLAG = NO ECHO
008148: 167 ToolCall $200C ; ReadChar
00814A: 168 PLA ; RETRIEVE CHARACTER

596
ProDOS 16 File Dump Utility

008176: 4C 1A 80 169 JMP PROMPT ; TRY AGAIN
170
008179: 22 A8 00 E1 171 QUIT JSR PRODOS ; DO QUIT CALL
00817D: 29 60 172 DA $29 ; QUIT CALL COMMAND VALUE
00817F: E8 81 00 00 173 ADRL QUITBLK ; ADDRESS OF PARM TABLE
008183: 00 00 174 BRK $00 ; SHOULD NEVER GET HERE...
008185: 8D 75 82 175 ERROR STA ERRCODE ; ADDRESS OF M$G TEXT
008189: AD 75 82 176 ADRL QUITBLK ; WriteString
00818C: 00 00 00 00 177 PushLong #MSSG2 ; RETRIEVE ERROR CODE
00818E: 00 00 00 00 178 ToolCall $200C ; PUT IT ON STACK
008192: 75 82 179 PushLong #HEXSTR+1 ; STRING DATA BUFFER
008194: 00 00 00 00 180 ToolCall $220B ; MAX LENGTH OF OUTPUT STRING.
008198: F8 00 00 00 181 ToolCall $1COC ; Int2Hex
00819C: 00 00 00 00 182 ToolLong #HEXSTR ; POINTER TO STRING DATA
00819E: 00 00 00 00 183 ToolCall $120C ; WriteString
0081A2: 8F 00 00 00 184 ToolCall $1F00 ; ADDRESS OF M$G TEXT
0081A6: 7E 83 00 00 185 ToolLong #MSSG2A ; WriteString
0081AA: 00 00 00 00 186 ToolCall $200C ; SPACE FOR RESULT
0081AC: 00 00 00 00 187 ToolLong #HEXSTR ; ECHO FLAG = NO ECHO
0081AE: 00 00 00 00 188 ToolCall $1COC ; ReadChar
0081B2: 7E 83 00 00 189 ToolLong #MSSG2A ; WAIT FOR KEYPRESS
0081B6: 00 00 00 00 190 ToolCall $200C ; RETRIEVE CHAR.
0081B8: 68 191 JMP PROMPT ; TRY AGAIN IF ERROR
0081BE: 00 00 00 00 192 QUITBLK ADRL $0000 ; NO PATHNAME
0081C0: 00 00 00 00 193 ADRL $0000 ; STD. QUIT
0081C4: 00 00 00 00 194 OPENBLK DA $0000 ; FILE REFERENCE NUMBER
0081C8: 0F 00 00 00 195 ADRL INBUF ; POINTER TO PATHNAME
0081CC: 09 00 00 00 196 ADRL $0000 ; HANDLE TO PRODOS BUFFER
0081D0: 00 00 00 00 197 READBLK DA $0000 ; FILE REFERENCE NUMBER
0081D4: 00 00 00 00 198 ADRL BUFFER + 1 ; POINTER TO DATA BUFFER
0081D8: 00 00 00 00 199 ADRL 255 ; 255 CHARACTERS TO READ
0081DE: 00 00 00 00 200 NUMREAD ADRL $0000 ; NUMBER OF CHARS READ
0081E2: 00 00 00 00 201 JMP PROMPT ; TRY AGAIN IF ERROR
0081E8: 00 00 00 00 202 QUITBLK ADRL $0000 ; NO PATHNAME
0081EC: 00 00 00 00 203 DA $0000 ; STD. QUIT
0081EE: 00 00 00 00 204 OPENBLK DA $0000 ; FILE REFERENCE NUMBER
0081F0: 7E 83 00 00 00 205 ADRL INBUF ; POINTER TO PATHNAME
0081F4: 09 00 00 00 206 ADRL $0000 ; HANDLE TO PRODOS BUFFER
0081F8: 00 00 00 00 207 READBLK DA $0000 ; FILE REFERENCE NUMBER
0081FA: 7F 82 00 00 00 208 ADRL BUFFER + 1 ; POINTER TO DATA BUFFER
0081FE: FF 00 00 00 209 ADRL 255 ; 255 CHARACTERS TO READ
008202: 00 00 00 00 210 NUMREAD ADRL $0000 ; NUMBER OF CHARS READ
008206: 8C 211 NUMREAD ADRL $0000 ; NUMBER OF CHARS READ
008208: 64 B8 212 TR ON ; DON'T PRINT ALL THE CHARS
00820A: C4 CC 213 TR ON ; DON'T PRINT ALL THE CHARS
00820C: CC 00 00 00 214 TR ON ; DON'T PRINT ALL THE CHARS
00820E: CC 00 00 00 215 MSSG hex 8C ; HOME & CLEAR SCREEN
008210: E9 CC C5 C1 216 ASC "PLEASE ENTER PATHNAME: "
008213: AA CF D2 A0 217 ASC "(" OR ""
008216: A9 CF D2 A0 218 ASC "QUIT"
008219: A9 CF D2 A0 219 ASC ")" ":,8D,8A,00
00821C: 597
Appendix C

00822D: 8D 8A
00822F: D0 D2 CF C4
00823E: 8D 8A
008240: D0 D2 C5 D3
008259: 8D 8A
008258: DO D2 CS D3
008275: 00 00
008277: 04 80 80 BO
00827C: 00 00
00827E: 00 00 00
00837E: SD 00
008380: 00 00 00
0083A8: 48
0083A9: AB
0083AA: 18
0083AB: FB
0083AC: C2 30
0083AE: 4C 79 81
0083B1: 94

---End Merlin-16 assembly, 946 bytes, errors: 0
The Apple IIgs is such an involved and extended system that no one book can hope to cover more than just a portion of the information that is available on the machine. The following list is provided to help you in selecting additional references that will be helpful in programming the Apple IIgs.


Appendix D


———. *Elementary Assembly Language on the Apple IIGS and the 65816*. Glencoe, IL: Scott Foresman.


# Appendix E

## ASCII Character Chart

This chart shows some of the possible forms of a byte value in memory. The first three columns show the hex value and its decimal and binary equivalents. This can be handy when conversions are needed. The next three columns show the same value with the high bit set.

The ASCII character column shows the character assigned to the two values (high bit clear and set). For control characters, the standard ASCII abbreviation is also given, along with a notation for those characters that have some particular significance on the Apple IIgs.

Note that for control characters, the ^ symbol is used. Thus a Control-A would be indicated ^A.

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
<th>Dec</th>
<th>Binary</th>
<th>ASCII Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00</td>
<td>0</td>
<td>0000 0000</td>
<td>$80</td>
<td>128</td>
<td>1000 0000</td>
<td>@ NUL Null</td>
</tr>
<tr>
<td>$01</td>
<td>1</td>
<td>0000 0001</td>
<td>$81</td>
<td>129</td>
<td>1000 0001</td>
<td>A SOH</td>
</tr>
<tr>
<td>$02</td>
<td>2</td>
<td>0000 0010</td>
<td>$82</td>
<td>130</td>
<td>1000 0010</td>
<td>B STX</td>
</tr>
<tr>
<td>$03</td>
<td>3</td>
<td>0000 0011</td>
<td>$83</td>
<td>131</td>
<td>1000 0011</td>
<td>C ETX</td>
</tr>
<tr>
<td>$04</td>
<td>4</td>
<td>0000 0100</td>
<td>$84</td>
<td>132</td>
<td>1000 0100</td>
<td>D EOT</td>
</tr>
<tr>
<td>$05</td>
<td>5</td>
<td>0000 0101</td>
<td>$85</td>
<td>133</td>
<td>1000 0101</td>
<td>E ENQ</td>
</tr>
<tr>
<td>$06</td>
<td>6</td>
<td>0000 0110</td>
<td>$86</td>
<td>134</td>
<td>1000 0110</td>
<td>F ACK</td>
</tr>
<tr>
<td>$07</td>
<td>7</td>
<td>0000 0111</td>
<td>$87</td>
<td>135</td>
<td>1000 0111</td>
<td>G BEL Bell</td>
</tr>
<tr>
<td>$08</td>
<td>8</td>
<td>0000 1000</td>
<td>$88</td>
<td>136</td>
<td>1000 1000</td>
<td>H BS Backspace</td>
</tr>
<tr>
<td>$09</td>
<td>9</td>
<td>0000 1001</td>
<td>$89</td>
<td>137</td>
<td>1000 1001</td>
<td>I HT TAB</td>
</tr>
<tr>
<td>$0A</td>
<td>10</td>
<td>0000 1010</td>
<td>$8A</td>
<td>138</td>
<td>1000 1010</td>
<td>J LF Linefeed</td>
</tr>
<tr>
<td>$0B</td>
<td>11</td>
<td>0000 1011</td>
<td>$8B</td>
<td>139</td>
<td>1000 1011</td>
<td>K VT Up arrow</td>
</tr>
<tr>
<td>$0C</td>
<td>12</td>
<td>0000 1100</td>
<td>$8C</td>
<td>140</td>
<td>1000 1100</td>
<td>L FF Formfeed</td>
</tr>
<tr>
<td>$0D</td>
<td>13</td>
<td>0000 1101</td>
<td>$8D</td>
<td>141</td>
<td>1000 1101</td>
<td>M CR Return</td>
</tr>
<tr>
<td>$0E</td>
<td>14</td>
<td>0000 1110</td>
<td>$8E</td>
<td>142</td>
<td>1000 1110</td>
<td>N SO</td>
</tr>
<tr>
<td>$0F</td>
<td>15</td>
<td>0000 1111</td>
<td>$8F</td>
<td>143</td>
<td>1000 1111</td>
<td>O SI</td>
</tr>
<tr>
<td>$10</td>
<td>16</td>
<td>0001 0000</td>
<td>$90</td>
<td>144</td>
<td>1001 0000</td>
<td>P DLE</td>
</tr>
<tr>
<td>$11</td>
<td>17</td>
<td>0001 0001</td>
<td>$91</td>
<td>145</td>
<td>1001 0001</td>
<td>Q DC1 XON</td>
</tr>
<tr>
<td>$12</td>
<td>18</td>
<td>0001 0010</td>
<td>$92</td>
<td>146</td>
<td>1001 0010</td>
<td>R DC2</td>
</tr>
<tr>
<td>$13</td>
<td>19</td>
<td>0001 0011</td>
<td>$93</td>
<td>147</td>
<td>1001 0011</td>
<td>S DC3 XOFF</td>
</tr>
<tr>
<td>$14</td>
<td>20</td>
<td>0001 0100</td>
<td>$94</td>
<td>148</td>
<td>1001 0100</td>
<td>T DC4</td>
</tr>
</tbody>
</table>
### Appendix E

