



Assembly Lines: The Complete Book

A Beginner's Guide to 6502
Programming on the Apple II

Roger Wagner
edited by Chris Torrence

16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58

Assembly Lines: The Complete Book

A Beginner's Guide to 6502
Programming on the Apple II

by
Roger Wagner

edited by
Chris Torrence

© 2014 Roger R. Wagner

Second printing, April 2017.

This work is made available under a Creative Commons Attribution-NonCommercial-ShareAlike 2.0 license. You are free to share and adapt the material in any medium or format under the following terms: (1) Attribution—You must give appropriate credit, provide a link to the license, and indicate if changes were made; (2) NonCommercial—You may not use the material for commercial purposes; (3) ShareAlike—If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. For the complete license see <http://creativecommons.org/licenses/by-nc-sa/2.0/>.

Assembly Lines: The Complete Book is an independent publication and has not been authorized, sponsored, or otherwise approved by Apple Inc.

Apple, the Apple logo, and all Apple hardware and software brand names are trademarks of Apple Inc., registered in the U.S. and other countries.

The contents of Volume 1 (chapters 1–15, appendices A–E) were originally printed in *Assembly Lines: The Book* (Roger R. Wagner, Softalk Publishing, North Hollywood, CA, 1982).

The contents of Volume 2 (chapters 16–33) were originally printed in *Softalk* magazine (Softalk Publishing, North Hollywood, CA, January 1982–June 1983).

The cover images of the Apple II Plus and the green bar computer paper were created by Chris Torrence. The “Usage Chart of 6502 Instructions” is adapted from Fig. 2-1 in *Inside the Apple IIe*, by Gary B. Little, and is used by permission. All other images and figures are © Roger R. Wagner.

While every precaution has been taken in the preparation of this book, the publisher, author, and editor assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein or from the use of programs and source code that may accompany it.

ISBN 978-1-312-08940-2

**Assembly
Lines:**

The Book

**A Beginner's Guide
to 6502 Programming
on the Apple II**

by

Roger Wagner

Table of Contents

Preface.....	xi
Introduction.....	xvii
1. Apple's Architecture.....	1
6502 Operation	2
Memory Locations	2
Hexadecimal Notation	4
It's Culture That Counts	7
2. The Monitor.....	9
Exploring the Monitor	9
Disassembly	10
3. Assemblers.....	13
The Mini-Assembler	13
Assemblers	15
Load/Store Opcodes	18
Putting it All Together	19
Conclusion	20
4. Loops and Counters.....	21
Binary Numbers	22
The Status Register	22
Incrementing and Decrementing	23
Looping with BNE	24
5. Loops, Branches, COUT, and Paddles.....	27
Looping with BEQ	27
Branch Offsets and Reverse Branches	28
Screen Output Using COUT	29
Reading a Game Paddle	32
Paddle Program Problems	33
Transfer Commands	34
A Note about BRUN and COUT	35
6. I/O Using Monitor and Keyboards.....	37
Comparisons; Reading the Keyboard	37
Compare Commands and Carry Flag	38
Using Monitor Programs for I/O Routines	41
Reading Data from the Keyboard	42
7. Addressing Modes.....	45
Indexed Addressing	46
Sometimes X and Y Aren't Interchangeable	47
Storing Pure Data	48
8. Sound Generation.....	53
Delays	54
Delay Value in Memory	56
Delay from the Keyboard or Paddles	58
9. The Stack.....	61
Stack Limit	64

10. Addition and Subtraction.....	65
Binary Numbers	65
Addition with ADC	66
Subtraction	72
Positive and Negative Numbers	72
The Sign Bit	73
The Sign Flag	75
11. DOS and Disk Access.....	77
The Overview: DOS	77
Diskette Organization	78
DOS Modifications	85
Disk-Volume Modification	86
Catalog Keypress Modification	87
Bell Modification and Drive Access	88
12. Shift Operators and Logical Operators.....	89
Shift Operators	89
Logical Operators	92
BIT	96
ORA and EOR	97
13. I/O Routines.....	105
Print Routines	105
Input Routines	108
14. Reading and Writing Files on Disk.....	113
Reading and Writing Data Files	113
Reading and Writing Text Files	120
15. Special Programming Techniques.....	127
Relocatable versus Non-relocatable Code	127
JMP Commands	128
Determining Code Location	131
JSR Simulations	134
Self-Modifying Code	137
Indirect Jumps	139
16. Passing Data from Applesoft BASIC.....	143
Simple Interfacing	144
The Internal Structure of Applesoft	145
Passing Variables	147
17. More Applesoft Data Passing.....	151
Applesoft Variables	151
Memory Maps	152
Passing Variables to Assembly Language	156
Passing Data from Assembly Language	161
Programming Tip	164
Conclusion	165
18. Applesoft Hi-Res Graphics.....	167
Ground School	167
Landmarks and Entry Points	168
A Test Flight: Hi-Res Demo	169
A Minor Diversion	172
Location	173
Motion	173

19. Calling Hi-Res Graphics Routines.....	177
Taking the Opposite of a Signed Number	178
The Real Thing: Hi-Res in Assembly	179
Table-Driven Graphics	183
Conclusion	187
20. Structure of the Hi-Res Display Screen.....	189
Loading a Hi-Res Screen: the "Fill" Effect	189
Another Problem: Shifting Colors	192
Other Problems: When Is White Not White?	195
Super Hi-Res Graphics	195
21. Hi-Res Plotting in Assembly.....	197
Normal Point Plotting	197
Alternate Plotting Modes	200
140-Point Resolution Mode	201
560-Point Resolution Mode	203
A Demonstration Program	206
22. Even Better Hi-Res Plotting.....	207
Interactions between Adjacent Bytes	208
Some "New and Improved" Routines	209
PLOT.140+	210
PLOT.560+	212
PLOT.560-White	213
A Final Demo Program	217
Conclusion	218
23. Hi-Res Graphics SCRN Function.....	219
An Overview	221
Sample Program	223
Conclusion	224
24. The Collision Counter, DRAW, XDRAW.....	225
Some Experiments	225
DRAW versus XDRAW	227
Principles of Animation and Collision	228
The Scanner	228
The Possibilities	234
25. Explosions and Special Effects.....	235
Explosions, Rays, and Other Things That Go Bump in the Night	235
A Little More Sophistication	239
Putting it All Together	241
The Shooter Program	245
26. Passing Floating-Point Data.....	251
Internalization of Data: Integer versus Real Variables	252

The Floating-Point Accumulator (FAC)	254
Passing Data from Applesoft to the FAC	255
Moving the FAC to a Memory Location	257
Moving Memory into the FAC	258
Passing FAC Data Back to Applesoft	259
Putting it All Together	260
Conclusion	262
27. Floating-Point Math Routines.....	263
More Applesoft Internals	265
An Example That Doesn't Work	266
Why it Doesn't Work	267
A Little More Finesse	269
Other Operations: Subtraction, Multiplication, and So On	270
Conclusion	270
28. The BCD, or Binary Coded Decimal.....	271
Limitations	273
The Carry Flag	273
Common Operations	274
Printing BCD Values	276
Conclusion	279
Special Note: Counting Down	280
29. Intercepting Output.....	281
Output	281
Intercepting Output	283
Other Output Devices	286
Conclusion	290
30. Intercepting Input.....	291
The Input Vector: KSW	291
Other Input Sources	294
Interception Routines	295
Something More Useful: Lowercase Input	297
Conclusion	300
31. Hi-Res Character Generator.....	301
Text and Hi-Res Screen Mapping	301
The Character Generator	304
A Hi-Res Character Set	309
Conclusion	312
32. Hi-Res Character Editor.....	313
How it Works	320
And Now with the Magnifying Glass	321
Running the Editor	325
Miscellaneous Notes	326
Conclusion	326
33. The 65C02.....	327
New Addressing Modes	328
Indirect Addressing	328
Indexed Absolute Indirect	329
New "Standard" Addressing Modes	330
At Last, the Real Scoop! New Instructions	331
Other Differences	335

Appendix A: Contest.....	339
Appendix B: Assembly Commands.....	344
Appendix C: 6502 Instruction Set.....	394
6502 Microprocessor Instructions	394
Usage Chart of 6502 Instructions	395
6502 Instruction Codes	398
65C02 Instruction Codes	402
Hex Operation Codes	403
Appendix D: Monitor Subroutines.....	404
Output Subroutines	404
Input Subroutines	405
Low-Res Graphics Subroutines	406
Hi-Res Graphics Subroutines	407
Floating Point Accumulator	408
Other Subroutines	410
Appendix E: ASCII and Screen Charts.....	411
You Get What You ASCII For...	411
Text Screen Memory Map	416
Hi-Res Memory Map	417
Appendix F: Zero-Page Memory Usage.....	418
Special Locations	418
Memory Usage Table	419
Appendix G: Beginner's Guide to Merlin.....	420
Control Modes	420
Getting Started	421
Deleting Lines	423
Inserting Lines	423
Editing Lines	424
Assembling the Code	424
Saving and Running Your Program	425
List of Programs.....	426
Directory Listing for Program Disks.....	427
Index.....	428
Quick Reference.....	432

Preface

In October 1980, in the second issue of *Softalk* magazine, a new series of articles made its debut. Its title was “Assembly Lines” with the subtitle “Everyone’s Guide to Machine Language.”¹ The author was Roger Wagner, the president of Southwestern Data Systems. By then, Roger had already established himself as a well-respected software publisher who cared about the end user. As Al Tommervik states, “His programs reflect concern that the user get more than utility—he should also gain knowledge—from use of the software.”²

Before that issue of *Softalk*, a few brave souls had learned assembly language all on their own, using clues from the ROM listings in the *Apple II Reference Manual*. These included developers such as Jordan Mechner (creator of *Karateka*), Silas Warner (*Castle Wolfenstein*), and Dan Bricklin and Bob Frankston (*VisiCalc*). But now here was a series of articles that taught the rest of us how to program the 6502.

I first became acquainted with *Assembly Lines* when, as a new Apple II Plus owner, I received my complimentary issue of *Softalk* magazine in January 1982. The series was already on Part 16; it had moved beyond the basics and was beginning to explore sound and hi-resolution graphics. Despite missing the crucial introductory articles, I eagerly looked forward to receiving *Softalk* each month to see what Roger Wagner had to teach us about the 6502 and assembly language.

Then, in March 1982, *Softalk* announced that it was publishing Roger’s articles in book form. *Assembly Lines: The Book* contained the first fifteen articles plus an appendix of 6502 assembly-language commands, Monitor subroutines, and an index.

Roger continued to write his monthly “Assembly Lines” articles until June 1983. Part 33 contained an introduction to the new 65C02 chip and ended with a farewell:

I want to thank the many readers of this column over the last several years for their enthusiastic support and valuable suggestions. I

¹ Eventually changed to “Everyone’s Guide to Assembly Language.”

² Tommervik, Allan, “Exec SDS: Southwestern Data Systems, Assembling Useful Utilities,” *Softalk*, August 1981 (Softalk Publishing Inc.), pp. 30–32.

have always believed that the human element to this industry, and in fact any endeavor, is the truly rewarding part.³

The article then ended with a note from the Editor, stating:

The first year's columns plus appendixes and revisions have been available for some time in *Assembly Lines: The Book*. Volume 2, covering the rest of the columns, will be released shortly by Softalk Books.

Despite numerous announcements about Volume 2 over the next year, when Softalk Publishing Inc. went bankrupt in August 1984, *Assembly Lines: The Book*, Volume 2 remained incomplete and unpublished.

It is therefore a great privilege and a long-overdue honor to present *all* of Roger Wagner's "Assembly Lines" articles in one complete volume. This volume contains all of the original *Assembly Lines: The Book*, including the appendixes, plus the content of the remaining eighteen *Softalk* articles.

The complete volume should appeal to long-time readers who may not have access to the original *Softalk* articles, especially those articles from the missing Volume 2. I also hope that Roger Wagner's clear explanations and his subtle but ever-present humor will encourage *new* readers to discover the joys of 6502 assembly-language programming on the Apple II. As David Finnigan notes in *The New Apple II User's Guide*, "There are still so many programs to be written, experiments to be conducted, and adventures to be had."⁴

With Roger Wagner leading the way, and with tractor-feed paper in one hand and *Merlin Assembler* at our side, who knows what amazing programs we can create?

Chris Torrence
Louisville, Colorado
December 1, 2014

³ Wagner, Roger, "Assembly Lines, Part 33," *Softalk*, June 1983 (Softalk Publishing Inc.), pp. 199–204.

⁴ Finnigan, David, *The New Apple II User's Guide* (Mac GUI, Lincoln, IL), p. xi.

Changes from the Original

In the original *Assembly Lines: The Book*, the first two *Softalk* articles (October and November 1980) were combined into chapter one, “Apple’s Architecture.” In this edition, that chapter has been re-split into two chapters: the first on the Apple’s architecture and the second on the Apple Monitor. By doing this, all of the chapter numbers now agree with the original *Softalk* article numbers.

All of the assembly-language programs now include the CHK pseudo-opcode at the bottom. The CHK instruction inserts a single-byte checksum at the end. You can use this checksum to verify that you have typed in the program correctly. CHK is available in the *Merlin Macro Assembler*. If your assembler does not support this pseudo-opcode you can ignore it in the code.

The original programs included the OBJ pseudo-opcode, which establishes the address at which the code will be assembled. The OBJ directive is not usually necessary and is incompatible with later versions of the *Merlin* assembler (*Merlin Pro* and *Merlin 8/16*). In this edition all of the OBJ lines have been commented out.

Spelling, grammar, and other minor corrections have been quietly made. More significant corrections (such as coding errors) are marked with a footnote and my initials [CT].

In Appendix B and C, the 65C02 instructions have been added. In Appendix C, the *Instruction Codes* table now contains a column with the clock cycles for each instruction. The *Usage Chart* in Appendix C was adapted from the chart in *Inside the Apple //e* by Gary B. Little and is used with his permission. I also added a new Appendix F (*Zero-Page Memory Usage*) and Appendix G (*Beginner’s Guide to Merlin*).

Acknowledgements

This edition was created on a MacBook Pro using OpenOffice 4.1.1. The assembly code was created using the *Merlin Macro Assembler* on an Apple //e computer and the Virtual][emulator (<http://www.virtualii.com>). The images were scanned using the FlipPal Mobile Scanner (<http://flip-pal.com>). The cover image of the green bar computer paper was created using IDL 8.4.

The text for Volume 1 was taken from the electronic version available from the Open Library, which is a project of the Internet Archive.

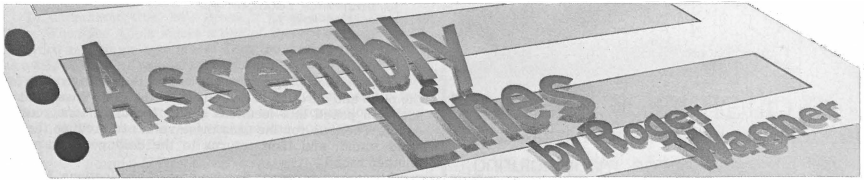
For Volume 2, I would like to acknowledge the generous help of Jim Salmons, Timlynn Babitsky, and Peter Caylor of the *Softalk* Apple Project, who provided PDF and OCR versions of the *Assembly Lines* articles. Their enthusiasm and expert advice made this project possible.

Special thanks is given to my fearless proofreaders: John Gruver, Antoine Vignau, and Shawn Lewis. Through their dedication and attention to detail they

caught many insidious errors. Thanks also to the members of 6502.org and comp.sys.apple2, who provided suggestions for chapter 33 on the 65C02. Additional thanks to fellow Apple II enthusiast Marc Golombeck, who provided valuable feedback on the second printing.

I could not have completed this book without the help and support of my wonderful family: my wife Gigi and my daughters Mia and Elyssa. Thanks to Gigi for being a guinea pig on the *Beginner's Guide to Merlin* appendix, and to Mia and Elyssa for reading pages of hex codes out loud while I proofread the programs.

Finally, I would like to thank Roger Wagner for giving his permission for the project, for answering innumerable emails about *Assembly Lines* minutiae, and for all of his contributions to the Apple II community over the years.



Volume I

Introduction

One often gets the impression that programming in assembly language is some very difficult and obscure technique used only by *those advanced programmers*. As it happens, assembly language is merely different, and if you have successfully used Integer or Applesoft BASIC to do some programming, there's no reason why you can't use assembly language to your advantage in your own programs.

This book will take a rather unorthodox approach to explaining assembly programming. Because you are presumably somewhat familiar with BASIC, we will draw many parallels between various assembly-language techniques and their BASIC counterparts. An important factor in learning anything new is a familiar framework in which to fit the new information. Your knowledge of BASIC will provide that framework.

I will also try to describe initially only those technical details of the micro-processor operations that are needed to accomplish our immediate goals. The rest will be filled in as we move to more involved techniques.

This book does not attempt to cover every aspect of assembly-language programming. It does, however, provide the necessary information and guidance to allow even a somewhat inexperienced person to learn assembly language in a minimum of time. You should find the text and examples quite readable, without being overwhelmed by technical jargon or too much material being presented at once.

I'd like to take this opportunity to briefly mention a few of my own programming philosophies. Writing programs to do a given task is essentially an exercise in problem solving. Problem solving is in fact a subject in itself. No matter what your programming goal is, it will always involve solving some particular aspect that, at that moment, you don't really know how to solve. The most important part is that, if you keep at it, you eventually will get the solution.