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
<th>Dec</th>
<th>Binary</th>
<th>ASCII Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>$15$</td>
<td>21</td>
<td>0001 0101</td>
<td>$95$</td>
<td>149</td>
<td>1001 0101</td>
<td>‘U’ NAK Right arrow</td>
</tr>
<tr>
<td>$16$</td>
<td>22</td>
<td>0001 0110</td>
<td>$96$</td>
<td>150</td>
<td>1001 0110</td>
<td>‘V’ SYN</td>
</tr>
<tr>
<td>$17$</td>
<td>23</td>
<td>0001 0111</td>
<td>$97$</td>
<td>151</td>
<td>1001 0111</td>
<td>‘W’ ETB</td>
</tr>
<tr>
<td>$18$</td>
<td>24</td>
<td>0001 1000</td>
<td>$98$</td>
<td>152</td>
<td>1001 1000</td>
<td>‘X’ CAN Cancel line</td>
</tr>
<tr>
<td>$19$</td>
<td>25</td>
<td>0001 1001</td>
<td>$99$</td>
<td>153</td>
<td>1001 1001</td>
<td>‘Y’ EM</td>
</tr>
<tr>
<td>$1A$</td>
<td>26</td>
<td>0001 1010</td>
<td>$9A$</td>
<td>154</td>
<td>1001 1010</td>
<td>‘Z’ SUB</td>
</tr>
<tr>
<td>$1B$</td>
<td>27</td>
<td>0001 1011</td>
<td>$9B$</td>
<td>155</td>
<td>1001 1011</td>
<td>↑ ↑</td>
</tr>
<tr>
<td>$1C$</td>
<td>28</td>
<td>0001 1100</td>
<td>$9C$</td>
<td>156</td>
<td>1001 1100</td>
<td>^ \ FS</td>
</tr>
<tr>
<td>$1D$</td>
<td>29</td>
<td>0001 1101</td>
<td>$9D$</td>
<td>157</td>
<td>1001 1101</td>
<td>↑ G5</td>
</tr>
<tr>
<td>$1E$</td>
<td>30</td>
<td>0001 1110</td>
<td>$9E$</td>
<td>158</td>
<td>1001 1110</td>
<td>~ RS</td>
</tr>
<tr>
<td>$1F$</td>
<td>31</td>
<td>0001 1111</td>
<td>$9F$</td>
<td>159</td>
<td>1001 1111</td>
<td>~ US</td>
</tr>
<tr>
<td>$20$</td>
<td>32</td>
<td>0010 0000</td>
<td>$A0$</td>
<td>160</td>
<td>1010 0000</td>
<td>space</td>
</tr>
<tr>
<td>$21$</td>
<td>33</td>
<td>0010 0001</td>
<td>$A1$</td>
<td>161</td>
<td>1010 0001</td>
<td>!</td>
</tr>
<tr>
<td>$22$</td>
<td>34</td>
<td>0010 0010</td>
<td>$A2$</td>
<td>162</td>
<td>1010 0010</td>
<td>&quot;</td>
</tr>
<tr>
<td>$23$</td>
<td>35</td>
<td>0010 0011</td>
<td>$A3$</td>
<td>163</td>
<td>1010 0011</td>
<td>#</td>
</tr>
<tr>
<td>$24$</td>
<td>36</td>
<td>0010 0100</td>
<td>$A4$</td>
<td>164</td>
<td>1010 0100</td>
<td>$</td>
</tr>
<tr>
<td>$25$</td>
<td>37</td>
<td>0010 0101</td>
<td>$A5$</td>
<td>165</td>
<td>1010 0101</td>
<td>%</td>
</tr>
<tr>
<td>$26$</td>
<td>38</td>
<td>0010 0110</td>
<td>$A6$</td>
<td>166</td>
<td>1010 0110</td>
<td>&amp;</td>
</tr>
<tr>
<td>$27$</td>
<td>39</td>
<td>0010 0111</td>
<td>$A7$</td>
<td>167</td>
<td>1010 0111</td>
<td>‘</td>
</tr>
<tr>
<td>$28$</td>
<td>40</td>
<td>0010 1000</td>
<td>$A8$</td>
<td>168</td>
<td>1010 1000</td>
<td>(</td>
</tr>
<tr>
<td>$29$</td>
<td>41</td>
<td>0010 1001</td>
<td>$A9$</td>
<td>169</td>
<td>1010 1001</td>
<td>)</td>
</tr>
<tr>
<td>$2A$</td>
<td>42</td>
<td>0010 1010</td>
<td>$AA$</td>
<td>170</td>
<td>1010 1010</td>
<td>*</td>
</tr>
<tr>
<td>$2B$</td>
<td>43</td>
<td>0010 1011</td>
<td>$AB$</td>
<td>171</td>
<td>1010 1011</td>
<td>+</td>
</tr>
<tr>
<td>$2C$</td>
<td>44</td>
<td>0010 1100</td>
<td>$AC$</td>
<td>172</td>
<td>1010 1100</td>
<td>,</td>
</tr>
<tr>
<td>$2D$</td>
<td>45</td>
<td>0010 1101</td>
<td>$AD$</td>
<td>173</td>
<td>1010 1101</td>
<td>-</td>
</tr>
<tr>
<td>$2E$</td>
<td>46</td>
<td>0010 1110</td>
<td>$AE$</td>
<td>174</td>
<td>1010 1110</td>
<td>.</td>
</tr>
<tr>
<td>$2F$</td>
<td>47</td>
<td>0010 1111</td>
<td>$AF$</td>
<td>175</td>
<td>1010 1111</td>
<td>/</td>
</tr>
<tr>
<td>$30$</td>
<td>48</td>
<td>0011 0000</td>
<td>$B0$</td>
<td>176</td>
<td>1011 0000</td>
<td>0</td>
</tr>
<tr>
<td>$31$</td>
<td>49</td>
<td>0011 0001</td>
<td>$B1$</td>
<td>177</td>
<td>1011 0001</td>
<td>1</td>
</tr>
<tr>
<td>$32$</td>
<td>50</td>
<td>0011 0010</td>
<td>$B2$</td>
<td>178</td>
<td>1011 0010</td>
<td>2</td>
</tr>
<tr>
<td>$33$</td>
<td>51</td>
<td>0011 0011</td>
<td>$B3$</td>
<td>179</td>
<td>1011 0011</td>
<td>3</td>
</tr>
<tr>
<td>$34$</td>
<td>52</td>
<td>0011 0100</td>
<td>$B4$</td>
<td>180</td>
<td>1011 0100</td>
<td>4</td>
</tr>
<tr>
<td>$35$</td>
<td>53</td>
<td>0011 0101</td>
<td>$B5$</td>
<td>181</td>
<td>1011 0101</td>
<td>5</td>
</tr>
<tr>
<td>$36$</td>
<td>54</td>
<td>0011 0110</td>
<td>$B6$</td>
<td>182</td>
<td>1011 0110</td>
<td>6</td>
</tr>
<tr>
<td>$37$</td>
<td>55</td>
<td>0011 0111</td>
<td>$B7$</td>
<td>183</td>
<td>1011 0111</td>
<td>7</td>
</tr>
<tr>
<td>$38$</td>
<td>56</td>
<td>0011 1000</td>
<td>$B8$</td>
<td>184</td>
<td>1011 1000</td>
<td>8</td>
</tr>
<tr>
<td>Hex</td>
<td>Dec</td>
<td>Binary</td>
<td>Hex</td>
<td>Dec</td>
<td>Binary</td>
<td>ASCII Character</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>----------</td>
<td>-----</td>
<td>-----</td>
<td>----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>$39</td>
<td>57</td>
<td>0011 1001</td>
<td>$B9</td>
<td>185</td>
<td>1011 1001</td>
<td>9</td>
</tr>
<tr>
<td>$3A</td>
<td>58</td>
<td>0011 1010</td>
<td>$BA</td>
<td>186</td>
<td>1011 1010</td>
<td>;</td>
</tr>
<tr>
<td>$3B</td>
<td>59</td>
<td>0011 1011</td>
<td>$BB</td>
<td>187</td>
<td>1011 1011</td>
<td>;</td>
</tr>
<tr>
<td>$3C</td>
<td>60</td>
<td>0011 1100</td>
<td>$BC</td>
<td>188</td>
<td>1011 1100</td>
<td>&lt;</td>
</tr>
<tr>
<td>$3D</td>
<td>61</td>
<td>0011 1101</td>
<td>$BD</td>
<td>189</td>
<td>1011 1101</td>
<td>=</td>
</tr>
<tr>
<td>$3E</td>
<td>62</td>
<td>0011 1110</td>
<td>$BE</td>
<td>190</td>
<td>1011 1110</td>
<td>&gt;</td>
</tr>
<tr>
<td>$3F</td>
<td>63</td>
<td>0011 1111</td>
<td>$BF</td>
<td>191</td>
<td>1011 1111</td>
<td>?</td>
</tr>
<tr>
<td>$40</td>
<td>64</td>
<td>0100 0000</td>
<td>$C0</td>
<td>192</td>
<td>1100 0000</td>
<td>@</td>
</tr>
<tr>
<td>$41</td>
<td>65</td>
<td>0100 0001</td>
<td>$C1</td>
<td>193</td>
<td>1100 0001</td>
<td>A</td>
</tr>
<tr>
<td>$42</td>
<td>66</td>
<td>0100 0010</td>
<td>$C2</td>
<td>194</td>
<td>1100 0010</td>
<td>B</td>
</tr>
<tr>
<td>$43</td>
<td>67</td>
<td>0100 0011</td>
<td>$C3</td>
<td>195</td>
<td>1100 0011</td>
<td>C</td>
</tr>
<tr>
<td>$44</td>
<td>68</td>
<td>0100 0100</td>
<td>$C4</td>
<td>196</td>
<td>1100 0100</td>
<td>D</td>
</tr>
<tr>
<td>$45</td>
<td>69</td>
<td>0100 0101</td>
<td>$C5</td>
<td>197</td>
<td>1100 0101</td>
<td>E</td>
</tr>
<tr>
<td>$46</td>
<td>70</td>
<td>0100 0110</td>
<td>$C6</td>
<td>198</td>
<td>1100 0110</td>
<td>F</td>
</tr>
<tr>
<td>$47</td>
<td>71</td>
<td>0100 0111</td>
<td>$C7</td>
<td>199</td>
<td>1100 0111</td>
<td>G</td>
</tr>
<tr>
<td>$48</td>
<td>72</td>
<td>0100 1000</td>
<td>$C8</td>
<td>200</td>
<td>1100 1000</td>
<td>H</td>
</tr>
<tr>
<td>$49</td>
<td>73</td>
<td>0100 1001</td>
<td>$C9</td>
<td>201</td>
<td>1100 1001</td>
<td>I</td>
</tr>
<tr>
<td>$4A</td>
<td>74</td>
<td>0100 1010</td>
<td>$CA</td>
<td>202</td>
<td>1100 1010</td>
<td>J</td>
</tr>
<tr>
<td>$4B</td>
<td>75</td>
<td>0100 1011</td>
<td>$CB</td>
<td>203</td>
<td>1100 1011</td>
<td>K</td>
</tr>
<tr>
<td>$4C</td>
<td>76</td>
<td>0100 1100</td>
<td>$CC</td>
<td>204</td>
<td>1100 1100</td>
<td>L</td>
</tr>
<tr>
<td>$4D</td>
<td>77</td>
<td>0100 1101</td>
<td>$CD</td>
<td>205</td>
<td>1100 1101</td>
<td>M</td>
</tr>
<tr>
<td>$4E</td>
<td>78</td>
<td>0100 1110</td>
<td>$CE</td>
<td>206</td>
<td>1100 1110</td>
<td>N</td>
</tr>
<tr>
<td>$4F</td>
<td>79</td>
<td>0100 1111</td>
<td>$CF</td>
<td>207</td>
<td>1100 1111</td>
<td>O</td>
</tr>
<tr>
<td>$50</td>
<td>80</td>
<td>0101 0000</td>
<td>$D0</td>
<td>208</td>
<td>1101 0000</td>
<td>P</td>
</tr>
<tr>
<td>$51</td>
<td>81</td>
<td>0101 0001</td>
<td>$D1</td>
<td>209</td>
<td>1101 0001</td>
<td>Q</td>
</tr>
<tr>
<td>$52</td>
<td>82</td>
<td>0101 0010</td>
<td>$D2</td>
<td>210</td>
<td>1101 0010</td>
<td>R</td>
</tr>
<tr>
<td>$53</td>
<td>83</td>
<td>0101 0011</td>
<td>$D3</td>
<td>211</td>
<td>1101 0011</td>
<td>S</td>
</tr>
<tr>
<td>$54</td>
<td>84</td>
<td>0101 0100</td>
<td>$D4</td>
<td>212</td>
<td>1101 0100</td>
<td>T</td>
</tr>
<tr>
<td>$55</td>
<td>85</td>
<td>0101 0101</td>
<td>$D5</td>
<td>213</td>
<td>1101 0101</td>
<td>U</td>
</tr>
<tr>
<td>$56</td>
<td>86</td>
<td>0101 0110</td>
<td>$D6</td>
<td>214</td>
<td>1101 0110</td>
<td>V</td>
</tr>
<tr>
<td>$57</td>
<td>87</td>
<td>0101 0111</td>
<td>$D7</td>
<td>215</td>
<td>1101 0111</td>
<td>W</td>
</tr>
<tr>
<td>$58</td>
<td>88</td>
<td>0101 1000</td>
<td>$D8</td>
<td>216</td>
<td>1101 1000</td>
<td>X</td>
</tr>
<tr>
<td>$59</td>
<td>89</td>
<td>0101 1001</td>
<td>$D9</td>
<td>217</td>
<td>1101 1001</td>
<td>Y</td>
</tr>
<tr>
<td>$5A</td>
<td>90</td>
<td>0101 1010</td>
<td>$DA</td>
<td>218</td>
<td>1101 1010</td>
<td>Z</td>
</tr>
<tr>
<td>$5B</td>
<td>91</td>
<td>0101 1011</td>
<td>$DB</td>
<td>219</td>
<td>1101 1011</td>
<td>[</td>
</tr>
<tr>
<td>$5C</td>
<td>92</td>
<td>0101 1100</td>
<td>$DC</td>
<td>220</td>
<td>1101 1100</td>
<td>\</td>
</tr>
<tr>
<td>$5D</td>
<td>93</td>
<td>0101 1101</td>
<td>$DD</td>
<td>221</td>
<td>1101 1101</td>
<td>]</td>
</tr>
<tr>
<td>$5E</td>
<td>94</td>
<td>0101 1110</td>
<td>$DE</td>
<td>222</td>
<td>1101 1110</td>
<td>^</td>
</tr>
<tr>
<td>$5F</td>
<td>95</td>
<td>0101 1111</td>
<td>$DF</td>
<td>223</td>
<td>1101 1111</td>
<td>_</td>
</tr>
<tr>
<td>$60</td>
<td>96</td>
<td>0110 0000</td>
<td>$E0</td>
<td>224</td>
<td>1110 0000</td>
<td>'</td>
</tr>
<tr>
<td>$61</td>
<td>97</td>
<td>0110 0001</td>
<td>$E1</td>
<td>225</td>
<td>1110 0001</td>
<td>a</td>
</tr>
<tr>
<td>Hex</td>
<td>Dec</td>
<td>Binary</td>
<td>Hex</td>
<td>Dec</td>
<td>Binary</td>
<td>ASCII Character</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>--------</td>
<td>-----</td>
<td>-----</td>
<td>--------</td>
<td>-----------------</td>
</tr>
<tr>
<td>$62</td>
<td>98</td>
<td>0110 0010</td>
<td>$E2</td>
<td>226</td>
<td>1110 0010</td>
<td>b</td>
</tr>
<tr>
<td>$63</td>
<td>99</td>
<td>0110 0011</td>
<td>$E3</td>
<td>227</td>
<td>1110 0011</td>
<td>c</td>
</tr>
<tr>
<td>$64</td>
<td>100</td>
<td>0110 0100</td>
<td>$E4</td>
<td>228</td>
<td>1110 0100</td>
<td>d</td>
</tr>
<tr>
<td>$65</td>
<td>101</td>
<td>0110 0101</td>
<td>$E5</td>
<td>229</td>
<td>1110 0101</td>
<td>e</td>
</tr>
<tr>
<td>$66</td>
<td>102</td>
<td>0110 0110</td>
<td>$E6</td>
<td>230</td>
<td>1110 0110</td>
<td>f</td>
</tr>
<tr>
<td>$67</td>
<td>103</td>
<td>0110 0111</td>
<td>$E7</td>
<td>231</td>
<td>1110 0111</td>
<td>g</td>
</tr>
<tr>
<td>$68</td>
<td>104</td>
<td>0110 1000</td>
<td>$E8</td>
<td>232</td>
<td>1110 1000</td>
<td>h</td>
</tr>
<tr>
<td>$69</td>
<td>105</td>
<td>0110 1001</td>
<td>$E9</td>
<td>233</td>
<td>1110 1001</td>
<td>i</td>
</tr>
<tr>
<td>$6A</td>
<td>106</td>
<td>0110 1010</td>
<td>$EA</td>
<td>234</td>
<td>1110 1010</td>
<td>j</td>
</tr>
<tr>
<td>$6B</td>
<td>107</td>
<td>0110 1011</td>
<td>$EB</td>
<td>235</td>
<td>1110 1011</td>
<td>k</td>
</tr>
<tr>
<td>$6C</td>
<td>108</td>
<td>0110 1100</td>
<td>$EC</td>
<td>236</td>
<td>1110 1100</td>
<td>l</td>
</tr>
<tr>
<td>$6D</td>
<td>109</td>
<td>0110 1101</td>
<td>$ED</td>
<td>237</td>
<td>1110 1101</td>
<td>m</td>
</tr>
<tr>
<td>$6E</td>
<td>110</td>
<td>0110 1110</td>
<td>$EE</td>
<td>238</td>
<td>1110 1110</td>
<td>n</td>
</tr>
<tr>
<td>$6F</td>
<td>111</td>
<td>0110 1111</td>
<td>$EF</td>
<td>239</td>
<td>1110 1111</td>
<td>o</td>
</tr>
<tr>
<td>$70</td>
<td>112</td>
<td>0111 0000</td>
<td>$F0</td>
<td>240</td>
<td>1111 0000</td>
<td>p</td>
</tr>
<tr>
<td>$71</td>
<td>113</td>
<td>0111 0001</td>
<td>$F1</td>
<td>241</td>
<td>1111 0001</td>
<td>q</td>
</tr>
<tr>
<td>$72</td>
<td>114</td>
<td>0111 0010</td>
<td>$F2</td>
<td>242</td>
<td>1111 0010</td>
<td>r</td>
</tr>
<tr>
<td>$73</td>
<td>115</td>
<td>0111 0011</td>
<td>$F3</td>
<td>243</td>
<td>1111 0011</td>
<td>s</td>
</tr>
<tr>
<td>$74</td>
<td>116</td>
<td>0111 0100</td>
<td>$F4</td>
<td>244</td>
<td>1111 0100</td>
<td>t</td>
</tr>
<tr>
<td>$75</td>
<td>117</td>
<td>0111 0101</td>
<td>$F5</td>
<td>245</td>
<td>1111 0101</td>
<td>u</td>
</tr>
<tr>
<td>$76</td>
<td>118</td>
<td>0111 0110</td>
<td>$F6</td>
<td>246</td>
<td>1111 0110</td>
<td>v</td>
</tr>
<tr>
<td>$77</td>
<td>119</td>
<td>0111 0111</td>
<td>$F7</td>
<td>247</td>
<td>1111 0111</td>
<td>w</td>
</tr>
<tr>
<td>$78</td>
<td>120</td>
<td>0111 1000</td>
<td>$F8</td>
<td>248</td>
<td>1111 1000</td>
<td>x</td>
</tr>
<tr>
<td>$79</td>
<td>121</td>
<td>0111 1001</td>
<td>$F9</td>
<td>249</td>
<td>1111 1001</td>
<td>y</td>
</tr>
<tr>
<td>$7A</td>
<td>122</td>
<td>0111 1010</td>
<td>$FA</td>
<td>250</td>
<td>1111 1010</td>
<td>z</td>
</tr>
<tr>
<td>$7B</td>
<td>123</td>
<td>0111 1011</td>
<td>$FB</td>
<td>251</td>
<td>1111 1011</td>
<td>{</td>
</tr>
<tr>
<td>$7C</td>
<td>124</td>
<td>0111 1100</td>
<td>$FC</td>
<td>252</td>
<td>1111 1100</td>
<td></td>
</tr>
<tr>
<td>$7D</td>
<td>125</td>
<td>0111 1101</td>
<td>$FD</td>
<td>253</td>
<td>1111 1101</td>
<td>}</td>
</tr>
<tr>
<td>$7E</td>
<td>126</td>
<td>0111 1110</td>
<td>$FE</td>
<td>254</td>
<td>1111 1110</td>
<td>~</td>
</tr>
<tr>
<td>$7F</td>
<td>127</td>
<td>0111 1111</td>
<td>$FF</td>
<td>255</td>
<td>1111 1111</td>
<td>Delete</td>
</tr>
</tbody>
</table>
absolute addressing mode 180, 474
absolute indexed, X addressing mode 475
absolute indexed, Y addressing mode 475
absolute long addressing mode 474
absolute long indexed, X addressing mode 476
absolute long indirect addressing mode 477
accumulator 44-47, 48, 116, 125, 131, 132, 164, 274-75
changing effective size of 132
ADC (ADd with Carry) instruction 145-49, 151, 478-79
addition 143-50
addressing modes 179-204, 474-77
  absolute 180, 474
  absolute indexed, X 475
  absolute indexed, Y 475
  absolute long indexed, X 476
  absolute long indirect 477
direct page 180, 475
direct page indexed indirect, X 476
direct page indexed, X 476
direct page indexed, Y 476
direct page indirect 475
direct page indirect long 475
direct page indirect long indexed, Y 477
immediate 179-80, 475
relative stack 211-12
relative stack indirect indexed 212-14
stack relative 477
stack relative indexed, Y 477
ADRL Merlin pseudo-op 275
& BASIC statement 236
AND instruction 163-66
AND logical operator 162-68, 480-82
Apple II computers, development history of 290-92
Apple II GS computer, enhanced features of 271
Apple II GS mini-assembler 39-55. See also mini-assembler
Apple II GS Technical Reference x
Apple Macintosh computer 114
Applesoft BASIC x
Applesoft BASIC, ML and 3-21
Applesoft BASIC, Monitor and 24-25
APW (Apple Programmer’s Workshop) assembler x, 59, 81-98, 137-38, 251
ASSEMBLE command 88-89
assembler directives 87-88
assembling a program and 88-90
code segments 87
comment field 86-87
editor 83-87
EQU directive 92-95
filenames and 92
how different from Merlin 81
KEEP directive 89-90, 91-92
label field 86
labels and 92-96
linking and 82, 282-83
loading 83
long addresses and 87-88, 97
macros and 325, 330-33
object file 82
opcode field 86
operand field 86
ORG directive 90-91
restrictions on use of 81
shell 82, 83
theory of operation 82
A register. See accumulator
ASCII codes 54, 118-20, 185, 186
ASC instruction (Merlin) 190-91
ASL (Arithmetic Shift Left) instruction 159-60, 483-84
ASM command (Merlin) 66, 67
ASSEMBLE command (APW) 88-89
assembler directives, APW 87-88
assemblers, other 59
assembling a program, APW and 88-90
assembling to disk, Merlin and 74-75
assembly language 6
  how different from ML 32-34
AUX field 338, 340-41
bank of memory 43, 218-22, 273, 274
banks, parallel
  in Ile and IIc 302-4
  in IIGS 304-8
bank-switched RAM and ROM, location of 302
BASIC. See also Applesoft BASIC
calls, list of available from 13
eight-bit mode and 135
error trapping in 8-10
FOR-NEXT loop equivalent of 101-21
how related to ML 6-7
interpreter 6-7, 229-31
limitations of 3-4
memory organization of 298-300
ML programs, adding to 225-44
ML routines, calling from 11-21
ML routines, POKEing from 23-34
Monitor, entering from 25
ProDOS and 247
ProDOS 16 and 271, 274
program, organization of in memory 227-31
ROM, Applesoft, location of 14
tokens 228
when preferable over ML 225