One of the key elements in this process, I believe, and the particular point to stress now, is that it is important to be a tool user. Programming in any language consists of using the various commands and functions available to you in that language and of putting them all together in a more complex and functioning unit. If you are not familiar with the options you have at any given moment—that is, your tools—the problem-solving process is immensely more difficult.

My intent in this book is to present in an organized way the various operations available in assembly language and how they can be combined to accomplish simple objectives. The more familiar you are with these elements, the easier it will be to solve a particular programming problem.

You may wish to keep your own list of the assembly-language commands and their functions as we go along. A list of these commands is included in Appendix C, but I think you'll agree that by taking the time to write each one down as you learn it, along with your own personal explanation of what it does, you will create a much stronger image in your mind of that particular operation.

You may wish to supplement this book with other books on 6502 programming. Recommended books include:

Randy Hyde, *Using 6502 Assembly Language* (Northridge, CA: DataMost, 1981).

Don Inman and Kurt Inman, *Apple Machine Language* (Reston, VA: Reston Publishing, 1981).

Lance A. Leventhal, *6502 Assembly Language* (Berkeley: Osborne/ McGraw-Hill, 1979).

Rodnay Zaks, *Programming the 6502* (Berkeley: Sybex, 1981).

There are undoubtedly others that are also available, and you should consider your own tastes when selecting which ones seem most appropriate to your own learning style.

An additional concern for a book like this is which assembler to use. (An assembler is an editor-like utility for creating assembly-language programs. If you're vague on this check chapter three for more information.) Although I'm somewhat biased, my favorite assembler is the one available from Southwestern Data Systems called *Merlin*. It not only contains a good assembler, but also a number of additional utilities and files of interest. *Merlin* is not required, however, as the examples given are written to be compatible with most of the assemblers currently available. These include the *Apple DOS Tool Kit*, *TED II*, the *S-C Assembler*, and many others.

Also available from Southwestern Data Systems is a utility called *Munch-A-Bug* (MAB) which allows a person to easily trace and de-bug programs, a process which can be of tremendous help, MAB also includes its own mini-assembler which can be used for the beginning listings provided in this book.

In terms of hardware, any Apple II or Apple II Plus should be more than adequate for your needs and no additional hardware is required. Disk access is discussed in several chapters, but is otherwise not a concern throughout the remainder of the book.

One warning before you start into the subject of assembly-language programming. As with any nontrivial endeavor, many people sell themselves short

because of what I call the instant expert myth. How many people hear someone play a piano well, and say, "My, what a beautiful thing. I think I'll get one and learn how to play myself." They then spend a substantial amount of money, sit down, and press a few keys. Surprise! To their great disappointment, the Moonlight Sonata does not magically flow from their fingers! They usually then become immediately discouraged and never pursue the area further, turning something that could give them tremendous pleasure into an expensive means of support for a flower vase.

I've seen this same effect in almost every area of human activity. If what you wanted was the Moonlight Sonata, a record will produce the sound you desire. People know that it takes talent (talent = 99% practice = 99% time) to play well, but are then disappointed when they can't sit down and perform like an expert immediately.

One of the great secrets to learning anything is to be satisfied with minor learning steps. Playing the Apple is in many ways much easier than learning to play a piano, but you should still not expect to sit down and write the world's greatest database in your first evening.

Set yourself some simple and achievable goals. Can you move one byte from one memory location to another? If you can you're well on your way to mastering programming. My feeling is that virtually anyone can become better than eighty to ninety percent of his fellow citizens in any area simply because eighty to ninety percent of the other people aren't willing or inclined to spend the necessary time to learn the skill. Reaching the top ninety-nine percent is certainly difficult, but ninety-five percent is surprisingly easy.

This book is written with the intention of providing those simple achievable steps. And surprisingly enough, by the time you finish this book you will have written a simple database of sorts, along with some sound routines, some programs that use paddles and the disk, and a few other nifties as well!

So hang in there and don't expect to be an expert on page five. I will guarantee that by page one hundred you may even surprise yourself as to how easy assembly-language programming really is.

One final note. I'd like to thank Al Tommervik for his tremendous help and support in this project as both editor and friend, and Greg Voss who provided many insightful suggestions in transforming the monthly series into the book. Also Eric Goetz for his encouragement to never accept less than the best, and his attentive (if not enthusiastic) listening to my various plans over the years.

Last but not least my thanks and sincere thoughts of appreciation to the many people that have shared in my own experiences in computing over the last few years. Whether they were readers of the column, users of my programs, or the wealth of new friends that have entered my life via the Apple, they have made all my efforts more than worthwhile and brought rewards beyond any simple economic gains of an ordinary job.

Alas for anyone who thinks that computers lead to a loss of the humanistic aspects of life. They need only look to the amazing community that has been drawn together from all parts of the world by the Apple to see that friendship and human creativity will always outshine the simple tools we use to express ourselves.

My wish for you, dear reader, is that you receive as much enjoyment from the Apple and programming as I have.

Roger Wagner
Santee, California
December 1, 1981

Apple's Architecture

October 1980

The first area to consider is the general structure of the Apple itself. To help visualize what's going on in there, why not take a look inside. That's right—rip the cover off and see what's in there! Don't be timid—get your nose right down in there and see what you shelled out all those hard-earned bucks for.

Providing you haven't gotten carried away in dismembering your Apple, the inner workings should appear somewhat like those in the photo below.



The main items of interest are the 6502 microprocessor (A) and the banks of memory chips (B). If you're not an electronics whiz, it really doesn't matter. You can take it as a device of magic for all it matters. The memory chips have the capability of storing thousands of individual number values and the 6502 supervises the activities therein. All the rest of the electronic debris within is supplied only to support the memory and the 6502. The circuits allow you to see displays of this data on the screen, and permit the computer to watch the keyboard for your actions.

The screen and keyboard are rather secondary to the nature of the computer and are provided only to make you buy the thing. As far as the Apple is concerned, it could talk to itself perfectly well without either the screen or the keyboard.

6502 Operation

So how does it work? The heart of the system is the 6502 microprocessor. This device operates by scanning through a given range of memory addresses. At each location, it finds some particular value. Depending on what it finds, it executes a given operation. This operation could be adding some numbers, storing a number somewhere, or any of a variety of other tasks. These interpreted values are often called *opcodes*.

In the old days, programmers would ply their trade by loading each opcode, one at a time, into successive memory locations. After a while, someone invented an easier way, using a software device to interpret short abbreviated words called *mnemonics*. A mnemonic is any abbreviated command or code word that sounds somewhat like the word it stands for, such as STX for STore X. The computer would then figure out which values to use and supervise the storing of these values in consecutive memory locations. This wonder is what is generally called an assembler. It allows us to interact with the computer in a more natural way. In fact, BASIC itself can be thought of as an extreme case of the assembler. We just use words like PRINT and INPUT to describe a whole set of the operations needed to accomplish our desired action.

In some ways, assembly language is even easier than BASIC. There are only fifty-five commands to learn, as opposed to more than one hundred in BASIC. Machine code runs very fast and generally is more compact in the amount of memory needed to carry out a given operation. These attributes open up many possibilities for programs that would either run too slowly or take up too much room in BASIC.

Memory Locations

Probably the most unfamiliar part of dealing with the Apple in regard to machine-level operations is the way addresses and numbers in general are treated. Unless you lead an unusually charmed life, at some point in your dealings with your Apple you have had it abruptly stop what it was doing and show you something like this:

```
8BF2- A=03 X=9D Y=00 P=36 S=F2
```

This occurs when some machine-level process suddenly encounters a break in its operation, usually from an unwanted modification of memory. Believe it or not, the Apple is actually trying to tell us something here. Unfortunately, it's rather like being a tourist and having someone shout, "*Alaete quet beideggen!*" at you.¹ It doesn't mean much unless you know the lingo, so to speak...

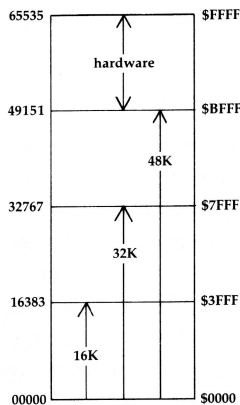
What has happened is that the Apple has encountered the break we mentioned and, in the process of recovering, has provided us with some information as to where the break occurred and what the status of the computer was at that crucial moment. The message is rather like the last cryptic words from the recently departed.

The leftmost part of the message is of great importance. This is where the break in the operation occurred. Just what do we mean by the word *where*? Remember all that concern about whether you have a 16K, 32K, or 48K Apple? The concern was about the number of usable memory locations in your machine. This idea becomes clearer through the use of a *memory map*, such as the one shown below.

Inside the Apple are many electronic units that store the numerical values we enter. By numbering these units, we assign each one a unique *address*. This way we can specify any particular unit or memory location, either to inquire about its contents or to alter those contents by storing a new number there.

In the Apple there are a total of 65536 of these memory locations, called *bytes*. The chart gives us a way of graphically representing each possible spot in the computer.

When the computer shows us an address, it does not do it in a way similar to the numbers on the left of the memory map, but rather in the fashion of the



¹ "Watch where you're stepping you nerd!" (in case you're not familiar with this particular dialect.)

ones on the right. You may well remark here: “I didn’t know BFFF was a number; it sounds more like a wet sneaker...”²

Hexadecimal Notation

To understand this notation, let’s see how the 6502 counts. If we place our byte at the first available location, its address is \$0. The dollar sign is used in this case to show that we are not counting in our familiar decimal notation, but rather in hexadecimal (base sixteen) notation, usually called *hex*, which is how the computer displays and accepts data at the Monitor level.

After byte \$0, successive locations are labeled in the usual pattern up to \$9. At this point the computer uses the characters A through F for the next six locations. The location right after \$F is \$10. This is not to be confused with ten. It represents the decimal number sixteen. The pattern repeats itself as in usual counting with:

\$10, \$11, \$12, \$13... \$19, \$1A, \$1B... \$1E, \$1F, \$20

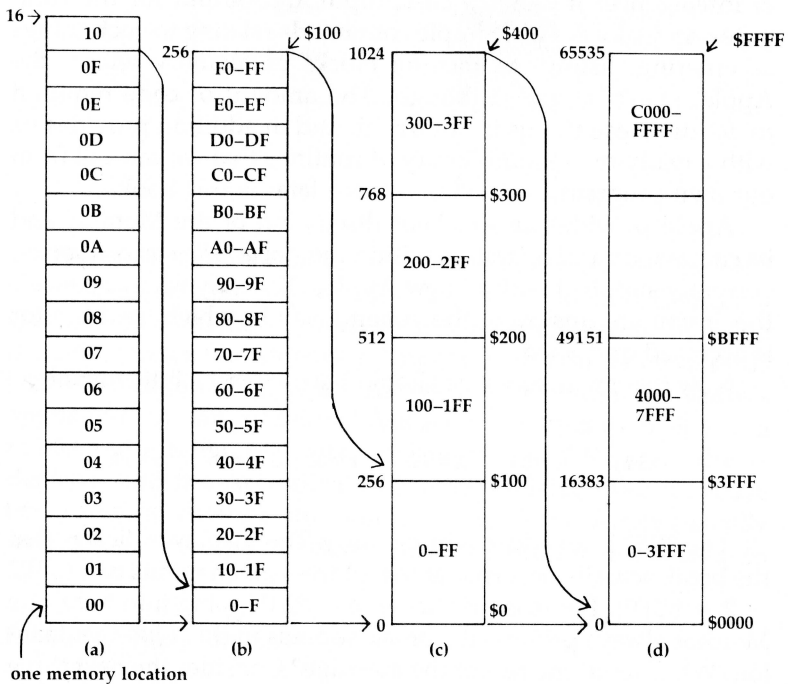
Try not to let this way of counting upset you. The pattern in which a person (or machine) counts is rather arbitrary, and should be judged only on whether it makes accomplishing a task easier or not. The biggest problem for most people is more a matter of having been trained to use names like *one hundred* when they see the numerals 100. How many items this corresponds to really depends more on the conventions we agree to use than on any cosmic decree. To aid in your escape from your possibly narrow view of counting, you may wish to read the diversionary story at the end of this chapter. In any event, it will be sufficient for our purposes to understand that \$1F is as legitimate a number as 31.

The hex number \$FF (255) is the largest value a single byte can hold. A block of 256 bytes (for instance \$00 to \$FF) is often called a *page* of memory. In the figure at right, all the addresses from \$00 to \$FF are shown in block (b). Four of these blocks together, as in (c), make up 1K of memory. As you can see, there are actually 1,024 bytes in 1K. Thus a 48K machine actually has 49,152 bytes of RAM (Random Access Memory).

Block (d) shows the Apple’s entire range again. If you do not have a full 48K of memory, then the missing range will just appear to hold a constant value (usually \$FF), and you will not be able to store any particular value there.

The range from \$C000 to \$FFFF, an additional 16K, is all reserved for hardware. This means that any data stored in this range is of a permanent nature and cannot be altered by the user. Some areas are actually a physical connection to things like the speaker or game switches. Others, like \$E000 to \$FFFF are filled in by the chips in the machine called ROMs.

² [John Gruver] Or, these days, maybe “Best Friends For Forever...”



ROM stands for Read Only Memory. These chips hold the machine-language routines that make up either Applesoft BASIC or Integer BASIC, depending on whether you have an Apple II Plus or the standard model. One of the chips is the Monitor, which is what initializes the Apple when it is first turned on so you can talk to it.

The Monitor can be thought of as a simple supervisor program that keeps the Apple functioning at a rather primitive level of intelligence. It handles basic input and output for the computer, and allows a few simple commands relating to such things as entering, listing, or moving blocks of memory within the Apple. Don't be fooled though. The amount of code required to do just these things is not trivial, and in addition provides us with a ready-made mini-library of routines that we can call from our own programs, as will be shown later in this book.

Apple provides an excellent discussion of the Monitor and its commands and operation within the *Apple II Reference Manual*, currently supplied with all new Apples. You may wish to consult this if you are unsure of the general way in which the Monitor is accessed and used. Now that break message should have at least a little meaning.

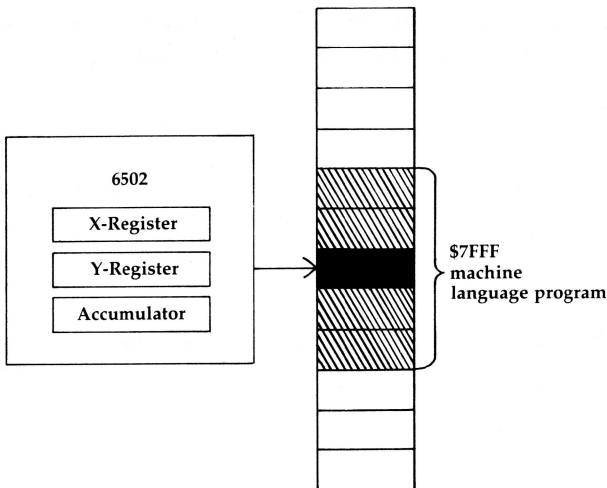
8BF2- A=03 X=9D Y=00 P=36 S=F2

The \$8BF2 is an address in memory. The display indicates that the break actually occurred at the address given minus two ($\$8BF2 - 2 = \$8BF0$). For reasons that aren't worth going into here, the Monitor always prints out a break address in this plus-two fashion. What about the rest of the message? Consider the next three items:

A=03 X=9D Y=00

The 6502, in addition to being able to address the various memory locations in the Apple, has a number of internal *registers*. These are units inside the 6502 itself that can store a given number value, and they are individually addressable in much the same way memory is. The difference is that instead of being given a hexadecimal address, they are called the Accumulator, the X-Register, and the Y-Register. In our error message, we are being told the status of these three registers at the break.³

The figure below illustrates what we know so far. The 6502 is a microprocessor chip that has the ability to scan through a given range of memory, which we will generally specify by using hex notation for the addresses. Depending on the values it finds in each location as it scans through, it will perform various operations. As an additional feature to its operation, it has a number of internal registers, specifically the Accumulator, the X-Register, the Y-Register. Memory-related operations are best done by entering the Monitor level of the Apple (usually with a CALL -151) and using the various routines available to us.



³ [CT] The final two items are the Status Register "P" and the Stack Pointer "S".

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 one 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. It would seem the tribe had been isolated from the rest of the world for at least 10,000 years. An interesting aspect of their life was a huge population of dogs living among the people. In fact, dogs so outnumbered 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 more rational base ten. They counted in the equivalent of base four.

In counting, they would be heard to say, "one, two, three..." Since they had never developed more than four symbols to count with (0, 1, 2, 3) when they got to the number after three, they wrote it as 10 and called it doggy, thus confirming the quantity in terms of a natural unit in their environment. Continuing to count they would say, "doggy-one (11), doggy-two (12), doggy-three (13)..."

At this point they would write the next number as 20 and call it twoggy. A similar procedure was used for 30.

20—twoggy	30—troggy
21—twoggy-one	31—troggy-one
22—twoggy-two	32—troggy-two
23—twoggy-three	33—troggy-three

Now, upon reaching 33, the next number must again force another position in the number display.

You're probably wondering what they called it. The digits are of course 100. Oh, the name? Why, of course, it's *one hundred*.

The Monitor

November 1980

Exploring the Monitor

It is possible to program the computer manually by entering numbers one at a time into successive memory locations. A program of this sort is called a *machine language* program because the 6502 can directly run the coded program steps. Humans, however, find this type of data difficult to read and are more likely to make mistakes while working with it.