BCC (Branch on Carry Clear) instruction 126, 160, 161, 187, 485
BCS (Branch on Carry Set) instruction 126, 161, 187, 486
BEQ (Branch if Equal) instruction 108-10, 111, 125, 126, 160, 187, 487
binary numbers 102-4
BIT instruction 166-68, 488-90
BLOAD BASIC statement 68-69
BLOAD BINARY BASIC statement 41-42
BMI (Branch on Minus) instruction 154, 155, 156, 167, 491-92
BNE (Branch if Not Equal) instruction 107-8, 117, 125, 126, 187, 493
Boolean logic 161-62
borrowing, subtraction and 150
BPL (Branch on Plus) instruction 154, 155, 160, 167, 494
BRA (Branch Always) instruction 110, 495
branch-on-carry instructions 126
B register. See data bank register
BRK instruction 53, 54, 495-96
BRL (Branch always Long) instruction 110, 497
BRUN BASIC statement 68
BSAVE BINARY BASIC statement 41
BVC (Branch on overflow Clear) instruction 167, 498
BVS (Branch on overflow Set) instruction 167, 499
CALL BASIC statement 11-21, 23, 24, 52, 117, 225
Monitor equivalent of 31-32
calls, list of available from Applesoft BASIC 13
carry flag 126, 131, 145, 159
changing memory locations with the Monitor 31
checksum 173-74
CHK pseudo-op (Merlin) 173
CHGET BASIC ROM routine 232, 234
CLC (Clear Carry) instruction 133, 145, 500
CLD (Clear Decimal mode) instruction 501
clear screen 195-99
80-column 220-22
indexed addressing and 197-99
CLI (Clear Interrupt mask) instruction 502
“Clock Reading” (program) 361-64
CLV (Clear overflow flag) instruction 503
CMP (Compare) instruction 125-38, 504-5
types of 125
code segments, APW 87
color, QuickDraw and 376-77
colors, graphics modes and 367-68
command processing 201-2
“Command Processor Example” (program) 202-3
comment field
APW 86-87
Merlin 65
comments, in assembly language programs 62