A more convenient method of programming is to assign some kind of code word to each value. The computer will translate this word into the correct number to store in memory. This translation is done by an *assembler*, and programs entered or displayed in this manner are called *assembly-language* programs.

As an example, let's look at some data within your Apple, first in the machine-language format and then in the assembly-language format. First we must enter the Monitor. Type in:

```
CALL -151
```

This should give you an asterisk (*) as a prompt. Now type in:

```
F800.F825
```

This tells the Monitor we want to examine the range of memory from \$F800 to \$F825. The general syntax of the command is:

```
<start address>.<end address>
```

the period being used to separate the two values.

Upon hitting <RETURN> you should get the following data:

```
F800- 4A 08 20 47 F8 28 A9 0F
F808- 90 02 69 E0 85 2E B1 26
F810- 45 30 25 2E 51 26 91 26
F818- 60 20 00 F8 C4 2C B0 11
F820- C8 20 0E F8 90 F6
*
```

The range I have picked is the very beginning of the Monitor ROM. The data here can be directly read by the 6502, but is very difficult for most humans to make much sense of. This is machine language.

Disassembly

Now type in:

```
F800L
```

This tells the Monitor to give us a disassembly of the next twenty instructions, starting at \$F800. The syntax here is:

```
<start address>L
```

To disassemble means to reverse the process we talked about earlier, taking each number value and translating it into the appropriate code word.

After hitting <RETURN> you should get:

```
F800- 4A      LSR
F801- 08      PHP
F802- 20 47 F8 JSR  $F847
F805- 28      PLP
F806- A9 0F    LDA  #$0F
F808- 90 02    BCC  $F80C
F80A- 69 E0    ADC  #$E0
F80C- 85 2E    STA  $2E
F80E- B1 26    LDA  ($26),Y
F810- 45 30    EOR  $30
F812- 25 2E    AND  $2E
F814- 51 26    EOR  ($26),Y
F816- 91 26    STA  ($26),Y
F818- 60      RTS
F819- 20 00 F8 JSR  $F800
F81C- C4 2C    CPY  $2C
F81E- B0 11    BCS  $F831
F820- C8      INY
F821- 20 0E F8 JSR  $F80E
F824- 90 F6    BCC  $F81C
```

This is a disassembled listing. Although it probably doesn't do a lot for you right now, I think it's obvious that it is at least more distinctive.

Let's look at it a little more closely. In BASIC, line numbers are used to begin each set of statements. They're particularly handy when you want to do a GOTO or GOSUB to some other part of the program. In assembly language, the addresses themselves take the place of the line numbers. In our example, the column of numbers on the far left are the addresses at which each operation is found. To the right of each address are one to three hex values, which are number values stored in successive addresses. These are the opcodes with their accompanying operands.

At \$F802, for instance, is the opcode \$20. Remember, the dollar sign is used to show we are using base sixteen. \$20 is the opcode for the command JSR. All mnemonics are made up of three letters. In this case, JSR stands for Jump to SubRoutine and is rather like a GOSUB in BASIC. The next two numbers, \$47 and \$F8, comprise the operand, that is, the number that the opcode is to use in its operation. To the right we see that these numbers give \$F847 as the object of the, JSR¹. Continuing with our analogy, what would be a GOSUB 1000 in BASIC appears as a JSR \$F847 in assembly language. The command JSR \$F847 will jump to the subroutine at \$F847 and return when done.

You've just learned your first word of assembly language: JSR! Looking through the listing, you can see several of these. The first one goes to some routine outside the listing. What about the other two JSR commands? You should be able to see that they reference routines within the listing. The second enters at \$F800, the third at \$F80E.

In BASIC, a GOSUB eventually ends with a RETURN. The JSR has an analogous counterpart. Looking at the entry point at \$F80E and what follows, can you find anything that looks like it might be the equivalent of a RETURN? Take the time to find it if you can before reading on.

If you picked the RTS, you're right, RTS stands for return from subroutine. As with a RETURN, when the program reaches the RTS, it returns to where it originally came from. Encountering the RTS at \$F818, program execution would resume at \$F824, if entry was from the JSR \$F80E at \$F821.

You might notice that almost all machine code blocks that you may have used along with BASIC programs, such as tone routines, usually end with a \$60 as the last byte. This is the opcode for RTS. In almost any assembly-language program you write, you must end with an RTS. This is because, to the computer as a whole, your program is a temporary subroutine of its overall operation.

When your program ends, the RTS lets the Apple return to its original operations of scanning the keyboard and such. When you do a CALL 768 from BASIC, for example, you are essentially doing a JSR to that machine routine. The 768 is the decimal value for the address of the start of the routine, equivalent to \$300 in hexadecimal. At the end of that routine, the RTS returns you to your BASIC program to let it continue with the next statement.

¹ Notice that it takes two bytes to store the value for an address. For example, for the address \$F847, the value "F8" is stored in one byte, and "47" in another. Reading an address is generally a matter of mentally combining the two bytes. The byte representing the left-hand portion of the number is often called the high-order byte; the byte representing the right-hand portion is called the low-order byte. It is important to realize that the two bytes that make up an address are almost always reversed in regards to what you might normally expect. That is to say that in an address byte-pair, the low-order byte always comes first, immediately followed by the high-order byte. This means that when examining raw memory, you must mentally reverse the byte to determine the address stored. Fortunately when using the "L" command, the disassembler does this for you.

3

Assemblers

December 1980

The Mini-Assembler

I mentioned earlier that the basic principle of the Apple is its ability to scan through a range of memory and execute different operations depending on what numeric values it finds at each location, or address. Instead of tediously loading each location by hand with mundane numbers to create a program, an assembler is used to translate abbreviated codewords, called mnemonics, into the proper number values to be stored in memory.

The types of assemblers available are quite diverse, and range from the Mini-Assembler present in an Apple with Integer BASIC (or the *Munch-A-Bug* package) to sophisticated editor/assemblers like *Merlin*.

For now, we'll use the Mini-Assembler to try a short program. If you have an Apple II, an Apple II Plus with a language card, or an Apple //e, the Mini-Assembler is available provided that you enter the Monitor from Integer BASIC. In any case, you'll want to get a more complete assembler to do any real program writing.

Starting with chapter four, I'll assume you have an assembler, and have learned at least enough about operating it to enter a program. Since the only two commands we have at this point are JSR and RTS, our routine will be very simple. In the Monitor at \$FBDD is a routine that beeps the speaker. Our routine will do a JSR to that subroutine, then return to BASIC via an RTS at the end.

To enter the program using the Mini-Assembler, follow these steps:¹ From Integer BASIC, enter the Monitor with a CALL -151. Then type in:

```
F666G
```

F666 is the address where the Mini-Assembler program starts. G tells the Monitor to execute the program there. You can think of G as go; its BASIC equivalent is RUN. The general syntax is:

¹ If you do not have the Mini-Assembler available, you can enter the same data into memory by entering the Monitor and typing in: 300: 20 DD FB 60<RETURN>.

Rejoin us at the 300L mark on the next page.

If you have an Apple with a 65C02, the memory addresses have changed. You can enter the Mini-Assembler by typing "!", and exit the assembler by hitting <RETURN>.


```
<start address>G
```

The prompt should change to an exclamation mark (!). To use the Mini-Assembler, you must follow a basic pattern of input. See page 49 in the newest *Apple II Reference Manual* for a thorough description of this. For now, though, enter:

```
!300: JSR FBDD<RETURN>
```

The Apple will immediately rewrite this as:

```
0300- 20 DD FB JSR $FBDD
```

The input syntax is to enter the address at which to start the program followed by a colon and a space, then enter the mnemonic, another space, and then the operand, in this case the address for the JSR to jump to. Next type in:

```
! RTS<RETURN>
```

which will be rewritten as:

```
0303- 60 RTS
```

Be sure to enter one <SPACE> before the RTS. What the assembler has done is to take our mnemonic input and translate it into the numeric opcodes and operands of actual machine language.

Now type in:

```
!$FF59G
```

This will exit the Mini-Assembler, giving you back the asterisk prompt (*) of the Monitor. You can now list your program by typing in:

```
300L
```

The first two lines of your listing should be:

```
0300- 20 DD FB JSR $FBDD
0303- 60 RTS
```

What follows after \$303 is more or less random and does not affect the code we have typed in. When run, this program will jump to the beep routine at \$FBDD. At the end of that routine is an RTS that will return us to our program at \$303. The RTS there will then do a final return from the program back to either the Monitor or BASIC depending on where we call it from.

From the Monitor type in:

```
300G
```

The speaker should beep and you will get the asterisk prompt back. Now go back into BASIC with a <CTRL>B. Type in:

```
CALL 768
```

The speaker should again beep and then give you the BASIC prompt back. CALL 768 should work from Integer or Applesoft.

As long as the programs are not very involved, the Mini-Assembler is handy for writing quick routines. A complete table of routines in the Monitor appears in Appendix D at the end of the book. Try to write your own JSRs to one or more of these routines. You might even try doing several in a row for fun.

Assemblers

Now let's look at the operation of a more typical assembler. This example assumes you're using an assembler similar to the ones mentioned in the introduction. If you have a different assembler that gives you different results, you may have to consult your operating manual for the proper procedures for entering source listings.

Before presenting the listing, I'd like to clarify two commonly used terms in assembly-language programming, *source code* and *object code*. Source code is the English-like text you enter into the assembler. This text has the advantage of being easily readable, and may include whole sentences or paragraphs of comments very similar to REM-type statements found in BASIC. Source code is, however, not directly executable by the 6502. It simply does not understand English-like text. As mentioned earlier, the 6502's preferred (and in fact only acceptable) diet is one- to three-byte chunks of memory in which simple and unambiguous numbers are found.

The assembler takes this text and produces the pure numeric data, called the object code, which is directly executable by the 6502.

Now the listing:

Object Code	Source Code
	1 *****
	2 * AL03-SAMPLE PROGRAM *
	3 *****
	4 *
	5 *
	6 * OBJ \$300
	7 * ORG \$300
	8 BELL EQU \$FBDD
	9 *
0300: 20 DD FB	10 START JSR BELL ; RING BELL
0303: 60	11 END RTS ; RETURN
0304: 66	12 CHK

To the right side of the listing is what is generally called the source code. This is the program, coded using mnemonics and various names or labels for

different parts of your routine. Very few actual addresses or values are used in the source code.²

To the left is the object code. This is what is actually put in memory as the machine language program. The object code is what the computer actually executes; it is obviously rather difficult to understand, at least compared to trying to understand it when you have the advantage of the source code. Being more readily able to understand the coding places greater importance on having the source listing for a given program and explains why your *Apple II Reference Manual* contains a source listing for the Apple Monitor. Such listings were considered necessary in documenting a system when the Apple came out.

However, source listings for Applesoft BASIC, Integer BASIC, and the Disk Operating System (DOS) are much harder to come by and are not directly distributed by Apple Computer Co., Inc. Independently created source listings for DOS and Applesoft BASIC have been prepared by individuals not directly associated with Apple Computer Co., Inc. and are commercially available. The DOS 3.3 source compiled by Randy Hyde is available from Lazer Systems, Inc. An Applesoft BASIC source listing is included in the *Merlin Assembler* from Southwestern Data Systems.

Most assemblers display both the object code and the source code when the ASM (for ASseMble) command is used. Object and source code are, however, usually saved to disk as two separate and distinct files. Initially, let's consider just the source listing.

The first thing to notice is that, just like in BASIC, we again have line numbers. In assembly language, though, the line numbers are solely for use with the program editor, and are not used at all to reference routines. Inserting a line is done with a special editor command, and all following lines are automatically renumbered to accommodate the new line.

Next notice the syntax, or proper ordering of the information. Generally the syntax consists of three basic elements, or *fields*, to each line. These fields are either defined by their position on the line or, more often, by *delimiters*. A delimiter is a character used to separate one field from another. In most assemblers, a space is used. Using this convention, you don't have to tab over to some specific position for each field on the line. Instead you just make sure each field is separated from the adjacent one by a space.

² [CT] Line 12 contains the CHK pseudo-opcode that is provided by the *Merlin Macro Assembler*. CHK instructs the assembler to insert a single byte containing a "checksum" for the entire program. If you are using *Merlin*, you can use the resulting checksum to verify that you have typed in the program correctly. For example, for this particular program you should get a checksum of \$66. If you are not using *Merlin* you should ignore the CHK instruction. In case you are curious, the checksum is computed by performing an exclusive OR on all of the program bytes.

The first field is for a *label* and is optional. Lines 10 and 11, for example, each have a label that applies to that point in the routine. In this case, the label `START` indicates where we first begin the program; `END` is the clever label used for the finish. You may even recognize this program as the one we used to beep the speaker earlier. Some assemblers limit the number of characters used in the label.

As the program becomes more complex, we can do the equivalents of `GOTO` and `GOSUB` by using these labels instead of a line number. You'll notice that to do this, `BELL` has to be defined somewhere in the listing. Since `BELL` does not occur as a label within our own program (lines 10 and 11), it is defined at the beginning using the `EQU` (`EQUals`) statement. The statement reads: "`BELL EQUals $FBDD.`" This way, whenever we use the label `BELL`, the assembler will automatically set up the `JSR` or whatever to the address `$FBDD`.

The second field is the *command field*, which includes the opcode and its operand. In line 10, the `JSR` is the opcode and the operand is `BELL`. Not all opcodes will have an operand.

The third field, to the right, is the *comment field*. Use of the comment field is optional and is reserved for any comments about the listing you might wish to make (for example, `RING BELL`). The semicolon in the source code is used as the delimiter for the comments field. Comments can also be done at the very beginning of the line by using an asterisk as the `REMark` character.³ As in `BASIC`, everything after the asterisk is ignored by the assembler.

Assemblers also have what are sometimes called pseudo opcodes or directives, like `EQU`. Although directives do not translate into 6502 code, they are interpreted by the assembler according to assigned definitions as the object code is assembled.

They are called directives because they direct the assembler to perform a specific function at that point such as store a byte, save a file to disk, etc.

The sample program uses two directives, `OBJ` and `ORG`, on lines 6 and 7 of the source listing. `OBJ` stands for `OBJect` and defines where the object code will be assembled in memory.⁴ `ORG` stands for `ORiGin` and defines the base address to be used when creating the `JSRs`, `JMPs`, and other functions that reference specific addresses within the program. Generally `OBJ` and `ORG` are the same, and for the time being we'll leave it at that. Consult your assembler manual for more specific information on the use of these commands.

³ [CT] A quick tip: If you are using the *Merlin* assembler, you can automatically fill the line with asterisks by hitting `<CTRL>P`. If you type a space and then hit `<CTRL>P` then *Merlin* will insert an asterisk at the beginning and end. See Appendix G for details.

⁴ [CT] The `OBJ` directive is not usually necessary, and it is incompatible with later versions of the *Merlin* assembler (*Merlin Pro* and *Merlin 8/16*). In this book all of the `OBJ` lines have been commented out.

Remember, only the actual program is converted into the object code. The remarks and the EQU, OBJ, and ORG statements are only used in the source code and are never transferred to the object code.

Load/Store Opcodes

One of the most fundamental operations in machine code is transferring the number values between different locations within the computer. You'll recall that in addition to the 64K of actual memory locations, there were registers inside the 6502 itself. These were the Accumulator, the X-Register, and the Y-Register. There are a number of opcodes that will load each of these registers with a particular value and, of course, another set to store these values somewhere in the computer. The table below summarizes these:

	Accumulator	X-Register	Y-Register
Load:	LDA	LDX	LDY
Store:	STA	STX	STY

The first mnemonic, LDA, stands for LoAD Accumulator. LDA is used whenever you wish to put a value into the Accumulator. Conversely, to store that value somewhere, you would execute the STA command, which stands for STore Accumulator. The opcodes for the X-Register and Y-Register are similar and perform the identical function with the associated registers.

Now the question is, how do we control what numbers get put into the register we're concerned with? There are basically two options. The first is to put a specific number there. This is usually indicated in the source listing by preceding the number we want to be loaded with a “#” character.

```

99 LDA #$05 ; LOAD ACC. WITH THE
100 ; VALUE '$05'
```

For instance, in this example, we have loaded the Accumulator with the value 5. How do you think we would load the X-Register or the Y-Register with the value 0?

The other alternative is to load the register with the contents of another memory location. To do this, we just leave off the “#” character.

```

99 LDA $05 ; LOAD ACC. WITH THE
100 ; CONTENTS OF LOC. $05
```

In this case, we are loading the Accumulator with whatever location \$05 happens to be holding at the moment.

These two options are called *addressing modes*. The first example (#\$05) we call the *immediate mode*, because it is not necessary to go to a memory location to get the desired value. The second case we call the *absolute mode*. In this mode,

we put a given value in the register by first going to a specified memory location that holds the value we want.

Putting it All Together

We now have the ability to transfer numbers about in the computer, to jump to other subroutines within the Apple via a JSR, and to return safely to the normal world via an RTS when we're done. In addition, we have an assembler that will allow us easily to generate a source listing for our program, which can also be easily modified. Let's put all this together to write a short program to print some characters on the screen. Appendix E contains two charts (the ASCII table, and the Text Screen Memory Map) that will supply the necessary information to achieve this.

When a character is printed on the screen, what is really happening is that a number value is being stored in the area of memory reserved for the screen display. Change a value there and a character on the screen will change. The Text Screen Memory Map gives the various addresses of each position on the screen. The upper left corner corresponds to location \$400, the lower right to \$7F7.

The ASCII table shows which number values create which screen characters. Suppose we want to print the word APPLE in normal text. The table indicates that we should use the following values:

```
A: $C1
P: $D0
P: $D0
L: $CC
E: $C5
```

If we want the word to appear on the seventh line of the screen, we should load these values into locations \$700 to \$704. To test this, enter the following program using your assembler. If you still don't have one, the Apple Mini-Assembler can be used, although we will soon reach the point where it will not be sufficient for our needs. If you are using the Apple Mini-Assembler, enter only the program itself, ignoring the OBJ and ORG statements. In place of JSR HOME enter JSR \$FC58.

At the beginning of the program, we define where it is to be assembled. Then we define a routine in the Apple called HOME, which is part of the Apple Monitor and is at \$FC58. Whenever this routine is called, the screen is cleared and the cursor put in the upper left corner. This ensures that only the word APPLE will be printed on the screen.

```
1 *****
2 *      AL03-TEST PROGRAM 1      *
3 *****
4 *              OBJ  $300
5              ORG  $300
```

```

        6 HOME EQU $FC58
        7 *
0300: 20 58 FC 8 START JSR HOME ; CLEAR SCREEN
0303: A9 C1 9 LDA #$C1 ; 'A'
0305: 8D 00 07 10 STA $700
0308: A9 D0 11 LDA #$D0 ; 'P'
030A: 8D 01 07 12 STA $701
030D: 8D 02 07 13 STA $702
0310: A9 CC 14 LDA #$CC ; 'L'
0312: 8D 03 07 15 STA $703
0315: A9 C5 16 LDA #$C5 ; 'E'
0317: 8D 04 07 17 STA $704
031A: 60 18 END RTS
031B: 72 19 CHK

```

The routine will begin by doing a JSR to the home routine to clear the screen. Then the Accumulator will be loaded with an immediate \$C1, the value for the letter A. This will then be stored at location \$700 on the screen, which will cause the letter A to be visible on the screen. The next value loaded is for the letter P, and this is stored at \$701 and \$702. It is not necessary to reload the Accumulator, since storing the number does not actually remove it from the Accumulator. The number is just duplicated at the indicated spot. The process continues in this pattern until all five letters have been printed, and then an RTS returns us to normal operation.

Once you have assembled the routine at \$300, try calling it both from the Monitor level with:

```
300G
```

and from BASIC (either one) with:

```
CALL 768
```

You should also change the LDA/STA to the X-Register and Y-Register equivalents to verify that they work in a similar manner.

Conclusion

You now have at your disposal a total of eight opcodes and a familiarity with assemblers. These few opcodes are probably the most often used, and with just these alone you can do quite a number of things. The JSR allows you to make use of all the routines already available in the Monitor. I highly recommend getting *The Apple Monitor Peeled* by W.M. Dougherty, available exclusively from Apple, for more information on using these routines. His book gives a lot of detail on what is available.

In the next chapter we'll look at some more advanced addressing techniques, and how to do counters and loops.

4

Loops and Counters

January 1981

Now we get into not only more mnemonics, but the techniques of using them to accomplish various overall operations. In particular, we'll look at counters and loops in assembly language. In BASIC, the FOR-NEXT loop is one of the more essential parts of many programs, and this is no less true in machine 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 the FOR-NEXT loop. In assembly language, the testing is done by examining flags in the Status Register. These flags indicate the status of the various registers and memory locations. The Status Register is a fourth register of the 6502, one we have not previously mentioned. Before going on with loops and counters it will be necessary to briefly discuss the Status Register and, in addition, binary numbers.

Like the other three registers—the Accumulator, the X-Register, and the Y-Register—the Status Register 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. The one of common experience is that in which we consider only the magnitude of the number. Noticing that 255 is larger than 128 gives us only a very simple form of information—whether a number is either less than, equal to, or greater than another number.

A second way of looking at numbers is in binary form. Base two allows us to see more information in a number and hence is that much more useful. We have already seen how a single byte can be represented either as 0 to 255 or as \$00 to \$FF. In binary the range is 00000000 to 11111111. For instance, 133 (base ten) was represented as \$85. In binary it has the appearance 10000101. In this case, each 1 or 0 represents 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, not harder. Besides, learning base sixteen (hex) wasn't that bad back at the beginning of this book, was it? So let's take a moment to see what this bits and bytes stuff is all about.

Binary Numbers

The Apple is an electronic device and, actually, in many ways, a simple one at that. In most parts of its circuitry, the current is either off or on. That's it. No in-between. Having two possible positions 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 ten. We have a total of ten 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 in hex that, by using 0 through 9 and A through F, we had sixteen possibilities; thus, we were in base sixteen. With the on/off nature of the Apple, we're limited to two possibilities: 0 or 1.

How high can we count in one position? Not very. We start at 0, then go to 1, and that's it. 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. If we use three positions, the lowest number is 100 (representing the quantity four in base ten).

For a given number base, there is a formula for the highest decimal number you can represent with a given number of positions:

$$N = B^P - 1$$

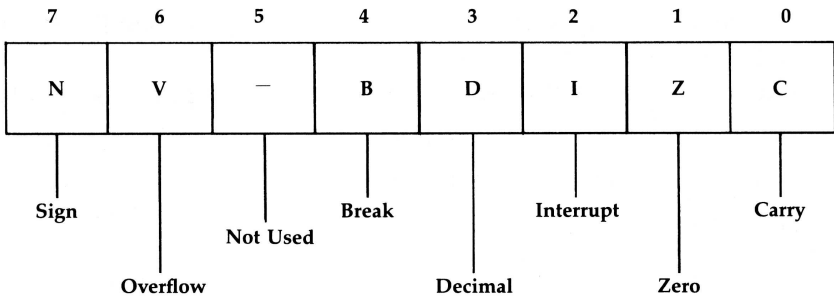
where N is the largest decimal number, B is the number base, and P is the number of positions available.

By using eight positions, we can go up to 11111111, which just happens to equal 255. How handy! This is the same maximum value of our bytes. And, if the truth be known, it's actually the other way around. We use the numbers through 255 because we are using eight bits to make up each byte. Whether each bit is a 0 or a 1 depends on whether the part of the circuit that is responsible for that bit is off or on.

The Status Register

Here at last is our representation of a single byte, made up of eight bits. In particular, the byte we are looking at is the Status Register of the 6502. The important difference between this register and the others is that it is not used to store number values. Instead it indicates various conditions.

The bits of the Status Register are numbered from right to left, 0–7. Each bit in this register indicates the status and/or results of different operations and is called a flag. It is by using this register that we can create counters and loops in our programs. The flag we will be immediately concerned with is bit one, the *zero flag*. In terms of the commands we already know, the zero flag is affected by an LDA, LDX, or LDY.



If the value loaded into the Accumulator, X-Register, 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, \$00. The setting or clearing of each Status Register flag is done automatically by the 6502 after each program step, indicating the results of any particular operation.

Incrementing and Decrementing

To create a *counter* and then a *loop*, we will use the Status Register to tell when a given register or memory location reaches 0. We will also need a way of changing the value of the counter in a regular fashion. In the 6502, this is done by *incrementing* or *decrementing* by one each time, as indicated.

	Accumulator	X-Register	Y-Register	Memory Loc
Increment by 1:	Not available	INX	INY	INC
Decrement by 1:	Not available	DEX	DEY	DEC

The table above shows the mnemonics used to increment or decrement a particular register or memory location.

Note that directly incrementing or decrementing the Accumulator is not possible. The increment/decrement commands affect the zero flag, depending on whether the result of the operation is 0 or not.

The usual syntax for using these commands in an assembly listing is:

```

10 INX
11 INY
12 DEX
13 DEY
14 INC $0600
15 DEC $AA53

```

For the 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 hex of course, usually preceded by the dollar sign.

One thing to mention here is the wrap-around nature of the operations. To understand this, examine the following chart:

Original contents	Increment	Decrement	Z-flag set?	Z
\$05	\$06	\$04	no, no	0, 0
\$0F	\$10	\$0E	no, no	0, 0
\$01	\$02	\$00	no, yes	0, 1
\$FF	\$00	\$FE	yes, no	1, 0
\$00	\$01	\$FF	no, no	0, 0

The effects of incrementing and decrementing different values are shown, along with the effects on the zero flag after the operations. The first case is simple, $5 + 1 = 6$, $5 - 1 = 4$. In both cases, the result is nonzero, so the zero flag is not set. For \$0F, the same holds true. Remember that, in hex, the next 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 one would normally give \$100. However, since a single byte only has a range of \$00 to \$FF, the 1 is ignored, and the value becomes \$00. This sets the zero flag. In the case of decrementing, $\$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 zero flag—is *clear*, that is, not set, for both operations.

Looping with BNE

The only procedure remaining 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, we start with the counter X set at 255. Then the value is printed, decremented, and the process repeated until the counter reaches 0. We

can make the loop execute any number of times by properly setting the initial value of X.

In machine code, the test and GOTO is done with a branch instruction. In this case, the one we'll use first is BNE. BNE stands for Branch Not Equal and is a branch instruction executed when a register is loaded with "a nonzero value." This can happen either directly with something like a LDA # $\$01$ or as the result of an arithmetic operation, such as an INX, DEC, or ADC. Here is the assembly-language equivalent of the BASIC listing:

```

1 *****
2 *      AL04-LOOP PROGRAM 1      *
3 *****
4 *
5 *          OBJ  $300
6          ORG  $300
7 HOME     EQU  $FC58
8 *
9 START    JSR  HOME
10         LDX  #$FF
11 LOOP    STX  $700
12         DEX
13         BNE  LOOP
14 END     RTS

```

And here is the way Apple's disassembler would show it:

```

*300L
0300-  20 58 FC    JSR  $FC58
0303-  A2 FF      LDX  #$FF
0305-  8E 00 07   STX  $0700
0308-  CA         DEX
0309-  D0 FA      BNE  $0305
030B-  60         RTS

```

In this program, we first do a JSR to the clear screen routine in the Monitor that we used in chapter three. Then we load the X-Register with a starting value of $\$FF$. Now we start the loop. Storing the X-Register at $\$700$ will make the loop's action visible as a character on the screen for each pass through the loop. Next, 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. People will also refer to the execution of a branch instruction by saying that the branch is ignored or taken depending on whether program flow falls through the branch instruction or goes to the new address indicated by the branch instruction.

Try entering this now, and also notice how fast the program runs. You probably weren't able to see very much, but all 255 values were put to the screen. The inverse A that's left on the screen is how a $\$01$ at $\$700$ appears. ($\$00$ doesn't

get printed—why?) To verify that each pass is being executed, replace the STX \$700 in the source listing with a JSR \$FBDD. If you don't want to hear 255 beeps, try changing the initial value of the X-Register in line 10. As before, you should be able to call this program from the Monitor with a 300G, or from BASIC with a CALL 768.

You may also wish to try the equivalent version of the program using the Y-Register or a memory location as the counter. I would suggest trying to write a program using INC, INX, or INY to drive the counter as a practice program.

5

Loops, Branches, COUT, and Paddles

February 1981

Looping with BEQ

In the previous chapter we started into the various techniques of creating and using counters and loops in assembly language. To accomplish the loop, we used the value in one of the registers as a counter and the branch instruction that tests for the presence of a nonzero number in the register to actually do the looping. Recall that this evaluation of zero/nonzero is done via the zero bit, or flag, of the Status Register of the 6502.

The complement of the BNE instruction is something called BEQ, which obscurely enough stands for Branch EQual. It operates in just the opposite fashion from BNE; that is, it branches only when the register or memory location reaches a value of 0.

For example, consider this BASIC listing:

```
10 HOME
20 X = 255
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 0. If it is, the branch instruction is carried out and the program ends. In assembly language, this program would be the equivalent:

```
1 *****
2 *      AL05-LOOP PROGRAM 2      *
3 *****
4 *
5 *          OBJ  $300
6 *          ORG  $300
7 HOME      EQU  $FC58
8 *
9 START     JSR  HOME
10          LDX  #$FF
11 LOOP     STX  $700
12          DEX
13          BEQ  END
```

```

14          JMP  LOOP
14 END      RTS

```

Notice that this program requires the addition of a new instruction to our repertoire: the JMP command. This is analogous to a GOTO in BASIC, and in this program will cause program execution to jump to the routine starting at LOOP each time. Only when the X-Register reaches 0 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
0300- 20 58 FC   JSR  $FC58
0303- A2 FF     LDX  #$FF
0305- 8E 00 07  STX  $0700
0308- CA       DEX
0309- F0 03     BEQ  $030E
030B- 4C 05 03  JMP  $0305
030E- 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 intelligible translation to the right is Apple’s interpretation of this data.

Branch Offsets and Reverse Branches

Notice that the JMPs and JSRs are immediately followed by the addresses (reversed) that they are to jump to, such as in the first JSR as \$300.

However, branch instructions are handled a little differently. The \$03 is an offset that tells the 6502 to jump three bytes past the next instruction.

Since the next instruction is at \$30B, the 6502 will branch to \$30E, thus skipping the JMP command and going directly to the RTS, which terminates the routine.

Branches can also be done in the reverse direction. Here is a rather inefficient, but illustrative example:

```

1 *****
2 *      AL05-LOOP PROGRAM 2A      *
3 *****
4 *
5 *          OBJ  $300
6 *          ORG  $300
7 HOME      EQU  $FC58
8 *
9 START    JSR  HOME
10         JMP  SETX
11 END      RTS

```

```

12 *
10 SETX   LDX  #$FF
11 LOOP   STX  $700
12        DEX
13        BEQ  END
14        JMP  LOOP

```

The Monitor listing for this would be:

```

*300L
0300-  20 58 FC   JSR  $FC58
0303-  4C 07 03   JMP  $0307
0306-  60        RTS
0307-  A2 FF     LDX  #$FF
0309-  8E 00 07   STX  $0700
030C-  CA        DEX
030D-  F0 F7     BEQ  $0306
030F-  4C 09 03   JMP  $0309

```

In this example, the branch, if taken, will cause the program to move back up through the listing. To indicate this branch in the opposite direction, the high bit is set. This is the same technique that is often used to show negative numbers in assembly-language programs. Please note that it is not just a matter of setting the high bit. If that were the case, the value following the BEQ command might be expected to be \$89. (The address of the next instruction (\$30C) minus where we want to go to (\$303) equals \$09. Then with the high bit on, we have \$89.)

This is almost correct. The actual value is arrived at by subtracting the branch distance from \$100. Thus \$100 minus \$09 equals \$F7. This is so that the destination address can still be arrived at through addition. Notice that \$30C + \$F7 = \$403. It is then very easy for the 6502 to correct this back one page to \$303.

If all this seems a bit confusing, try not to let it bother you. In actual practice, there is not much reason to be concerned about the way in which the offset byte is determined since your assembler will determine the proper values for you when assembling code, and Apple's disassembler, as well as many others, including *Sourceror*, will give the destination address when reading other code.

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 you understand exactly what happens at each step, and how it relates to the rest of the program. If you're not sure, go back over it until that proverbial light comes on!

Screen Output Using COUT

As the X-Register is incremented in this program, we'll stuff the value into \$700 so we can see something on the screen as the counter advances.

Now you may remark from your experience in chapter four that although this program is pleasantly simple in its logic, it is not much fun to watch on the screen because it runs so quickly.

To solve this, we will start to make more extensive use of the routines already present in the Monitor to do certain tasks and thus make our programming requirements a little simpler. Referring to the Monitor subroutines in Appendix D, it happens that the first routine listed is something called COUT. This is the routine that actually sends a character we want output to whatever device(s) may currently be in use. Most of the time this just goes directly to the next routine listed, COUT1 (clever with the names, aren't they?), which specifically handles the screen output. What this means for us is that anytime we want to output a character, we don't have to write our own routines to worry about all the in-depth details about the screen (cursor position, screen size, whether it's time to scroll)—we just load the Accumulator with the ASCII value for the character we want to print and then do a JSR \$FDED!

Now comes some programming technique. We would like to have the counter value in the Accumulator so we can print it via COUT, but unfortunately our increment/decrement commands only work for the X-Register, the Y-Register, and given memory locations. To solve this, we'll have to expand our listing a little. 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 \$06 to \$09. These are not used by either Integer, Applesoft, DOS, or the Monitor. This is important for avoiding conflicts with the Apple's normal activities while running your own programs.

And now our revised listing:

```

1 *****
2 *      AL05-LOOP PROGRAM 2B      *
3 *****
4 *
5 *          OBJ  $300
6          ORG  $300
7 CTR      EQU  $06
8 HOME    EQU  $FC58
9 COUT    EQU  $FDED
10 *
11 START   JSR  HOME
12         LDA  #$FF
13         STA  CTR
14 LOOP    LDA  CTR
15         JSR  COUT
16         DEC  CTR
17         BEQ  END
18         JMP  LOOP
19 END     RTS

```

Apple's "L" command will give this after you've assembled it in memory:

```
*300L
0300- 20 58 FC   JSR   $FC58
0303- A9 FF     LDA   #$FF
0305- 85 06     STA   $06
0307- A5 06     LDA   $06
0309- 20 ED FD   JSR   $FDED
030C- C6 06     DEC   $06
030E- F0 03     BEQ   $0313
0310- 4C 07 03   JMP   $0307
0313- 60       RTS
```

A call to this routine via our usual 300G from the Monitor, or a CALL 768 from BASIC, should clear the screen, then print all the available characters on your Apple in all three display modes (normal, flashing, and inverse). The beep you hear is the <CTRL>G (bell) being *printed* to the screen via COUT. The invisible control characters account for the blank region between the two main segments of output characters. You will also see some characters that are not normally generated by the Apple, such as underscore, reverse slash, and the left square bracket (`_`, `\`, `[`).

The alphabet is backward because we started at the highest value and worked our way down. From chapter four, though, you'll remember that when a byte is incremented by one from \$FF, the result *wraps around* back to \$00. This will produce an action testable by a BEQ. Using this wrap-around effect of the increment command, we can rewrite the program to be a little more conventional like so:

```
1 *****
2 *      AL05-LOOP PROGRAM 3      *
3 *****
4 *
5 *          OBJ  $300
6          ORG  $300
7 CTR      EQU  $06
8 HOME    EQU  $FC58
9 COUT    EQU  $FDED
10 *
11 START   JSR  HOME
12        LDA  #$00
13        STA  CTR
14 LOOP    LDA  CTR
15        JSR  COUT
16        INC  CTR
17        BEQ  END
18        JMP  LOOP
19 END     RTS
```

With the Apple showing:

```
*300L
0300- 20 58 FC   JSR   $FC58
0303- A9 00     LDA   #$00
0305- 85 06     STA   $06
0307- A5 06     LDA   $06
0309- 20 ED FD   JSR   $FDED
030C- E6 06     INC   $06
030E- F0 03     BEQ   $0313
0310- 4C 07 03   JMP   $0307
0313- 60        RTS
```

A call to this routine should now print out the characters in a more familiar manner. At last our programs are starting to do something interesting! It gets better!

Reading a Game Paddle

Let's try reading a game paddle and use what we get back to print something to the screen! Granted, I'm not any more sure than you are what good this might be, but it's guaranteed to be a new program in your library!

The PREAD subroutine in Appendix D indicates that a paddle can be read by loading the X-Register with the value for the number of the paddle you wish to read, followed by a JSR \$FB1E. When the routine returns, the value of the paddle will be in the Y-Register. All we have to do then is grab this value, stuff it in the Accumulator, and then do our JSR COUT.