comparisons 125-39
computer memory 5
constants, Merlin and 73-74
control characters 119
control key 119-20
COP (CO-Processor enable) instruction 506
COUT (Character OUTPUT) Monitor routine 116-18, 128, 131, 181, 291
CPX (Compare to X register) instruction 507
CPY (Compare to Y register) instruction 508
cursor movement 15-16
data bank register 218, 219-20
data storage 207-22
data storage, assembler 185
data structures 3
data word 114
document, Merlin 65
decimal numbers 161-62
decrementing 105-7
decrementing multiple bytes 114-15
designator, menu item 419-21
dex instruction 105-6, 510
dey instruction 105-6, 511
DFB (Define Byte) pseudo-op 250
direct page 290-91
direct page addressing mode 180
direct page indexed indirect, X addressing mode 476
direct page indexed, X addressing mode 476
direct page indexed, Y addressing mode 476
direct page indirect long indexed, Y addressing mode 477
direct page register 291
Disable item Menu Manager routine 415, 461
disk access, ProDOS and 247
dithering 367
double hi-res graphics 367
DrawCString QuickDraw routine 379
drawing program 447-70
DrawMenuBar Menu Manager routine 415
DrawString QuickDraw routine 379
dR register 216-18. See also direct page register
dS (Define Storage) pseudo-op, Merlin 262
dSK Merlin directive 74-75
EMStartup call 389
emulation bit 131-37
Enable item Menu Manager routine 415, 461
enhanced quit option, ProDOS 8 254-55
ENT (ENTRY point) Merlin directive 281
eOM (End Of Macro) Merlin directive 322
eO instruction 169-71, 173, 512-14
EQU directive 185
APW 92-95
Merlin 70-74
error codes
ProDOS MLI 263-64
Tool Locator 319-20
error messages, QuickDraw 385
error trapping, ProDOS system files 258-59
“EVENT.DISPLAY” (program) 437-43
“EVENT.SEGMENT” (program) 437

606
event code bit modifiers 387
event codes 387
event-driven programs 389–92
Event Manager 385–99, 413
“Event Manager Demo” (program) 393–99
event mask 388
event queue 386
event record 386
EXT (EXTemal) Merlin directive 281
fast memory 305
file handling, ProDOS 8 MLI and 257–68
filenames, APW and 92
file read command, ProDOS 262–63
file types, ProDOS 256
FramErect QuickDraw routine 373
GETCORIGIN QuickDraw routine 443
GETDP (GET Direct Page) tool call 375
GETLN2 Monitor command 260
GetNextEvent Event Manager routine 391, 414
GetPortRect QuickDraw routine 443
global page, ProDOS 300
GOSUB BASIC statement 23, 24
ML equivalent of 34
graphics information, storing 368–69
handle 337, 341–42, 344
hardware connections
  memory locations of 16–18
  reading 128–29
hexadecimal numbers 25–27, 30
  converting to decimal 27–28
  reason for using 29–30
HEX instruction (Merlin) 185
HIDEWINDOW Window Manager command 461
high bit, ASCII codes and 119
hi-res graphics 367
hot spot, cursor 464–65
immediate addressing mode 179–80
implicit addressing 180
INC instruction 105–6, 144, 515
incrementing 105–7
incrementing multiple bytes 111–14
indexed addressing 181
indexed indirect addressing 183–85, 201–2
indirect addressing 181–82, 193–94
indirect long addressing 182–83
input buffer 237
input from Monitor 262
input, monitoring 385–86
InsertMenu Merlin directive 415
instructions, 65816 473–587
interpreter, BASIC 6–7, 229–31
interrupt, maskable 298
INX instruction 105–6, 275, 516
INY instruction 105–6, 517
I/O ROM, location of 300–301
JML (JuMp Long) instruction 116, 518
JMP instruction 235, 518
JSL (Jump Subroutine Long) instruction 51, 75, 97, 210, 519–20
JSR instruction 34, 50, 65, 75, 207, 519–20
  maintaining with stack 207
KEEP directive (APW) 89–90, 91–92
kernel 248
keyboard 275
  address of 128
  buffering 129
  reading data from 127–31
K register. See program bank register
label field
  APW 86
  Merlin 64
labels
  APW and 92–96
  Merlin 8/16 and 64–65, 70–76
languages, programming 4
Launcher program selector 60, 83
LDA (LoaD Accumulator) instruction 44, 45, 47, 50, 73–74, 75, 97, 129, 145, 195, 275
LDX instruction 47, 522
LDY instruction 47, 195, 523
library routines x–xi
LineTo QuickDraw routine 372
linker, APW 90
linking
  APW and 82, 282–83
  Merlin and 280–82
LIST BASIC statement, Monitor equivalent of 32–34, 188
L Monitor command 32–34
loader file, APW 82
loading an ML program 41–42
logical and shift operators 159–75
logical operators 161–71
looping
  BEQ and 108–10
  BNE and 107–8
LSR (Logical Shift Right) instruction 159, 161, 524–25
MAC (MACro definition) Merlin directive 322
macrop, assembly-language 322–33
  APW and 325, 330–33
  library, building 325–30
  Merlin and 325–30
main field 338
mask 165–66
math 143–56
memory
  addressable 5–6
  compacting 343
  examining with Monitor 24–25
  graphics and 369
  map 7–8
organization, Apple II GS 289–308
purging 343–44
requesting 342–43
special 341
storing data in 42–43
unmanaged 342
Memory Manager system tool 273–74, 337–60
menu bar, creating 421–22
menu definition 416–19
Menu Manager call list 416
Menu Manager tool 414, 415–28
Menu program example 422–36
Merlin 8/16 assembler x, 59–81, 250
ADRL pseudo-op 275
ASC instruction 190–91
ASM command 66, 67
assembling to disk and 74–75
assembling with 66–77
CHK pseudo-op 173
comment field 65
constants and 73–74
DS (Define Storage) pseudo-op 262
DSK directive 74–75
how different from APW 81
label field 64
labels and 64–65, 70–76
linking and 280–82
long addresses and 75–76
macros and 325–30
main menu 61
operand field 65
saving programs from 68
symbol table 67
mini-assembler, entering 39
miscellaneous tool set 359–60
ML
instructions 6
programs, adding to Applesoft BASIC 225–44
routines, calling from BASIC 11–21. See also ROM routines
routines, POKEing from BASIC 23–34
why learn 3–4
MLI (machine language interface), ProDOS 249, 257
Monitor 24–25, 26, 27, 39, 135–37, 187, 589–91
entering from BASIC 25
ProDOS 16 and 217, 274
MoveTo QuickDraw routine 372
moving data between registers 49–50
moving data in memory 48–49
"Moving the Cursor" (program) 15
multiple programs sharing memory 216–18
multiplication 159, 160
MVN (block MoVe Next) instruction 526–27
MVP (block MoVe Previous) instruction 526–27
MX Merlin directive 274
NewHandle Memory Manager command 371
NEWRGN call 451
nonmaskable interrupt 297–98
NOP (No oPeration) instruction 112, 528
object file 67–68
APW 82
opcode 64
opcode field, APW 86
OPEN command, ProDOS 262
operand field
APW 86
Merlin 65
operating system 247
OR
exclusive 168, 169–71
inclusive 168
logical 168–71
ORA (OR Accumulator) instruction 168–69, 529–30
ORG assembler directive (Merlin) 69–70, 185
ORG directive (APW) 90–91
overflow flag 166–67
page three 292
PAINTPIXELS QuickDraw routine 448
palette 370
Pascal computer language 4
“Passing Variables” (program) 234–36
“Passing Variables to an Applesoft BASIC Pro-
gram” (program) 239–40
pathname 254, 258, 260
PEA (Push Effective Address) instruction 214–15, 531–32
PEEK BASIC statement 8–10
Monitor equivalent of 30–31
“P8 File Dump Demo” (program) 258–68
PEI (Push Effective Indirect address) instruction
214–15, 533
pen 372
PER (Push Effective Relative address) instruction
214–15, 534–35
PHA (PushH Accumulator) instruction 208, 536
PHB (PushH data Bank register) instruction 220, 537
PHD (PushH Direct page register) instruction 218,
538
PHK (PushH program bank register) instruction 219,
539
PHP (PushH Processor status) instruction 540–41
PHX (PushH X) instruction 208, 542
PHY (PushH Y) instruction 208, 543–44
PLA (Pull Accumulator) instruction 208, 544
PLB (Pull data Bank register) instruction 220, 545
PLD (Pull Direct page register) instruction 217,
546
PLP (Pull Processor status) instruction 547
PLX (Pull X) instruction 208, 548
PLY (Pull Y) instruction 208, 549
POKE BASIC statement 10–11, 23–34
Monitor equivalent of 31
passing variables to ML with 231–32
PORTLOCINFO data structure 448
PPToPort (Paint Pixels To Port) QuickDraw routine
448–50
PQUIT ProDOS 16 routine 271
procedures in ML 111–14
Processor Status Register 101–2. See also status
register
ProDOS x, 41, 44, 50, 60, 216, 247–86
Applesoft BASIC and 247
calling from ML 249
compatibility of 247
file types 82
ProDOS 8 247–68
loading directly 339
“ProDOS 8 Launcher Demo” (program) 254–55
‘ProDOS 8 Tool Locator Demo’ (program) 315–17
Professional Disk Operating System. See ProDOS
program bank register 43, 218-19
program control 207-22
program counter 43
program documentation 85
Programming the 65816 ix
pseudo-op 185
pun 30
QuickDraw (QuickDraw) tool set 370-99, 403-5
calls, list of 383-85
starting 374-75
Quit command, ProDOS 16 273, 302
rectangle 373
registers 42-51
D 216-18
Data Bank 218, 219-20
Direct Page 291
moving data between 49-50
PHB (Push data Bank) instruction 220
PHD (Push Direct page) instruction 218
PHK (Push program bank) instruction 219
PLB (Pull data Bank) instruction 220
PLD (Pull Direct page ) instruction 217
Program Bank 43, 218-19
S 209-11, 292
Shadow 305-6
stack and 208
Status 101-2, 104-5, 126, 131-37, 167
relative addressing 180
relative stack addressing mode 211-12
relative stack indirect indexed addressing mode
212-14
relocatable files 272
relocating ML program in memory 70, 90-91, 274
REP (ReSet Processor status bits) instruction 133, 550
RESET key 295
RETURN BASIC statement, ML equivalent of 24, 34-35
ROL (ROtate Left) instruction 161, 551
ROM routines 14-16, 128, 231-44
naming with variables in BASIC 15-16
ROR (ROtate Right) instruction 161, 552
RTI (ReTurn from Interrupt) instruction 553
RTL (ReTurn from subroutine Long) instruction 51, 554
RTS (ReTurn from Subroutine) instruction 33, 51, 67, 207, 554
saving an ML program 41
SBC (Subtract with Carry) instruction 150, 151, 555-56
SCB (Scan line Control Byte) 370-71
screen, location of in memory 46
“Screen Color Demo” (program) 18-21
color manipulation 185-203
SEC (SEt Carry) instruction 133, 150, 557
SED (SEt Decimal mode) instruction 558
SEI (SEt Interrupt disable) instruction 298, 559
SEP (SEt Processor status bits) instruction 133, 560
SetBackColor QuickDraw routine 379
SetColorEntry QuickDraw routine 377
SETCOLOR QuickDraw routine 376
SETCURSOR QuickDraw routine 464
SetForeColor QuickDraw routine 379
SETRECT call 451
SetSolidPenPat QuickDraw routine 372
shadowing 305
Shadow register 305-6
“Shell” (program) 422-36
shift operators 159-61
short address 43
sign bit 151-55
sign flag, in status register 154
“Simple P8 System File” (program) 252-54
“Simple P16 System File” (program) 277-78
“Simple QuickDraw Demo” (program) 379-83
“SIMPLE.SKETCH” (program) 451-57
65816 microprocessor 5, 23, 42
6502 microprocessor 5
“Slide Show” (program) 347-59
softswitches 17-18, 302
sound ix, 231-34
“Sound Routine” example programs 230-33
source file 67-68
S register 209-11, 292
stack 207-18, 291-92
keeping track of 208-9
pushing addresses to 214-15
“Stack Indirect Indexed Example” (program)
213-14
stack pointer. See S register
stack pointer, setting 211
stack relative addressing mode 477
stack relative indexed,Y addressing mode 477
STA (STore Accumulator) instruction 45-45, 47, 50, 75, 97, 561
status register 101-2, 104-5, 126, 131-37, 167
STILLDOWN Event Manager routine 465
STP (STop Processor) instruction 562
StringWidth QuickDraw routine 391
strobe 128
STX instruction 563
STY instruction 195, 564
subroutines, ML 34-35
subtraction 150-54
super hi-res graphics ix, 367-79
location of 306
super hi-res screen, defining memory equivalent to
448-50
“Switching to Hi-Res” (program) 17
symbol table, Merlin 67
SYSTEM.SETUP file 272
.SYSTEM files 248–49
system I/O 127
system loader 271–72
system startup, ProDOS 16 and 271
Taskmaster Window Manager routine 414
TAX (Transfer Accumulator to X) instruction 49, 566–67
TAY (Transfer Accumulator to Y) instruction 49, 568
TCD (Transfer aCCumulator to Direct-page register) instruction 569
TCS (Transfer aCCumulator to Stack) instruction 211, 570
TDC (Transfer Direct page to aCCumulator) instruction 423, 571
text, printing in super hi-res with QuickDraw 378–79
text display, location in memory 298
toggling 63
TOOL.SETUP file 272
Toolbox, Apple IIGS 311–33
Tool call ix
Tool Locator tool set 313–17
command list 314
ProDOS 16 and 320–21
tools
bugs in 338
calling 311
calling from ProDOS 8 317–19
list of 312
Memory Manager 344–47
TRB (Test and Reset Bits) instruction 166, 572
TR ON (TRuncate ON) Merlin directive 320
TSB (Test and Set memory Bits) 573
TSC (Transfer Stack to aCCumulator) instruction 211, 574
TSX (Transfer Stack to X) instruction 211, 575–76
TXA (Transfer X to Accumulator) instruction 49, 577–78
TXS (Transfer X to Stack) instruction 211, 579
TXTPTR BASIC ROM routine 232, 234, 237
TXY (Transfer X to Y) instruction 49, 580
TYA (Transfer Y to Accumulator) instruction 49, 581
TYP directive 250
type field 338
TYX (Transfer Y to X) instruction 49, 582
User ID 337
"Using Meaningful Variable Names" (program) 16 variables
passing from BASIC to ML 231–38
passing from ML to BASIC 238–44
vector 236, 295
video circuitry 305
VTAB BASIC statement, assembler equivalent of 191–93
VTAB Monitor routine 191–92
WAI (WAit for Interrupt) instruction 583
WDM instruction 584
WFRAME parameter 407, 408–13
window
monitoring events in 437–43
moving 413
windows ix, 403–15
text files and 447–48
window data, recording elsewhere in memory 447–48
window definitions 406–7
Window Manager 405–15
window parameter list 407–8
window record 406
word processor x
XBA (eXchange A and B accumulators) instruction 50, 132, 585
XCE (eXchange Carry with Emulation bit) instruction 132
X register 47–49, 125, 131, 132, 181, 183
Y register 47–49, 125, 131, 133, 181, 183
zero flag 125, 126
zero page 290–91
To order your copy of *Apple IIgs Machine Language for Beginners Disk*, call our toll-free US order line: 1-800-346-6767 (in NY 212-887-8525) or send your prepaid order to:

*Apple IIgs Machine Language for Beginners Disk*  
**COMPUTE! Books**  
F.D.R. Station  
P.O. Box 5038  
New York, NY 10150

All orders must be prepaid (check, charge, or money order). NC residents add 5% sales tax. NY residents add 8.25% sales tax.

Send _____ copies of *Apple IIgs Machine Language for Beginners Disk* at $15.95 per copy. (971BDSK)

Subtotal $__________

Shipping and Handling: $2.00/disk $__________

Sales tax (if applicable) $__________

Total payment enclosed $__________

☐ Payment enclosed  
☐ Charge ☐ Visa ☐ MasterCard ☐ American Express

Acct. No. __________________ Exp. Date ______  
(Required)

Name ________________________________

Address __________________________________

City __________________ State ______ Zip ______

Please allow 4-5 weeks for delivery.
Programming the 65816

The Apple IIgs takes the Apple II personal computer to a new level of technology. With its power and speed, however, comes a steep learning curve. On the IIgs, machine language programming—which always produces the most elegant, the most compact, and the fastest software—requires that you learn how to use the machine's powerful microprocessor, the 65816. COMPUTE!'s Apple IIgs Machine Language for Beginners is the perfect introduction and tutorial to 65816 machine language on the latest Apple II.

This step-by-step guide is written in a light but informative style that's packed with information, yet is easy to read. Written for both beginning and experienced machine language programmers, COMPUTE!'s Apple IIgs Machine Language for Beginners is the definitive guide to programming in machine language on the Apple IIgs. It goes beyond the fundamentals to show you how to take advantage of the advanced features of this powerful computer. It's the one book every IIgs machine language programmer should own.