```
1 *****
2 *      AL05-PADDLE PROGRAM 1      *
3 *****
4 *
5 *          OBJ  $300
6 *          ORG  $300
7 TEMP      EQU  $06
8 PREAD     EQU  $FB1E
9 HOME      EQU  $FC58
10 COUT     EQU  $FDED
11 *
12 START    JSR  HOME
13          LDX  #$00
14 LOOP     JSR  PREAD
15          STY  TEMP
16          LDA  TEMP
17          JSR  COUT
18          JMP  LOOP
19 * INFINITE LOOP
```

You should get this in memory:

```
*300L
0300- 20 58 FC   JSR   $FC58
```

```

0303-  A2 00      LDX  #$00
0305-  20 1E FB   JSR  $FB1E
0308-  84 06      STY  $06
030A-  A5 06      LDA  $06
030C-  20 ED FD   JSR  $FDED
030F-  4C 05 03   JMP  $0305

```

This routine when called will quickly fill up the screen and then change the stream of characters output as you turn paddle 0. Since we have no test for an end, RESET is the only way to stop this infinite loop.

Depending on your propensity toward being hypnotized, you may lose touch with the world for indefinite periods of time while running this program. At the inverse and flashing end, it's also remarkably good at stimulating migraine headaches in record time. By carefully controlling the paddle, you can also observe some interesting bits of ASCII trivia. For example, after the inverse and flashing range, you should be able to stop the flow by moving into the control character range. With sufficient dexterity, you should be able to lock onto the persistent beep of the bell (<CTRL>G).

Shortly after this point, the screen will zip into motion when you hit the line feed character (<CTRL>J) and, of course, also at <CTRL>M (carriage return). What fun, eh! When normal character output returns as you pass the halfway point, you can delight in various patterns of screen filling. Why, you may even wish to try writing your name by deft control of the paddle—child's play!

Paddle Program Problems

Returning to reality here, it is worth mentioning that some problems in accuracy can arise from repeatedly reading the paddle so quickly. The analog circuits don't have time to return to 0, and various problems creep in.

Also, we have been a bit negligent in looking out for conflicting use of the registers by the various routines we are calling. There is often no assurance that the register you're using for your own routine won't be clobbered by the Monitor routine you use. In the case of the paddle and output routines, you'll note they did mention how the X-Register, the Y-Register, and the Accumulator were affected by each of the routines.

For the record, here is a reasonable facsimile of our program in Applesoft:

```

10 HOME
20 T = PDL(0)
30 PRINT CHR$(T);
40 GOTO 20

```

It is also worth mentioning that our assembly-language version takes eighteen bytes, while the Applesoft one takes thirty-eight, not counting space used by the variable T.

Execution speed may seem to be similar, but this is because of the printing of the characters to the screen. In most cases, machine execution would be orders of magnitude faster.

Transfer Commands

In our program, we have to go through a rather inelegant way of transferring the value from the Y-Register to the Accumulator, using a temporary storage byte. Fortunately, there is an easier way. There are four commands for transferring contents of the X-Register or the Y-Register to and from the Accumulator. They are as follows:

TXA: Transfers contents of X-Register to Accumulator.

TYA: Transfers contents of Y-Register to Accumulator.

TAX: Transfers contents of Accumulator to X-Register.

TAY: Transfers contents of Accumulator to Y-Register.

Each of these actions conditions the zero flag upon execution, so it is possible to test what has been transferred. There is no command to transfer directly between the X-Register and the Y-Register.

This gives us an even shorter program:

```

1 *****
2 *   AL05-PADDLE PROGRAM 1A   *
3 *****
4 *
5 *       OBJ   $300
6 *       ORG   $300
7 PREAD  EQU   $FB1E
8 HOME   EQU   $FC58
9 COUT   EQU   $FDED
10 *
11 START  JSR   HOME
12        LDX  #$00
13 LOOP   JSR   PREAD
14        TYA
15        JSR  COUT
16        JMP  LOOP
17 * INFINITE LOOP

```

Now it's only fifteen bytes long!

```

*300L
0300- 20 58 FC   JSR   $FC58
0303- A2 00     LDX   #$00
0305- 20 1E FB   JSR   $FB1E
0308- 98        TYA
0309- 20 ED FD   JSR   $FDED
030C- 4C 05 03   JMP   $0305

```

With twenty commands at your disposal, you now know just over a third of the total vocabulary of the language. Soon, you'll be dangerous!

A Note about BRUN and COUT

If you try to BRUN ALØ5.LOOP2B, rather than use a CALL 768 or 3ØØG, strange things will happen. This is because DOS interferes with any binary program which uses input or output routines when such a program is BRUN, rather than called from the Monitor or BASIC. This is because DOS is always watching COUT for DOS commands, such as PRINT D\$;"CATALOG". When you BRUN a file you are essentially in a DOS subroutine, and further use of COUT makes DOS more or less forget where to return to when everything is completed. There are two solutions to this problem. The first is trivial—don't BRUN files that use COUT. Instead, BLOAD the file and then call the routine in the usual way.

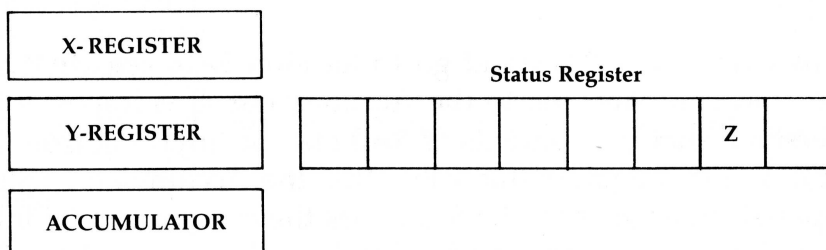
If, however, you insist on BRUNing a file, the other choice is to exit via the warm-reentry vector \$3DØ. A jump to this address replaces the final RTS in any program you wish to BRUN. For example, replacing line 19 in LOOP PROGRAM 2B with JMP \$3DØ will allow you to BRUN the file with no problems. Please keep this in mind when attempting to BRUN any other listings throughout this book.

I/O Using Monitor and Keyboards

March 1981

Comparisons; Reading the Keyboard

Now we're getting to where we can actually do some interesting things with what we know so far. The basic ideas you should be comfortable with at this point are fairly simple. The 6502 microprocessor is our main operational unit. There are three main registers: the Accumulator, the X-Register, and the Y-Register. Also present is the Status Register, which holds a number of one-bit flags to indicate various conditions. So far, the only one we've considered is the Z-flag, for indicating whether a zero or nonzero number is present in one of the other three registers.



6502 Model

Programs are executed by the 6502 scanning through memory. Addresses in memory are analogous to line numbers in BASIC. A JSR \$FC58 in assembly language is just as valid as a GOSUB 1000 in BASIC. In using an assembler, we can give names to routines at given addresses and make things that much simpler by saying JSR HOME, when HOME has been defined as \$FC58.

In chapter five, we used testing commands like BEQ and BNE to create simple loops. We used the X-Register and the Y-Register as counters and incremented or decremented by one for each cycle of the loop.

Now let's expand our repertoire of commands by adding some new ones and, in the process, add some flexibility to what we can do with loops and tests in general.

In our previous programs we relied on our counters reaching 0 and testing via the Z-flag to take appropriate action. Suppose, however, that we wish to test for a value other than 0. This is done using two new ideas.

Compare Commands and Carry Flag

The first is the compare command, the mnemonic for which is `CMP`. This 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:

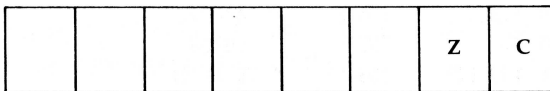
```
CMP #$A0
```

This would be read, “Compare Accumulator with an immediate `A0`.” This would tell the 6502 to compare the Accumulator to the specific value `A0`. On the other hand, we may want to compare the Accumulator with the contents of given memory location. This would be indicated by:

```
CMP $A0
```

In this case, the 6502 would go to location `A0`, see what was there, and compare that to the Accumulator. It is important to understand that the contents of `A0` may be anything from `00` to `FF`, and it is against this value that the Accumulator will be compared. In each case, the 6502 does the comparison by internally subtracting the specified value from the Accumulator. The Accumulator remains unchanged, however, and the result of the comparison is reflected elsewhere.

The second important idea is that of the *carry flag*. The carry flag enables us to determine the result of the comparison. Right next to the Z-flag in the Status Register is the bit called the *carry*.



The carry is used during addition and subtraction by the 6502. In our case, since the compare operation involves subtraction, the carry flag can be used to test the result. You do this with two new branch commands, `BCC` and `BCS`. `BCC` stands for Branch Carry Clear. If the Accumulator is less than the value compared against, `BCC` will branch appropriately. `BCS` stands for Branch Carry Set and is taken whenever the Accumulator is equal to or greater than the value used. This means that we can now not only test for specific values but also test for ranges. Try this example.

```

1 *****
2 *      AL06-PADDLE PROGRAM 2A      *
3 *****
4 *
5 *          OBJ  $300
6 *          ORG  $300
7 *
8 PREAD     EQU  $FB1E
9 HOME      EQU  $FC58
10 COUT     EQU  $FDED
11 *
12 START    JSR  HOME
13          LDX  #$00
14 LOOP     JSR  PREAD
15          TYA
16          CMP  #$C1      ; CMP TO ASCII FOR "A"
17          BCC  LOOP      ; TRY AGAIN IF LESS THAN
18          CMP  #$DB      ; CMP TO ASCII FOR "["="Z"+1
19          BCS  LOOP
20          JSR  COUT
21          JMP  LOOP
22 * INFINITE LOOP

```

When assembled and listed from memory, it should look like this:

```

*300L
0300- 20 58 FC   JSR  $FC58
0303- A2 00     LDX  #$00
0305- 20 1E FB   JSR  $FB1E
0308- 98        TYA
0309- C9 C1     CMP  #$C1
030B- 90 F8     BCC  $0305
030D- C9 DB     CMP  #$DB
030F- B0 F4     BCS  $0305
0311- 20 ED FD   JSR  $FDED
0314- 4C 05 03   JMP  $0305

```

Let's step through the program. After the JSR to the clear screen routine, we load X with 0 in preparation for reading a paddle. The #\$00 will tell the routine that we wish to read paddle 0. After the read, the answer is returned in the Y-Register, which we transfer to the Accumulator with a TYA. It is at this point that we use our test section. If the Accumulator is less than the ASCII value for the letter A, we avoid the printout by going back to LOOP. I have used the ASCII value for A plus \$80 so that we get normal output on the screen. If we test for \$41 instead, flashing characters will be output to the screen.

The next comparison is for the ASCII value for the character “[”. This comparison assures that the BCS will catch all values higher than the one for Z. The first table in Appendix E is useful in seeing where these numbers come from.¹

Only numbers from \$C1 to \$DA will make it through to be printed out using COUT (\$FDED).

Again the loop is infinite, so RESET is required to exit.

The X-Register and Y-Register can also be compared in a similar manner by codes CPX and CPY. Can you rewrite this program to use CPY instead of CMP?

BEQ and BNE are also still usable after a compare operation. Here's a summary:

Command	Action
CMP	Compares Accumulator to something
CPX	Compares X-Register
CPY	Compares Y-Register
BCC	Branch if register < value
BEQ	Branch if register = value
BNE	Branch if register <> value
BCS	Branch if register >= value

¹ ASCII (for American Standard Code for Information Interchange) is a coding scheme for transmitting text. It is also used in the Apple for encoding text in memory, screen display, disk files, printer output, and many other areas. Appendix E gives a chart of all the characters and their ASCII values. One important note: it is possible to encode all the alphabetic characters (upper and lowercase), numerics, special symbols, and control codes using only 128 characters. This means that ASCII is considered a 7-bit code. This means that all the information required to determine which character has been sent is contained in bits 0–6 of the byte. Thus \$8A is reasonably equivalent to \$0A as far as its ASCII interpretation is concerned. The matter of the high bit being set or clear can create considerable confusion when it is not made clear what the computer expects.

Generally the Apple operates internally with the high bit set on all characters. That is to say, characters retrieved from the keyboard via \$C000 and characters stored in the screen area of memory and on disk all usually have the high bit set (i.e. a value equal to or greater than \$80). This is also the way Applesoft stores data within program lines. To keep you on your toes, though, Apple printer cards usually do not support having the high bit set when sending output to a printer, and strings within a program (such as A\$ = "CAT") also have the high bit clear. Also, when using COUT (the Monitor text output routine), the high bit should be set (always load the Accumulator with values greater than \$80) before calling COUT.

I wish I could say it was all easier than that, but then again if it were all that easy, you wouldn't have to have bought this book, and then where would I be?

Using Monitor Programs for I/O Routines

As you may have noticed, I enjoy using the paddles as input devices. This is because they're an easy way of sending values from \$00 to \$FF into the system in a very smooth and natural way. We can get similar data from the keyboard, though. There the advantage is that we can jump from one value to another with no transition between the two values.

A good part of many formal assembly-language courses deals with system I/O—that is, getting data in and out via different devices. Writing such things as printer drivers, disk or tape access routines, hardware interface software, etc., are the areas that hardcore programmers spend their youths mastering. Using the Monitor routines on the Apple simplifies this for us greatly because we don't have to do a lot of I/O details. You've already shown this by using the paddles (\$FB1E) for input and the screen (\$FDED) for output without having to know anything about how the actual operation is carried out. The keyboard is even easier.

I mentioned earlier that the address range from \$C000 to \$FFFF is devoted to hardware—these memory ranges cannot be altered by running programs. (I'm ignoring the RAM cards for the time being.) The range from \$D000 to \$FFFF is used by ROM routines that we've been calling. The range from \$C000 to \$CFFF is assigned to I/O devices. Typically the second digit (or maybe I should call it a hexit) from the left gives us the slot number of the device. For instance, if you have a printer in slot one, listing the code at \$C100 will reveal the machine language code on ROM of the card that makes it work. At \$C600 you'll probably find the code that makes the disk drive in slot six 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 you type in:

```
*C030
```

from the Monitor, in addition to getting some random value displayed, 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. (See page 6 of the 1981 *Apple II Reference Manual*.) In assembly language you would usually load a register with the contents of \$C000, such as:

```
LDA $C000
```

Reading Data from the Keyboard

Because it is difficult to read the keyboard at exactly the instant someone has pressed the key, the keyboard is designed to hold the last key pressed until either another key is pressed or until you clear the *strobe*, as it's called, by accessing an alternate memory location, $\$C010$. The strobe is wired to clear any characters off the keyboard that may be hanging around for any number of various reasons. When you check for a character, you don't want to pick one up that someone inadvertently entered prior to your enquiry (perhaps by nervously drumming their fingers across the keyboard while waiting for one of Apple's lightning-like disk accesses!). It is also always a good idea to clear the keyboard when you're done with it, otherwise you may similarly have the key pressed for your input still hanging around for whatever reads the keyboard next, such as an input statement in BASIC. The strobe is cleared by *either* a read or a write operation. It is the mere access to $\$C010$ in any manner that accomplishes the clear. Thus a LDA $\$C010$ would work just as well as a STA $\$C010$.² 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 a $\$0$ in $\$C000$. Perhaps, but that's not the way they do it. Instead, they add $\$80$ to whatever the ASCII value is of the key you pressed. If a value less than $\$80$ is at $\$C000$, it means a key has not been pressed.

So, to illustrate this (and I admit it got a little involved for my tastes), let's look at some sample programs to read data from the keyboard.