Here's a sample of what's inside:

- Complete tutorials for using the Merlin 8/16 and APW assemblers
- Clear explanations of the most important 65816 instructions, with comprehensive examples
- How to add machine language routines to Applesoft BASIC
- How to use the Toolbox routines in your machine language programs
- Managing windows and menus
- Using QuickDraw and the Event Manager
- A complete 65816 reference section
- Scores of programming examples
- And much more

The author, Roger Wagner, is a popular magazine columnist, guest speaker, software developer, and author of the popular book on 6502 machine language: Assembly Lines: The Book. Formerly a math and science teacher, Mr. Wagner is now president of Roger Wagner Publishing, Inc., a publisher of software for Apple computers.

$19.95
Errata:
Apple IIgs Machine Language for Beginners
by Roger Wagner
(revised 8/29/90 - ECM)

Page 19: Line 290 of the program should read:

```
290 POKE 49186,0S: POKE 49204,0B: REM RESTORE SCREEN
```

Pages 61, 67: The Main Menu of Merlin 16 displays F: Full Screen Editor, not "E: Editor, command mode."

Page 67: Paragraph 3: After a successful assembly, it is not necessary to type Q to Quit, as Merlin automatically returns to the Main Menu.

Page 69: Assembler Directives, paragraph 2: When a source file is loaded, Merlin 16 automatically goes to the editor, so pressing a command key is not necessary.

Page 93: The KEEP and ORG line positions should be interchanged so that ORG comes before the KEEP directive. In the program itself, the instruction to load the Accumulator with the letter "A" should read:

```
LDA #$C1 ; LETTER "A"
```

Page 94: The output listing for Program 5-6 is incorrect, and is more similar to, although not identical to, the output when assembling Program 5-7.

Chapter 6: APW users should note that when entering the example programs, the listings shown assume the 8 bit mode of the 65816. When using the APW assembler, you will have to remember to include the directives LONGA OFF and LONGI OFF at the beginning of your source listings. LONGI OFF tells the APW assembler to use the 8 bit mode for 'Index' register operations such as LDX, STY, etc.

Page 115: In the listing at the bottom of the page, the CMP instruction should be CMP #$FFFF. This compares the Accumulator to the immediate value of $FFFF, not the memory location $FFFF.

Page 133: Setting and clearing the Carry, paragraph 2: "If you want to clear the Emulation bit to 0 (sometimes called...)"

Above the last paragraph, the instructions should read:

```
SEP #$20 ; %00100000 binary
REP #$30 ; %00110000 binary
```

In the last paragraph, the text should read "...SEP #$20 sets bit 5, the m bit, to 1, thus setting the Accumulator ..." In the next sentence, the text should read "... REP #$30..."

Page 134: In the third paragraph, the last sentence should end: "... the 65816, i.e., the letter "A" followed by an inverse "@". Note that this program should be tested in the 40-column display mode."

Page 138: REP and SEP: The Monitor vs. Merlin: In Merlin 16, the instructions REP and SEP do not required the pound-sign ("#"). Thus, both REP #$20 and REP $20 are acceptable forms of the instructions. The dollar-sign ("$") is required to tell the assembler that the number is in hexadecimal notation. When using the Monitor to List (disassemble), however, you'll notice that the dollar-sign is not displayed. This is because the Monitor assumes all numbers shown (or entered in the Mini-Assembler) are in hex.

Page 146: "This gives the correct two-byte result of $0310, and is equivalent to..."

Page 152: Paragraph 4: Complementary angles are those two angles whose sum is 90 degrees, not 180 degrees. Sigh...
The two "ORA Value:" lines should read "EOR Value". The result of the second EOR operation for Example #2 is $83 = 1000 0011.

In Figure 10-2, the LDA instruction should appear in the form:

LDA [S80], Y

Between lines 16 and 17 of Program 11-1, the bytes for addresses $00030A-$00031F are data generated by the assembler for the ASC data on line 16, and should not be confused as text to be added by the reader. These bytes will not be tabbed as shown in the book into the assembler fields, but rather, will appear normally in the data area to the left during the assembly.

Between lines 36 and 37 in Program 11-1, you should insert the instruction CLC in preparation for the addition operation. This will change the length of the assembled file to 93 bytes.

Insert a line with the instruction INY just before the label LOOP in program 12-1.

Second paragraph from the bottom, the reference to line 36 should instead refer to line 42.

The last paragraph should be ignored since all Merlin listings in the book using BRK $00, and thus give the same object file length and listing appearance as the APW assembler.

Local labels can be any label following a colon (example :LOOP), in addition to :1 through :9.

The program example should use AND #$DF, not ORA #$DF.

Paragraph 5: Reference to lines 69 through 74 should be 79 through 84; reference to lines 69, 70 should be 79, 80.

Last paragraph: "... (DS followed by a \) which..., not "...followed by a Z...". The very last line should read DS \, not DSZ.

Paragraph 2, reference to "lines 76-83" should be 91-98. "Lines 90-96" should be 100-106.

The label on line 147 should be PARMTBL2; the label on line 152 should be PARMTBL3.

The byte displayed for location $00213E should be $DO, not $80. Line 175 should be DS \, a Merlin command that defines a dummy block sufficient to fill to the next page boundary in memory. Line 158 was omitted from the listing, and should read:

158 TR ON ; DON'T PRINT ALL HEX BYTES

Line 173 (the checksum) should have a value of $F6, with the changes made above.

Although the program will work as listed, the error handling on lines 89 and 106 is not really correct. When an error occurs during the reading of a ProDOS file, the file should be closed as part of the error-handling routine. Therefore, a better design would be for the JSR ERROR on line 89 to be followed by a JSR RDKEY and a JMP CLOSE, similar to the lines 72-74 above. Short of writing an entire routine to ask the user to re-insert disks, etc., an error in trying to close the file (line 106) can be handled with a JSR RDKEY and JMP BEGIN, as is done on lines 72-74.

Because of an early change to ProDOS 16, the listings shown in the book that access the keyboard and strobe locations no longer work unless they are done using long addressing. These programs will work with version 1.1 of ProDOS 16, but all later versions of ProDOS will require a change to the programs. In paragraph 3, the LDA KYBD should be LDAL KYBD.
Lines 12-14 of Program 14-1 should be:

12 KYBD EQU $E0C000 ; KEYBOARD SOFTSWITCH
13 STROBE EQU $E0C010 ; KEYBOARD STROBE
14 SCREEN EQU $E00400 ; LINE 1 ON SCREEN

Line 27 of Program 14-1 should be LDAL KYBD. Because the BIT instruction does not have a long addressing form, line 31 should be replaced with STAL STROBE. If you get a Fatal System Error $110A when trying to run this program, it means you are using the original version of ProDOS 16 (v. 1.1). Merlin 16 defaults to the later versions, so to solve your problem you need only re-boot and try the program again with version 1.2 or later of ProDOS 16.

The byte value displayed for the checksum (line 47) will be $0A after the changes above.

Because the program listing shown is in the 8-bit mode, only a single INX instruction is needed in the LOOP part of the program. The comment after the first INX should read "NEXT CHAR".

Lines 16-18 of Program 14-2 should be:

12 KYBD EQU $E0C000 ; KEYBOARD SOFTSWITCH
13 STROBE EQU $E0C010 ; KEYBOARD STROBE
14 SCREEN EQU $E00400 ; LINE 1 ON SCREEN

Line 31 should read LDAL KYBD; line 35 should be ST AL STROBE.

The checksum value in line 101 will be $CC after you make the changes above.

Make sure that when entering the text for the Linker command file shown, that you have the opcodes TYPE, LINK, etc. in the opcode column, and not the label column. It is also not necessary to run the Linker.GS as indicated in the manual, since all versions of Merlin.16 default to loading the Linker.GS when it starts up.

The instructions here are directed to assembling and linking program 14-1, the P16.SYSTEM program. The first paragraph after step 5 should read "To do the quick link, first load the P16.SYSTEM source file, and then immediately type NEW from the Command Box. Then type LINK with no specified pathname in the Command Box (CO). Merlin will use the last name (P16.SYSTEM) as a source file and will automatically assemble and link the file. The final object file generated by the link..."

The descriptions of Bit 3 and Bit 4 of the shadow register are reversed. Bit 3 controls the Super Hi-Res area, and Bit 4 controls Auxmem.

The labels COUT, HOME, KYBD, and STROBE on lines 7-10 are not used in the program, and as such are not needed in the program.

Line 55 should read BRA BOX. The label on line 57 should be SHUTDOWN. The checksum for this listing will then be $46.

Calling Tools from ProDOS 16: The reference to lines 29 and 32 in the second paragraph of the text should read lines 29 through 32.

The label on line 54 should be SHUTDOWN.

The example macro for _TLStartUp should read:

_TLStartUp MAC
LDX #$0201
JSL $E10000
EOM

In the interest of standardization, the Merlin 16 GS Tool Macro Library does use the underscore at the beginning of each tool name, contrary to the indication of the book.
Page 327: The third line of the PushWord macro should be:

```
PEA 1
```

Page 328: Contrary to the second paragraph from the bottom, the macros in listing 16-5 are *not* expanded since the listing does use the Merlin 16 LST OFF and EXP OFF directives.