```

1 *****
2 *   AL06-KEYBOARD PROGRAM 1A   *
3 *****
4 *
5 *           OBJ   $300
6 *           ORG   $300
7 *
8 KYBD       EQU   $C000
9 STROBE     EQU   $C010
10 COUT      EQU   $FDED

```

² Having now just said that read and write operations are essentially equivalent for clearing the strobe, let me cover myself enough to say that there is one slight difference. A write operation actually accesses the location twice, whereas a read operation only accesses once. Most of the time this doesn't make any difference. Since most people can't type at 100,000 characters per second, it's hard to get a character in between the two clear operations. However, there are now available for the Apple *keyboard buffers* which will store a whole string of characters entered by the user, instead of the usual one normally used for the keyboard. As each character is read in, it is taken out of the buffer by clearing the strobe. You guessed it! A write operation—such as a STA $\$C010$ or a POKE -16368,0—will clear two characters out of the buffer: the one you just read *and* the next one in line. Therefore, it is generally good practice to clear the strobe with a read operation, such as a LDA $\$C010$, X = PEEK -16368, or the like. Like I said, if it were too easy...

```

11 HOME      EQU  $FC58
12 *
13 START     JSR  HOME
14 LOOP      LDA  KYBD
15           CMP  #$80
16           BCC  LOOP
17           JSR  COUT
18           JMP  LOOP
19 * INFINITE LOOP

```

Once entered, this should disassemble as:

```

*300L
0300- 20 58 FC   JSR  $FC58
0303- AD 00 C0   LDA  $C000
0306- C9 80     CMP  #$80
0308- 90 F9     BCC  $0303
030A- 20 ED FD   JSR  $FDED
030D- 4C 03 03   JMP  $0303

```

Trying this program, you should notice that the program runs on, printing the same character until you press another key. That's because we never cleared that strobe you thought I was rambling on about. Once the key press gets on the board, it's never cleared until it is replaced by a new key.

A better program is:

```

1 *****
2 *   AL06-KEYBOARD PROGRAM 1B   *
3 *****
4 *
5 *           OBJ  $300
6           ORG  $300
7 *
8 KYBD      EQU  $C000
9 STROBE    EQU  $C010
10 COUT     EQU  $FDED
11 HOME     EQU  $FC58
12 *
13 START    JSR  HOME
14 LOOP     LDA  KYBD
15         CMP  #$80
16         BCC  LOOP
17         STA  STROBE
18         JSR  COUT
19         JMP  LOOP
20 * INFINITE LOOP

```

which lists as:

```

*300L
0300- 20 58 FC   JSR  $FC58
0303- AD 00 C0   LDA  $C000
0306- C9 80     CMP  #$80
0308- 90 F9     BCC  $0303

```

```

030A- 8D 10 C0 STA $C010
030D- 20 ED FD JSR $FDED
0310- 4C 03 03 JMP $0303

```

This should work better. Here we clear the keyboard whenever we've gotten a character and printed it. Why not clear it right after the read on line 15? If we did that, we'd be lucky to catch a glimpse of the character at \$C000 as the user pressed the key. As it is, we can probably get away with it because of the speed of the loop. But if we had to go away to another routine for a while, or otherwise delay getting back to the LDA \$C000, we'd probably miss it.

You should also type in enough to wrap around onto the next line, and also try the arrow keys and <RETURN>. You may think this all performs as expected (with the exception of the missing cursor), but this all 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 Monitor rather than worrying about the details yourself.

Also, please notice how the STA was chosen because we didn't want to lose the contents of the Accumulator in doing the access. This information concerns technique more than actual commands, but is worth mentioning if you're going to get along with your Apple successfully.

On page 130 of the 1981 *Apple II Reference Manual* you'll find a listing of the soft-switches and other goodies at \$C000 to C0FF. These can be very useful in having your Apple relate to the outside world.

You may wish to experiment with these. Also don't forget about all the routines listed in Appendix D. These are also fun to experiment with and are provided to encourage you to write short programs just to test your wings. As I've mentioned before, they're also useful in saving you the trouble of writing your own I/O and other more involved routines.

Addressing Modes

April 1981

Let's look at the various addressing modes used in assembly-language programming. This concept is rather fundamental in programming and you may justifiably wonder why we have not covered it sooner. Well, as it happens, we have; I just didn't call it by name at the time. In chapter one we laid out the basic structure of sixty-five thousand individual memory locations. Since then, we've worked most of our magic by simply manipulating the contents of those locations.

Flexibility in the ways in which you can address these locations is the key to even greater power in your own programs.

Consider this chart of the addressing modes available on the 6502:

Addressing Modes	Example	Hex Bytes
Immediate	LDA #\$A0	A9 A0
Absolute	LDA \$7FA	AD FA 07
Zero Page	LDA \$80	A5 80
Implicit/Implied	TAY	A8
Relative	BCC \$3360	90 0F
Indexed	LDA \$200,X	BD 00 02
Indirect Indexed	LDA (\$80),Y	B1 80
Indexed Indirect	LDA (\$80,X)	A1 80

In looking at the examples, you should find all but the last three very familiar. We have used each of them in previous programs.

Immediate mode was used to load a register with a specific value. In most assemblers, this is indicated by the use of the number sign (#) preceding the value to be loaded. This contrasts with 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. *Zero page* is just a variation on the absolute mode. The main difference is the number of bytes used in the coding. It takes three in the general case; in zero page, only two are required.

Implicit, or *implied*, is certainly the most compact instruction in that only one byte is used. 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.

Relative addressing is done in relation to where the first byte of the instruction itself is found. Although the example interprets 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 too was covered previously.

With these addressing modes, we can create quite a variety of programs. The problem with these modes is that the programs are rather inflexible with data from the outside world, such as those in input routines, and also when doing things like accessing tables and large blocks of data.

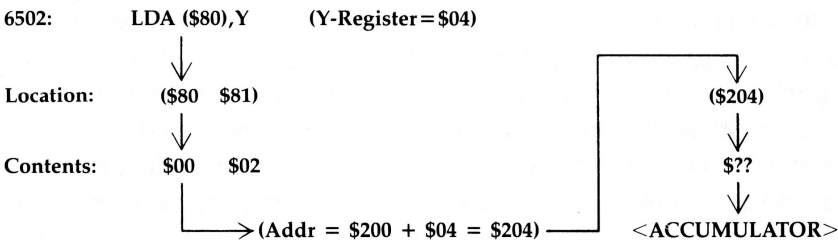
Indexed Addressing

To access such data, we introduce the new idea of *indexed addressing*. In the pure form, the contents of the X-Register or Y-Register are added to the address given in the instruction to determine the final address. In the example given, if the X-Register holds a \$0, the Accumulator will be loaded with the contents of location \$200. If, instead, the X-Register holds a \$04, then location \$204 will be accessed. The usefulness in accessing tables and the like should be obvious.

The problem that arises here occurs when you want to access a table that grows or shrinks dynamically as the data within it changes. Another problem occurs when the table grows larger than 256 bytes. Because the maximum offset possible using the X-Register or Y-Register is 255, we would normally be out of luck.

The solution to the byte limit is to use the *indirect indexed* mode. Indirect indexed is really an elegant method. First, the 6502 goes to the given zero-page location (the base address must be on page zero). In the example, it would go to \$80 and \$81 to get the low-order and high-order bytes of the address stored there. Then it adds the value of the Y-Register to that address.

INDIRECT INDEXED ADDRESSING



Oftimes, these two-byte zero-page address pairs are called *pointers*, and you will hear them referred to in dealing with various programs on the Apple. In fact, by looking at pages 140 to 141 of the *Applesoft II BASIC Programming Reference Manual*, you will observe quite a number of these byte pairs used by Applesoft to keep track of all sorts of continually changing things, like where the program is, the locations of strings and other variables, and many nifty items.

If we wanted to simulate the LDA \$200,X command with the indirect mode, we would first store a #\$00 in \$80 and a #\$02 in \$81, with 00 and 02 being the low-order and high-order bytes of the address \$200. Then we'd use the command LDA (\$80),Y.

A much better (but unfortunately rarely used) term is *post-indexing*, referring to the fact that the index is added *after* the base address is determined.

Sometimes X and Y Aren't Interchangeable

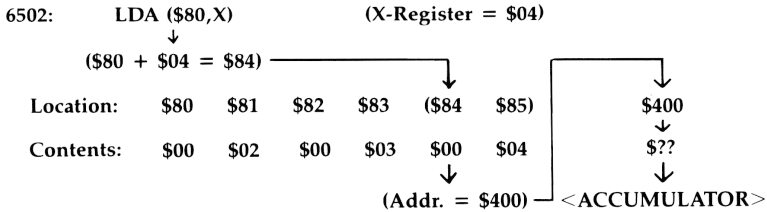
You may have noticed that I used the X-Register in one case and the Y-Register in the other. It turns out that the X-Register and the Y-Register cannot always be used interchangeably. The difference shows up depending on which addressing mode and what actual command you are using (LDA, STX, or others). As it happens, indirect indexed addressing can only be done using the Y-Register.

To know which addressing modes can be used with a given command, you can refer to either of two appendices provided at the back of this book. Appendix B is rather like a dictionary of all the possible 6502 commands and devotes several paragraphs to each command. Appendix C, on the other hand, is a more condensed form of the first appendix and may make it easier to compare available modes between a variety of commands.

I highly recommend making frequent use of Appendix B while you are learning assembly-language programming. It is essentially your toolbox of available commands for solving a particular programming problem. Whenever you try to write a particular routine and aren't sure just how to approach it, skim through this section of all possible commands and see if any particular command inspires a possible approach. Granted, this is likely to happen more when you're working on rather simple goals such as moving a byte from here to there, but even the largest programs are made up of just such simple steps as that.

The last addressing mode, *indexed indirect*, is probably the most unusual. In this case, the contents of the X-Register (the Y-Register cannot be used for this mode) are added to the base address before going to get the contents. In a case similar to the other one, if the X-Register held \$0, an 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!

INDEXED INDIRECT ADDRESSING



Usually, then, the X-Register is loaded with multiples of two to access a series of continuous pointers in page zero. This is also called *pre-indexing* since the index is added to the zero-page location before determining the base address.

Storing Pure Data

Before we can put all this new information to work, we now need to answer one more question. How do you store just pure data within a program? All the commands we've covered so far are actual commands for the 6502. There is no data command as such. What are available, though, are the Assembler directives of your particular assembler. These will vary from one assembler to another, so you'll have to consult your own manual to see how your assembler operates.

In general, the theory is to define a block of one or more bytes of data and then to skip over that block with a branch or jump instruction when executing your program. Usually, data can be entered either as hex bytes or as the ASCII characters you wish to use. In the second case, the assembler will automatically translate the ASCII characters into the proper hex numbers.

Most assemblers have hex command for directly entering the hex bytes of a data table. The Apple *DOS Tool Kit* assembler is one exception. It does not have the HEX command (nor many others) and you must use the DFB ("define byte") command. Using it, line 20 of the following listing should read: 20 DATA DFB \$C1, \$D0, \$D0, \$CC, \$C5. A sample program using the indexed address mode is given here:

```

1 *****
2 *   AL07-SAMPLE DATA PROGRAM   *
3 *****
4 *
5 *       OBJ   $300
6 *       ORG   $300
7 *
8 COUT   EQU   $FDED
9 *
10 START  LDX   #$00

```

```

11 LOOP      LDA  DATA,X
12           JSR  COUT
13           INX
14           CPX  #$05
15           BCC  LOOP
16           LDA  #$8D
17           JSR  COUT
18 EXIT      RTS
19 *
20 DATA     HEX  C1D0D0CCC5
21 *
22 * DATA = 'APPLE'

```

When looked at in memory, it should appear like this:

```

*300L
0300- A2 00      LDX  #$00
0302- BD 13 03   LDA  $0313,X
0305- 20 ED FD   JSR  $FDED
0308- E8         INX
0309- E0 05      CPX  #$05
030B- 90 F5      BCC  $0302
030D- A9 8D      LDA  #$8D
030F- 20 ED FD   JSR  $FDED
0312- 60         RTS
0313- C1 D0      CMP  ($D0,X)
0315- D0 CC      BNE  $02E3
0317- C5 00      CMP  $00

```

This program is an improved version of the one we did earlier to print the word APPLE on the screen. It uses the indexed address mode to scan through the data table to print the word APPLE. Notice that data tables may be wildly interpreted to the screen when disassembling. This is because the Apple has no way of knowing what part of the listing is data and tries to list data as a usual assembly-language program.

Basically, the idea of the program is to loop through, getting successive items from the data table using the offset of the X-Register. When the X-Register reaches 05 (the number of items in the table), we are finished printing. After printing, we terminate with a carriage return. Remember that in assembly language we must usually do everything ourselves. This means we cannot assume an automatic carriage return at the end of a printed string.

Note that 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 in order to have the letter printed out properly when using COUT.

The indirect addressing modes are used when you want to access memory in a very compact and efficient way. Let's consider the problem of clearing the screen, for instance. We want to put a space character in every memory location in the screen block (\$400-\$7FF). Here is one way of doing this:

```

1 *****
2 * AL07-SCREEN CLEAR PROGRAM 1A *
3 *****
4 *
5 *      OBJ  $300
6 *      ORG  $300
7 *
8 PTR    EQU  $06
9 *
10 ENTRY LDA  #$04
11      STA  PTR+1
12      LDY  #$00
13      STY  PTR
14 * SETS PTR (6,7) TO $400
15 START LDA  #$A0
16 LOOP  STA  (PTR),Y
17      INY
18      BNE  LOOP
19 NXT   INC  PTR+1
20      LDA  PTR+1
21      CMP  #$08
22      BCC  START
23 EXIT  RTS

```

Listed from the Monitor, it should appear like this:

```

*300L
0300- A9 04      LDA  #$04
0302- 85 07      STA  $07
0304- A0 00      LDY  #$00
0306- 84 06      STY  $06
0308- A9 A0      LDA  #$A0
030A- 91 06      STA  ($06),Y
030C- C8                INY
030D- D0 FB      BNE  $030A
030F- E6 07      INC  $07
0311- A5 07      LDA  $07
0313- C9 08      CMP  #$08
0315- 90 F1      BCC  $0308
0317- 60                RTS

```

We start off by initializing locations \$06 and \$07 to hold the base address of \$400, the first byte of the screen memory area. Then we enter a loop that runs the Y-Register from \$00 to \$FF. Since this is added to the base address in \$06, \$07, we then store an \$A0 (a space) in every location from \$400 to \$4FF. When Y is incremented from \$FF, it goes back to \$00, and this is detected by the BNE on line 18. At \$00, 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.

By changing the value of the # $\$A0$ to some other character, we can clear the screen to any character we wish. In fact, you can get the value from the keyboard as we've done in earlier programs.