Page 329: Line 30 should read LDA ^RESUME.

Page 345: The Command Value for MMStartUp should be $0202, not $0102.

Page 347: Second paragraph of "Using the Memory Manager:" You do not have to set your RAM disk to 800K to hold all the pictures created. Only about 144K is really needed. It will also take 144K of system (non-RAM disk) memory to store all the pictures, so you will need at least 512K of expansion RAM for program 17-1 to work. Program 17-1 does not set the ProDOS prefix, so you will have to put the program file on /RAM5 (or wherever) with the pictures to be loaded.

Page 348: Line 9 of Program 17-1 should not have an asterisk (which makes the line a comment). Rather DSK should appear in the opcode column with MM.DEMO.P8 as the operand. Lines 25-28 are missing from the listing, and appear as follows:

```
25   LST OFF    ; DON'T PRINT MACROS
26   USE UTIL.MACS    ; USE MACRO.LIBRARY
27   LST ON    ; LISTING BACK "ON"
28   EXP OFF    ; DON'T EXPAND MACROS
```

Page 349: MMSTART on line 49 should begin with a PushWord #0000, with the ToolCall $0202 on the next line. This will involve inserting a new line into the listing.

Page 354: The checksum on line 318 will be $0B after the changes above.

Page 355: In the first paragraph of text, the references to HGR should be HGR2. At the bottom of the page, the diagram for MMStartUp should have shown the requirement of pushing a word onto the stack as space for the result prior to doing the call.

Page 359: The Command Value for MTStartUp should be $0103; the value for MTShutDown should be $0303.

Page 361: The byte range indicated for the Minute value should be 0-59, not 9-59.

Page 363: The opcodes SEC and DEX on lines 74 and 81 should be indented to the opcode column in the listing.

Page 376: The fourth paragraph text reference to the Color Value column should refer to Master Value.

Page 381: The comment on line 119 should read "CLEAR HIGH BYTE"

Page 392: The end of the second paragraph should include: "(Add enough spaces to the end of each message so that the single quote mark is directly below the letter "L" in the word "TABLE" on line 323.)

Page 395: On line 104, the semi-colon is *not* the beginning of a comment. The line should read:

```
MSSG2 STR 'Press keys; use "Q" to Quit'
```

The comment on line 134 should read "QUIT" KEY (HI BIT CLR)?

Page 396, 397: Note that for lines 200 and 204, there should be 2 spaces after each 0000. On page 397, there should be 4 spaces after Event; and 6 spaces after Type:

Page 399: The label on line 323 should be EVENTMSSG. Be sure to add enough spaces to each of the messages on lines 325-340 so that there are 16 characters *including* the 00 at the end. When entered properly, the closing quote for each message will line up with the "L" in "TABLE" on line 323.
Insert the instruction **TAX** (to transfer the value of the Acc. to the X-register) between the instructions **ASL** and **JSR** (**MENTBL,X**) on both pages.

At the bottom of the page, the hexadecimal values for the characters in the Event Mgr. demo should be $11, $12, $13 and $14.

The last line on the page should refer to **NewMenu** in place of **NewHandle**.

The **RTS** on line 215 should be in the opcode column, not the label column.

Lines 420 and 421: the Z should be the \ character. On line 414, the comment should read "PUT WINDOW AT FRONT ($FFFF = -1)".

Merlin 16 uses **Open-Apple-Y** to select to the end of the listing, not **Open-Apple-Q**.

In the second paragraph, the first sentence should end "... called **SPECIAL**, which consisted of nothing more than an **RTS**.".

On the lines with **XMSG** and **YMSG**, there should be 4 spaces after the 1st 0000, and 2 spaces after the 2nd 0000.

In the source listing, the third line from the bottom should have a second **INY** instruction added, so that there are two **INY**’s. (needed to increment two bytes forward).

The **ERASE** routine has a bug in it. The **DOCSETUP** routine calls **ERASE** in order to erase the window. The **ERASE** routine fills the memory occupied by the window region with color #14 and then adds the rectangle to the current update region (causing the system to redraw it during the next call to TaskMaster). This works fine when **ERASE** is picked from the menu because the window has already been created... however, when **DOCSETUP** calls **ERASE**, the window has not been created yet, causing the **GetPortRect** and **InvalidRect** calls to lead to unexpected results (a.k.a. system crashes).

In order to fix this bug, change the label **ERASE** to **ERASE2**, and the **JSR ERASE** in **DOCSETUP** to **JSR ERASE2**. Finally, move the four lines above the **RTS** at the top of page 453 (PushLong **#WINRECT**, **ToolCall** $2004, **PushLong** **#WINRECT**, **ToolCall** $3A0E) above the new label **ERASE2** and place the label **ERASE** on the first PushLong statement.

The final chunk of code should look like this:

```
****
ERASE PushLong #WINRECT           ; POINT TO WINDOW RECTANGLE
          ToolCall $2004               ; GetPortRect
          ; MAKE WINRECT = WINDOW
PushLong #WINRECT                   ; THE WINDOW RECTANGLE
          ToolCall $3A0E               ; InvalidRect
          ; FORCE TASKMASTER TO UPDATE
ERASE2 PushLong PICHNDL            ; HLock
          ToolCall $2002              ; MAKE SURE IT DOESN'T MOVE
LDA [PICHNDL]                       ; LONG INDIRECT LOAD
STA PTR                                ; GET THE MEM ADDRESS
LDY $502                              ; (PTR) = ADDR. OF PICTURE
LDA [PICHNDL],Y                       ; CLEAR BLOCK OF MEMORY TO COLOR #14
STA PTR+2                             ; BEG. OF BLOCK
CLR $5EEEE                            ;
LDY $50000                             ;
```
Page 454: The tool calls SetPenSize and SetSolidPenPat work with the current port. In order to change the pen size and color in the drawing window, we must set make that window's GrafPort current. So, before the line with the label SETPEN, insert these two lines:

```
PushLong WPTR ; SET PORT TO OUR WINDOW
ToolCall $1B04 ; SetPort
```

Page 466: With the changes above, the checksum for the new sketcher program will be $96, not $05.

Page 485: The hex byte for the BCC opcode should be $90, not $6D.

Page 497: The Addressing Modes examples for BRL should be:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative Only</td>
<td>BRL LABEL</td>
<td>82 FF FF</td>
</tr>
<tr>
<td></td>
<td>BRL $FFFF</td>
<td>82 ff ff</td>
</tr>
</tbody>
</table>

Page 518: In the example program segment, there should be a TAX instruction after the ASL and before the JMP (CMDTBL,X).

Page 520: In the example program segment, there should be a TAX instruction after the ASL and before the JMP (CMDTBL,X).

Page 525: In the odd/even example program segment, EVEN and ODD are reversed. It should read BCS ODD and BCC EVEN.

Page 526: In accordance with the expressed preference of Apple Computer, the assembler syntax for the MVN and MVP instructions should be revised to reflect the requirement for a complete label as the operand for these instructions. Although the instruction itself only encodes the source and destination bank bytes, the assembler requires that the complete address be used. There is also an error in the hex code shown for the MVP instruction. Thus, the Addressing Mode Available chart should appear as follows:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Only</td>
<td>MVN LABEL1, LABEL2</td>
<td>54 00 FF</td>
</tr>
<tr>
<td></td>
<td>MVP LABEL1, LABEL2</td>
<td>44 00 FF</td>
</tr>
</tbody>
</table>

and the last line of the program segment on page 527 should read as:

```
MVN SRCE,DEST ; SOURCE AND DEST. ADDRESSES
```
ORA can also be used to convert upper case letters to lower case:

ENTRY LDA CHAR ; GET CHARACTER
CMP $8C1 ; "A" - ASSUMES HIGH BIT ASCII
BCC DONE ; LESS THAN "A"
CMP $8E0 ; 1ST LOWER CASE LETTER
BCS DONE ; GREATER OR EQUAL
XVERT ORA $20 ; SET BIT 5: UC -> LC
STA CHAR ; PUT CHAR BACK
DONE RTS

The Addressing Modes examples for PEA should be:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate Only</td>
<td>PEA $ffe</td>
<td>F4 ff ff</td>
</tr>
<tr>
<td>Indirect Only</td>
<td>PEI ($ef)</td>
<td>D4 ff</td>
</tr>
</tbody>
</table>

The Addressing Modes examples for PEI should be:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Common Syntax</th>
<th>Hex Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative Only</td>
<td>PER $ffe</td>
<td>62 xx xx</td>
</tr>
</tbody>
</table>

The first line of the PER example should read:

8000: 62 03 00 PER LABEL ; XVRTED TO 3 BY ASSEMBLER

In addition, the first paragraph after the example should read "The microprocessor will take the relative offset of 3 (the operand of the PER instruction), add this to the program counter for the next instruction ($8003), and push the result ($8006) on the stack."

RTI is equivalent to:

PLP
RTS (or RTL)

in that the status register is restored from the stack, and a return (RTS or RTL, depending on the processor status) is done using the address remaining on the stack.

The correct syntax for the Pattern Search command is as follows:

\"A\"<1234.5678P
\25<1234.5678P
\"ABCD\"<1234.5678P
\25 7F 3E<1234.5678P