```

1 *****
2 * AL07-SCREEN CLEAR PROGRAM 1B *
3 *****
4 *
5 *           OBJ  $300
6           ORG  $300
7 *
8 PTR       EQU  $06
9 CHAR     EQU  $08
10 KYBD    EQU  $C000
11 STROBE   EQU  $C010
12 *
13 ENTRY   LDA  #$04
14         STA  PTR+1
15         LDY  #$00
16         STY  PTR
17 * SETS PTR (6,7) TO $400
18 READ    LDA  KYBD
19         CMP  #$80           ; KEYPRESS?
20         BCC  READ          ; NO, TRY AGAIN
21         STA  STROBE        ; CLEAR KYBD STROBE
22         STA  CHAR
23 CLEAR   LDY  #$00
24         LDA  CHAR
25 LOOP    STA  (PTR),Y
26         INY
27         BNE  LOOP
28 NXT     INC  PTR+1
29         LDA  PTR+1
30         CMP  #$08
31         BCC  CLEAR
32 AGAIN   JMP  ENTRY

```

It should appear like this in listed form:

```

*300L
0300-  A9 04      LDA  #$04
0302-  85 07      STA  $07
0304-  A0 00      LDY  #$00
0306-  84 06      STY  $06
0308-  AD 00 C0   LDA  $C000
030B-  C9 80      CMP  #$80
030D-  90 F9      BCC  $0308
030F-  8D 10 C0   STA  $C010
0312-  85 08      STA  $08
0314-  A0 00      LDY  #$00
0316-  A5 08      LDA  $08
0318-  91 06      STA  ($06),Y
031A-  C8         INY
031B-  D0 FB      BNE  $0318
031D-  E6 07      INC  $07

```

```
031F-  A5 07      LDA  $07
0321-  C9 08      CMP  #$08
0323-  90 EF      BCC  $0314
0325-  4C 00 03   JMP  $0300
```

Enter this program and run from BASIC with a CALL 768. Each press will clear the screen to a different character. The screen should clear to the same character as the key you press, including the <SPACE> bar and special characters. In this program especially, you can see how fast machine language is. To clear the screen requires loading more than one thousand different locations with the given value. In Applesoft, this process would be quite slow by comparison. In assembly language, 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.

In this variation on program 1A we've used the principles from chapter six where we read the keyboard until we got a value greater than \$80, meaning a key has been pressed. This value is held temporarily in the variable CHAR so that it can be retrieved each time after incrementing the PTR in the NXT section.

See what variations you can make on this, or try the hi-res screen (\$2000 through \$3FFF).

Sound Generation

May 1981

Sound generation in assembly language is such a large topic in itself that an entire book could be done on that subject alone. However, simple routines are so easy that they're worth at least a brief examination here. These routines will not only allow you to put the commands you've learned to further use, but are also just plain fun.

The first element of a sound-generating routine is the speaker itself. Recall that the speaker is part of the memory range from \$C000 to \$C0FF that is devoted entirely to hardware items of the Apple II. In earlier programs, we looked at the keyboard by examining memory location \$C000. The speaker can be similarly accessed by looking at location \$C030. The exception here is that the value at \$C000 (the keyboard) varied according to what key was pressed, whereas with \$C030 (the speaker) there is no logical value returned.

Every time location \$C030 is accessed, the speaker will click once. This is easy to demonstrate. Simply enter the Monitor with a CALL -151. Enter C030 and press <RETURN>. You'll have to listen carefully, and you may have to try it several times. Each time, the speaker will click once. You can imagine that, if we could repeatedly access the speaker at a fast enough rate, the series of clicks would become a steady tone. In BASIC this can be done, although poorly, by a simple loop such as this:

```
10 X = PEEK(-16336): GOTO 10
```

The pitch of the tone generated depends on the rate at which the speaker is accessed. Because Integer BASIC is faster in its execution than Applesoft, the tone generated will be noticeably higher in pitch in the Integer version.

In assembly language, the program would look like this:

```
0300- AD 30 C0 LDA $C030
0303- 4C 00 03 JMP $0300
```

In this case I'm showing it as the Apple would directly disassemble it, as opposed to the usual assembly-language source listing. The program is so short that the easiest way to enter it is by typing in the hex code directly. To do this, enter the Monitor (CALL -151) and type:

```
300: AD 30 C0 4C 00 03
```


Then run the program by typing `300G`.

Disappointed? The program is working. The problem is that the routine is actually too fast for the speaker to respond. What's lacking here is some way of controlling the rate of execution of the loop. This is usually accomplished by putting a delay of some kind in the loop. We should also be able to specify the length of the delay, either before the program is run or, even better, during the execution of the program.

Delays

We can do this in any of three ways: (1) physically alter the length of the program to increase the execution time of each pass through the loop; (2) store a value somewhere in memory before running the program and then use that value in a delay loop; or (3) get the delay value on a continual basis from the outside world, such as from the keyboard or paddles.

For the first method, you can use a new and admittedly complex command. The mnemonic for this instruction is `NOP` and stands for No OPERATION. Whenever the 6502 microprocessor encounters this, it just continues to the next instruction without doing anything. This code is used for just what we need here—a time delay.

It is more often used, though, as either a temporary filler when assembling a block of code (such as for later data tables) or to cancel out existing operations in a previously written section of code. Quite often, this command (`$EA`, or 234 in decimal) is used in this manner to patch parts of the Apple DOS to cancel out various features that you no longer want to have active (such as the `NOT DIRECT` command error that prevents you from doing a `GOTO` directly to a line that has a DOS command on it).

In our sound routine, an `NOP` will take a certain amount of time even to pass over and will thus reduce the number of cycles per second of the tone frequency. The main problem in writing the new version will be determining the number of `NOPs` that will have to be inserted.

The easiest way to get a large block of memory cleared to a specific value is to use the move routine already present in the Monitor. To clear the block, load the first memory location in the range to be cleared with the desired value. Then type in the move command, moving everything from the beginning of the range to the end up one byte. For instance, to clear the range from `$300` to `$3A0` and fill it with `$EAs`, you would, from the Monitor of course, type in:

```
300: EA
301<300.3A0M
```

Note that we are clearing everything from `$300` to `$3A0` to contain the value `$EA`.

Now type in:

```
300: AD 30 C0
3A0: 4C 00 03
```

Then type in 300L, followed with L for each additional list section, to view your new program.

```
*300L
0300- AD 30 C0   LDA   $C030
0303- EA                NOP
0304- EA                NOP
0305- EA                NOP
0306- EA                NOP
0307- EA                NOP
0308- EA                NOP
0309- EA                NOP
*      *              *
*      *              *
*      *              *
0395- EA                NOP
0396- EA                NOP
0397- EA                NOP
0398- EA                NOP
0399- EA                NOP
039A- EA                NOP
039B- EA                NOP
039C- EA                NOP
039D- EA                NOP
039E- EA                NOP
039F- EA                NOP
03A0- 4C 00 03   JMP   $0300
```

Now run this with the usual 300G.

The tone produced should be a very nice, pure tone. The pitch of the tone can be controlled by moving the JMP \$300 to points of varying distance from the LDA \$C030. Granted, this is a very clumsy way of controlling the pitch and is rather permanent once created, but it does illustrate the basic principle.

For a different tone, hit RESET to stop the program, then type in:

```
350: 4C 00 03
```

When this is run (300G), the tone will be noticeably higher. The delay time is about half of what it was, and thus the frequency is twice the original value. Try typing in the three bytes in separate runs at \$320 and \$310. At \$310 you may not be able to hear the tone, because the pitch is now essentially in the ultrasonic range.

I think you'll also notice that all these tones are of a very pure nature and, in general, much nicer than those generated by a BASIC program.

Delay Value in Memory

Usually the way tone programs work is to pass a pitch value from BASIC by putting the value in a memory location. This program is an example of that technique.

```

1 *****
2 *      AL08-SOUND ROUTINE 2      *
3 *****
4 *
5 *
6 *      OBJ  $300
7 *      ORG  $300
8 *
9 PITCH  EQU  $06
10 SPKR  EQU  $C030
11 *
12 ENTRY  LDY  PITCH
13        LDA  SPKR
14 LOOP   DEY
15        BNE  LOOP
16        JMP  ENTRY
17 * INFINITE LOOP

```

When assembled, it should look like this:

```

*300L
0300-  A4 06      LDY  $06
0302-  AD 30 C0   LDA  $C030
0305-  88        DEY
0306-  D0 FD      BNE  $0305
0308-  4C 00 03   JMP  $0300

```

In this program, we get a value of \$00 to \$FF from location \$06 (labeled *pitch*) and put it in the Y-Register. The speaker is then clicked. At that point, we enter a delay loop that cycles n times where n is the number value for pitch held in location \$06.

To run this program, first load location \$06 with values of your choice (0 to 255 decimal, \$00 to \$FF hex) and then run with 300G. This is more compact and controllable than the first example, but it still suffers from being an infinite loop. What we need to do is specify a duration for the tone as well. Again you have the option of either making the value part of the program or passing it in the same way as we're currently doing the value for pitch. Here's a listing for the new program:

```

1 *****
2 *      AL08-SOUND ROUTINE 3      *
3 *****
4 *
5 *
6 *      OBJ  $300

```

```

7          ORG  $300
8 *
9 PITCH    EQU  $06
10 DURATION EQU  $07
11 SPKR    EQU  $C030
12 *
13 ENTRY   LDX  DURATION
14 LOOP    LDY  PITCH
15         LDA  SPKR
16 DELAY   DEY
17         BNE  DELAY
18 DRTN    DEX
19         BNE  LOOP
20 EXIT    RTS

```

Disassembled, it will appear like this:

```

*300L
0300-  A6 07      LDX  $07
0302-  A4 06      LDY  $06
0304-  AD 30 C0   LDA  $C030
0307-  88        DEY
0308-  D0 FD      BNE  $0307
030A-  CA        DEX
030B-  D0 F5      BNE  $0302
030D-  60        RTS

```

This routine is used by loading \$06 with a value for the pitch you desire, \$07 with a value for how long you want the tone to last, and then running the routine with the 300G.

Examining this listing, you'll see that it is basically an extension of routine 2. In this revised version, instead of always jumping back to the LDY of the play cycle, we decrement a counter (the X-Register). This counts the number of times we're allowed to pass through the loop, and thus the final length of the play.

This can be used by BASIC programs, as illustrated by this sample Applesoft listing:

```

10 PRINT CHR$(4);"BLOAD AL08.SOUND3,A$300"
20 INPUT "PITCH, DURATION?";P,D
30 POKE 6,P: POKE 7,D
40 CALL 768
50 PRINT
60 GOTO 20

```

This makes it very easy to experiment with different values for both pitch and duration. The main bug in this routine is that even for a fixed value for duration, the length of the note will vary depending on the pitch specified. This is because less time spent in the delay loop means less overall execution time for the routine as a whole.

Delay from the Keyboard or Paddles

The next variation is to get the pitch on a continual basis from an outside source. In this case, the source will be the keyboard. Type in and assemble this source listing:

```

1 *****
2 *      AL08-SOUND ROUTINE 4      *
3 *****
4 *
5 *
6 *      OBJ  $300
7      ORG  $300
8 *
9 KYBD   EQU  $C000
10 STROBE EQU  $C010
11 SPKR   EQU  $C030
12 *
13 ENTRY  LDA  KYBD
14        STA  STROBE
15        CMP  #$80
16        BEQ  EXIT
17        TAY
18 LOOP   LDA  SPKR
19 DELAY  DEY
20        BNE  DELAY
21        JMP  ENTRY
22 EXIT   RTS

```

In memory, it should look like this:

```

*300L
0300- AD 00 C0   LDA  $C000
0303- 8D 10 C0   STA  $C010
0306- C9 80     CMP  #$80
0308- F0 0A     BEQ  $0314
030A- A8        TAY
030B- AD 30 C0   LDA  $C030
030E- 88        DEY
030F- D0 FD     BNE  $030E
0311- 4C 00 03  JMP  $0300
0314- 60        RTS

```

Running this will give you a really easy way of passing tone values to the routine. Characters with low ASCII values will produce higher tones than ones with higher values. This means that the control characters will produce unusually high tones. To exit press <CTRL><SHIFT>P (<CTRL>@).

Let's review how the routine functions.

At the entry point (\$300), the very first thing done is to get a value from the keyboard. The strobe is then cleared, and an immediate check done for #\$80. Remember that \$80 is added to the ASCII value for each key pressed when read at \$C000. A value less than \$80 means no key has been pressed. Checking specifi-

cally for \$80, the computer looks to see if a <CTRL>@ has been pressed. This is just a nice touch to give us a way of exiting the program. After the check, we transfer the Accumulator value (equivalent to pitch in the earlier programs) to the Y-Register and finish with the same routine used in Sound Routine 2.

Of course, I have to give you at least one program using the paddles. This one gives us an opportunity to use the external routines in the Monitor, too. Don't forget that using the routines already present in the Monitor is the real secret to easy assembly-language programming. It saves you the trouble of having to write your own I/O and other sophisticated routines and lets you concentrate on those aspects unique to your program.

Now for the program:

```

1 *****
2 *      AL08-SOUND ROUTINE 5      *
3 *****
4 *
5 *
6 *      OBJ   $300
7      ORG   $300
8 *
9 PDL     EQU  $FB1E
10 SPKR   EQU  $C030
11 *
12 ENTRY  LDX  #$00
13        JSR  PDL
14        LDA  SPKR
15        LDX  #$01
16        JSR  PDL
17        LDA  SPKR
18        JMP  ENTRY
19 * INFINITE LOOP

```

The Monitor will list this as:

```

*300L
0300-  A2 00      LDX  #$00
0302-  20 1E FB   JSR  $FB1E
0305-  AD 30 C0   LDA  $C030
0308-  A2 01      LDX  #$01
030A-  20 1E FB   JSR  $FB1E
030D-  AD 30 C0   LDA  $C030
0310-  4C 00 03   JMP  $0300

```

Running this should produce some really interesting results. The theory of this routine is elegantly simple. It turns out that just reading a paddle takes a certain amount of time, sufficient to create our needed delay. The greater the paddle reading, the longer the delay to read it.

What happens in this routine is that we actually have two distinct delays created, one by each paddle. Remember that to read a paddle, you first have to load the X-Register with the number of the paddle you wish to read and then do

the JSR to the paddle read routine. The value is returned in the Y-Register, but in this case we don't need to know what the value was.

The combination of the two different periods of delay creates the effect of two tones at once and a number of other very unique sounds.

This has been only the most basic discussion of sound generation in assembly language, but I think you'll find that it illustrates what can be done with only a few commands, and that machine language offers many advantages in terms of memory use and execution speed.

9

The Stack

June 1981

One of the more obscure parts of the operation of the Apple is related to something called the stack. This is a part of memory reserved for holding return addresses for GOSUBs and FOR-NEXT loops, and a few other operations in direct machine code.

If you want to impress your friends with your knowledge of assembly language, just throw this term around in a confident manner and they'll figure you must be an expert!

The stack can be thought of like those spring-loaded plate holders they have in restaurants. Plates are loaded onto the top of a cylinder with a spring-loaded platform in it. As more plates are added, the rest get pushed down. The plates must always be removed in the opposite order from that in which they are put in. The catch phrase for this is LIFO, for Last-In, First-Out. The first location loaded in the 6502 stack is \$1FF. Rather than pushing everything down toward \$100 each time a new value is put on the stack, the 6502 has a Stack Pointer that is adjusted as new data is added. Successive values are added in descending order, with the Stack Pointer being reset each time to indicate the position of the next available location. Thus the table is created in reverse order, building downward.

The technical details of its operation are not required to make good use of the stack. One of the most convenient things the stack can be used for is to hold values temporarily while you're doing something else. Normally in a program, we'd have to assign a zero-page location to hold a value. For instance, consider this program:

```
1 *****
2 * AL09-BYTE DISPLAY PROGRAM 1 *
3 *****
4 *
5 *      OBJ  $300
6 *      ORG  $300
7 *
8 CHR    EQU  $06
9 PRBYTE EQU  $FDDA
10 COUT  EQU  $FDED
11 PREAD EQU  $FB1E
12 HOME  EQU  $FC58
13 *
```



```

14 START   JSR  HOME
15 GETCHR  LDX  #$00
16         JSR  PREAD
17         STY  CHR
18         TYA
19         JSR  PRBYTE
20         LDA  #$A0      ; SPACE
21         JSR  COUT
22         LDA  CHR
23         JSR  COUT
24         LDA  #$8D      ; RETURN
25         JSR  COUT
26         JMP  GETCHR

```

This will be listed by the Monitor as:

```

*300L
0300- 20 58 FC   JSR  $FC58
0303- A2 00     LDX  #$00
0305- 20 1E FB   JSR  $FB1E
0308- 84 06     STY  $06
030A- 98        TYA
030B- 20 DA FD   JSR  $FDDA
030E- A9 A0     LDA  #$A0
0310- 20 ED FD   JSR  $FDED
0313- A5 06     LDA  $06
0315- 20 ED FD   JSR  $FDED
0318- A9 8D     LDA  #$8D
031A- 20 ED FD   JSR  $FDED
031D- 4C 03 03   JMP  $0303

```

This program gets a value from \$00 to \$FF from paddle 0, and stores it in location \$06. This is needed because the JSR to \$FDDA (a handy routine that prints the hex number in the Accumulator) scrambles the Accumulator and Y-Register. We want to keep the value at hand because the ASCII character corresponding to it is then printed out right after the number using COUT. The cycle then repeats until you press RESET.

Location \$06 is used for only a moment each pass to store the value temporarily. In addition, it commits that zero-page location to use and thus limits our choices when we need other places to store something. A better system is to make use of the stack. The commands to do this are PHA and PLA. PHA stands for “Push Accumulator onto stack.” When this is used in line 17 below, the value currently in the Accumulator is put onto the stack. The Accumulator itself goes unaltered, and none of the status flags, such as the carry or zero flags, are conditioned. The value is simply copied and stored for us.

Later on, when we want to retrieve the value, the PLA (“Pull Accumulator from stack”) on line 21 pulls the value back off the stack into the Accumulator. A PLA command does condition the zero flag, and also the sign bit, which has not been covered yet.

Important: For each PHA there must be a PLA executed before encountering the next RTS in a program.

Here's the revised program:

```

1 *****
2 * AL09-BYTE DISPLAY PROGRAM 2 *
3 *****
4 *
5 *          OBJ  $300
6          ORG  $300
7 *
8 PRBYTE   EQU  $FDDA
9 COUT     EQU  $FDED
10 PREAD   EQU  $FB1E
11 HOME    EQU  $FC58
12 *
13 START   JSR  HOME
14 GETCHR  LDX  #$00
15         JSR  PREAD
16         TYA
17         PHA
18         JSR  PRBYTE
19         LDA  #$A0      ; SPACE
20         JSR  COUT
21         PLA
22         JSR  COUT
23         LDA  #$8D      ; RETURN
24         JSR  COUT
25         JMP  GETCHR

```

This will list like so:

```

0300- 20 58 FC   JSR  $FC58
0303- A2 00     LDX  #$00
0305- 20 1E FB   JSR  $FB1E
0308- 98        TYA
0309- 48        PHA
030A- 20 DA FD   JSR  $FDDA
030D- A9 A0     LDA  #$A0
030F- 20 ED FD   JSR  $FDED
0312- 68        PLA
0313- 20 ED FD   JSR  $FDED
0316- A9 8D     LDA  #$8D
0318- 20 ED FD   JSR  $FDED
031B- 4C 03 03   JMP  $0303

```

The stack is also used automatically by the 6502 for storing the return address for each JSR as it's encountered. Each time you do a PHA, this address is buried one level deeper. You must have done an equivalent number of PLAs at some point in the routine before reaching the next RTS to have things work properly.

Also remember, if you want to store more than one value, you must retrieve the values in the opposite order in which they were stored. Once a value is

removed from the stack with the PLA, it is essentially gone forever from the stack unless you put it back directly.

Stack Limit

There is a limit to how much you can put in the stack. The limit of sixteen nested GOSUBs and FOR-NEXT loops in BASIC is related to this. Technically you can put 256 one-byte values or 128 RTS addresses on the stack, but the Apple also uses it for its own operations, and many times you have BASIC going, too.

In general, though, the stack rarely fills up unless you're getting extreme in its use, and at that point the code probably will be so tangled in nested subroutines that you may want to consider a rewrite anyway!

Try using the stack in some of your own programs; I think you'll find it quite useful.

10

Addition and Subtraction

July 1981

Now let's look at the simple math operations of addition and subtraction in assembly language. To an extent, we've already done some of this. The increment and decrement commands (INC/DEC, and so on) add and subtract for us. Unfortunately, they only do so by one each time (VALUE+1 or VALUE-1).

If you're really ambitious you could, with the commands you have already, add or subtract any number by using a loop of repetitive operations, but this would be a bit tedious, not to mention slow. Fortunately a better method exists. But first, a quick review of some binary math facts.

In chapter four we discussed the idea behind binary numbers and why they're so important in computers. I would like to further elaborate on the topic now and show how the idea of binary numbers applies to basic arithmetic operations in assembly-language programming.

Binary Numbers

By now you're certainly comfortable with the idea of a byte being an individual memory location which can hold a value from \$00 to \$FF (0 to 255). 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.

Each byte can be thought of as being physically made up of eight individual switches that can be in either an on or off position. We can count by assigning each possible combination of ons and offs a unique number value.

In the following diagrams, if a switch is off, it will be represented by a 0 in its particular position. If it's on, a 1 will be shown. When all the switches are off, we'll call that 0.

In base two, each of the eight positions in the byte is called a bit, and the positions are numbered from right to left: [7 6 5 4 3 2 1 0].

The pattern for counting is similar to normal decimal or hex notation. The value is increased by adding one each time to the digit on the far right, *carrying* as necessary. In base ten you'd have to carry every tenth count, and in hex every sixteenth. In base two, the carry will be done *every other time!*

So...the first few numbers look like this:

Hex	Decimal	Binary
\$00	0	0000 0000
\$01	1	0000 0001
\$02	2	0000 0010
\$03	3	0000 0011
\$04	4	0000 0100

Notice that in going from the value 1 to the value 2, we add a 1 to the 1 already at 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	1111 1101
\$FE	254	1111 1110
\$FF	255	1111 1111

Now the most important part. 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 a one to the next position to the left; unfortunately, that next position doesn't exist!

Addition with ADC

This is where the carry bit of the Status Register is used again. Before, it was used in the compare operations (CMP, for instance), but, as it happens, it is also conditioned by the next command, ADC. This stands for ADD with Carry. When the \$FF is incremented using an ADC command, things will look like this:

			Carry
\$100	256	0000 0000	1

The byte has returned to a value of 0 and the carry bit is set to a one.

We discussed the wrap-around to 0 earlier, with the increment/decrement commands, but we didn't mention the carry. That's because the INC/DEC commands don't affect the carry flag.

However, the ADC command does condition the carry flag. The carry will be set whenever the result of the addition is greater than \$FF.¹ With ADC, you can make your counters increment by values other than one—rather like the FOR I = 1 TO 10 STEP 5 statement in BASIC. But ADC is used more often for general math operations, such as calculating new addresses or screen positions, among a wide variety of other applications.

Whenever ADC is used, the value indicated is added to the contents of the Accumulator. The value can be stated either directly by use of an immediate value or with an indirect value.

¹ [CT] Similarly, the carry will be cleared when the result is \$FF or less.

Important Note: Although the ADC conditions the carry after it is executed, it cannot be assumed that the carry is conveniently standing in a clear condition when the addition routine is begun.

For example, consider this simple program:

```
LDA #$05
ADC #$00
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 \$05. If, however, the carry had been set (perhaps as the result of some other operation), then the carry bit would be included and the result would be \$06.

The point of all this is that the carry flag must be cleared before the first ADC operation. The example above should have been written as:

```
CLC (Clear Carry)
LDA #$05
ADC #$00
STA RESULT
```

In this case, result will always end up holding the value \$05. Here are some sample programs for using the ADC. Note the use of the CLC before each ADC.

```

1 *****
2 * AL10-ADC SAMPLE PROGRAM 1 *
3 *****
4 *
5 *          OBJ $300
6 *          ORG $300
7 *
8 N1        EQU $06
9 N2        EQU $08
10 RSLT     EQU $0A
11 *
0300: A5 06 12 START   LDA N1
0302: 18    13         CLC
0303: 65 08 14         ADC N2
0305: 85 0A 15         STA RSLT
0307: 60    16 END     RTS
```

```

1 *****
2 * AL10-ADC SAMPLE PROGRAM 2 *
3 *****
4 *
5 *          OBJ $300
6 *          ORG $300
7 *
8 N1        EQU $06
9 RSLT     EQU $0A
10 *
11 *
```

Assembly Lines

```

0300: A5 06    12  START   LDA   N1
0302: 18      13          CLC
0303: 69 80    14          ADC   #$80
0305: 85 0A    15          STA   RSLT
0307: 60      16  END     RTS

1 *****
2 *  AL10-ADC SAMPLE PROGRAM 3  *
3 *****
4 *
5 *          OBJ   $300
6          ORG   $300
7 *
8 N1        EQU   $06
9 INDX      EQU   $08
10 RSLT     EQU   $0A
11 TBL      EQU   $300
12 *
0300: A5 06    13  START   LDA   N1
0302: A6 08    14          LDX   INDX
0304: 18      15          CLC
0305: 7D 00 03 16          ADC   TBL,X
0308: 85 0A    17          STA   RSLT
030A: 60      18  END     RTS

1 *****
2 *  AL10-ADC SAMPLE PROGRAM 4  *
3 *****
4 *
5 *          OBJ   $300
6          ORG   $300
7 *
8 N1        EQU   $06
9 INDX      EQU   $08
10 RSLT     EQU   $0A
11 PTR      EQU   $1E
12 *
0300: A9 00    13  START   LDA   #$00
0302: 85 1E    14          STA   PTR
0304: A9 03    15          LDA   #$03
0306: 85 1F    16          STA   PTR+1
0308: A5 06    17          LDA   N1
030A: A4 08    18          LDY   INDX
030C: 18      19          CLC
030D: 71 1E    20          ADC   (PTR),Y
030F: 85 0A    21          STA   RSLT
0311: 60      22  END     RTS

```

In the first program, the value in N1 is added to the contents of N2. In the second, N1 is added to the immediate value \$80. Note the CLC before the ADC to ensure an accurate result. This result is then returned in location \$0A. This routine could be used as a subroutine for another assembly-language program, or it could be called from BASIC after passing the values to locations \$06 and \$08.

The latter two programs are more elaborate examples where the indirect modes are used to find the value from a table starting at \$300. In program 3, an index value is passed to location \$08. That is used as an offset via the X-Register. In program 4, the low-order and high-order bytes for the address \$300 are first put in a pair of pointer bytes (\$1E, \$1F) and the offset is put in the Y-Register.

In all of these programs, however, we are limited to adding two single-byte values and further restricted to a one-byte result. Not very practical in the real world.

The solution is to use the carry flag to create a two-byte addition routine. Here's an example:

```

1 *****
2 * AL10-ADC SAMPLE PROGRAM 5A *
3 *****
4 *
5 *           OBJ $300
6           ORG $300
7 *
8 N1        EQU $06
9 N2        EQU $08
10 RSLT     EQU $0A
11 *
0300: 18    12 START   CLC
0301: A5 06 13         LDA  N1
0303: 65 08 14         ADC  N2
0305: 85 0A 15         STA  RSLT
0307: A5 07 16         LDA  N1+1
0309: 65 09 17         ADC  N2+1
030B: 85 0B 18         STA  RSLT+1
030D: 60    19 END     RTS

```

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. (If you're unsure of the idea of low- and high-order bytes, refer to chapter two, footnote one). This allows us to use values and results from \$00 to \$FFFF (0 to 65535). This is sufficient for any address in the Apple II, although by using three or more bytes, we could accommodate numbers much larger than \$FFFF.

A few words of explanation about this program. First, the CLC has been moved to the beginning of the routine. Although it need only precede the ADC command, it has no effect on the LDA, so it is put at the beginning of the routine for aesthetic purposes. Once the two low-order bytes of N1 and N2 are added and the partial result stored, the high-order bytes are added. If the result of this first addition is greater than 255, the carry will be set and an extra unit added in the second addition. Note that the carry remains unaffected during the LDA N1+1 operation.

The Monitor listing is given for this one so that you can call it from this BASIC program:²

```

0 REM AL10-ADC 5A ADDITION ROUTINE
10 HOME
15 FOR I = 0 TO 13: READ OP: POKE 768 + I,OP: NEXT I
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
90 DATA 24, 165, 6, 101, 8, 133, 10, 165, 7, 101, 9, 133, 11, 96

```

The ABS() statements on line 30 eliminate values less than 0. 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 always to be \$FF or less, so only one byte is needed for N2. A two/one addition routine looks like this:

```

1 *****
2 * AL10-ADC SAMPLE PROGRAM 5B *
3 *****
4 *
5 * OBJ $300
6 * ORG $300
7 *
8 N1 EQU $06
9 N2 EQU $08
10 RSLT EQU $0A
11 *
0300: 18 START CLC
0301: A5 06 LDA N1
0303: 65 08 ADC N2
0305: 85 0A STA RSLT
0307: 90 06 BCC END
0309: A5 07 LDA N1+1
030B: 69 00 ADC #$00
030D: 85 0B STA RSLT+1
030F: 60 20 END RTS

```

For speed, if a carry isn't generated on line 14, the program skips directly to the end. If, however, 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. If speed is not a concern, the BCC can be left out with no ill effect.

² [CT] This was changed to include the machine code within the DATA statement.

The following program shows an alternate method using the INC command. In this case, the BCC is required for proper operation.

```

1 *****
2 * AL10-ADC SAMPLE PROGRAM 5C *
3 *****
4 *
5 *          OBJ $300
6          ORG $300
7 *
8 N1        EQU $06
9 N2        EQU $08
10 RSLT     EQU $0A
11 *
0300: 18   12 START   CLC
0301: A5 06 13         LDA  N1
0303: 65 08 14         ADC  N2
0305: 85 0A 15         STA  RSLT
0307: 90 06 16         BCC  END
0309: A5 07 17         LDA  N1+1
030B: 85 0B 18         STA  RSLT+1
030D: E6 0B 19         INC  RSLT+1
030F: 60   20 END     RTS

```

The reason for bringing up listing 5C is that the most common reason for adding one to a two-byte number is to increment an address pointer by one. In that case, the result is usually put right back in the original location rather than in a separate RESULT. A routine for this is more compact and would look like this:

```

1 *****
2 * AL10-ADC SAMPLE PROGRAM 5D *
3 *****
4 *
5 *          OBJ $300
6          ORG $300
7 *
8 N1        EQU $06
9 *
0300: 18   10 START   CLC
0301: E6 06 11         INC  N1
0303: D0 02 12         BNE  END
0305: E6 07 13         INC  N1+1
0307: 60   14 END     RTS

```

Subtraction

Subtraction is done like addition except that a *borrow* is required. Rather than using a separate flag for this operation, the computer recognizes the carry flag as sort of a reverse borrow.

That is, a *set* carry flag will be treated by the subtract command as a *clear borrow* (no borrow taken); a *clear* carry as a *set borrow* (borrow unit taken).³

The command for subtraction is SBC, for SuBtract with Carry. The borrow is cleared with the command SEC, for SEt Carry. (Remember, things are backward here). A subtraction equivalent of program 5A looks like this:

```

1 *****
2 * AL10-SBC SAMPLE PROGRAM 6 *
3 *****
4 *
5 *      OBJ  $300
6 *      ORG  $300
7 *
8 N1     EQU  $06
9 N2     EQU  $08
10 RSLT  EQU  $0A
11 *
0300: 38 12 START SEC
0301: A5 06 13 LDA N1
0303: E5 08 14 SBC N2
0305: 85 0A 15 STA RSLT
0307: A5 07 16 LDA N1+1
0309: E5 09 17 SBC N2+1
030B: 85 0B 18 STA RSLT+1
030D: 60 19 END RTS

```

The program can be called with the same BASIC program that we used for the addition routine.

Positive and Negative Numbers

So far we have discussed only how to represent whole numbers greater than or equal to zero using one or two bytes. A reasonable question then is: “How do we represent 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 instance, $3 - 5 = -2$, or when adding a positive number to a negative number, for instance, $5 + -8 = -3$.

³ [CT] Just like ADC, SBC also conditions the carry flag. If the result requires a borrow then the carry is *cleared* (borrow *set*, for example $\$50 - \80). If the result does not require a borrow then the carry is *set* (borrow *clear*, for example $\$50 - \30).

To be successful, then, we must come up with a system using the eight bits in each byte that will be consistent with the signed arithmetic that we are currently familiar with.

The Sign Bit

The most immediate solution to the question of signed numbers is to use bit 7 to indicate whether a number is positive or negative. If the bit is clear, the number is positive. If the bit is set, the number will be regarded as negative.

Thus +5 would be represented: `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 -127 to +127. When using two bytes to represent a number such as an address, this means that we'll be limited to the range of -32767 to +32767. Sound familiar? If you've had any experience with Integer BASIC, then you'll recognize this as the maximum range of number values within that language.⁴

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 negative number using this scheme we get disturbing results:

```

      5  00000101
+     -8  10001000
-     -3  10001101 = -13!
```

Although we should get -3 as the result, using our signed bit system we get -13. Tsk, tsk. There must be a better way. Well, with the help of what is essentially a little numeric magic we can get something which works, although some of the conceptual simplicity gets lost in the process.

What we'll invoke is the idea of number *complements*. The simplest complement is what is called a *one's complement*. The one's complement of a number is obtained by reversing each 1 and 0 throughout the original binary number.

For example, the one's complement to 5 would be:

```

00000101 = 5
11111010 = -5
```

For 8, it would be:

```

00001000 = 8
11110111 = -8
```

This process is essentially one of *definition*: we simply declare to the world that 11110111 will now represent -8 without specifically trying to justify it.

⁴ [CT] Technically, for two's complement, the minimum should be -32768. However, Applesoft and Integer BASIC restrict the minimum integer to -32767. See chapter 17 for a way to fool Applesoft into displaying -32768.

Undoubtedly there are lovely mathematical proofs of such things that present marvelous ways of spending an afternoon but, for our purpose, a general notion of what the term means will be sufficient. Fortunately computers are very good at following arbitrary numbering schemes without asking “but why is it that way?”

Now let’s see if we’re any closer to a working system:

$$\begin{array}{r}
 5 \quad 00000101 \\
 + \text{-}8 \quad 11110111 \\
 \hline
 \text{-}3 \quad 11111100 = -3 \qquad (00000011 = +3)
 \end{array}$$

Hmmm...Seems to work pretty well. Let’s try another:

$$\begin{array}{r}
 \text{-}5 \quad 11111010 \\
 + \quad 8 \quad 00001000 \\
 \hline
 3 \quad 00000010 = 2 \text{ (plus carry)}
 \end{array}$$

Well, our answers will be right half the time... It turns out there is a final solution, and that is to use what is called the *two’s complement* system.

The only difference between this and the one’s complement system is that after deriving the negative number by reversing each bit of its corresponding positive number, *we add one*.

Sounds mysterious. Let’s see how it looks:

For -5: $ \begin{array}{r} 5 = 00000101 \\ \downarrow \\ 11111010 \\ \downarrow \\ \text{-}5 = 11111011 \end{array} $	one’s complement... now add one...	For -8: $ \begin{array}{r} 8 = 00001000 \\ \downarrow \\ 11110111 \\ \downarrow \\ \text{-}8 = 11110000 \end{array} $
--	---	--

Now let’s try the two earlier operations:

$ \begin{array}{r} 5 \quad 00000101 \\ + \text{-}8 \quad 11111000 \\ \hline \text{-}3 \quad 11111101 = -3 \end{array} $ <p>Does 11111101 equal -3?</p> <p>sample #: $00000011 = 3$ one’s comp: 11111100 two’s comp: $11111101 = -3$</p>	$ \begin{array}{r} \text{-}5 \quad 11111011 \\ + \quad 8 \quad 00001000 \\ \hline 3 \quad 00000011 = 3 \\ \text{(plus carry)} \end{array} $
---	---

At last! It works in both cases. It turns out that two’s complement math works in all cases, with the carry being ignored.

Now that you’ve mastered that, I’ll let you off the hook a bit by saying that none of this knowledge will be specifically required in any programs in this

book. However, it is a good thing to know about and is very useful in understanding the next idea, that of the sign and overflow flags in the Status Register.

The Sign Flag

Since bit 7 of any byte can represent whether the number is positive or negative, a flag in the Status Register is provided for easy testing of the nature of a given byte. Whenever a byte is loaded into a register, or when an arithmetic operation is done, the sign flag will be conditioned according to what the state of bit 7 is.

For example, LDA # $\$80$ will set the sign flag to 1, whereas LDA # $\$40$ would set it to 0. This is tested using the commands BPL and BMI. BPL stands for Branch on Plus, and BMI stands for Branch on Minus.

Regardless of whether you are using signed numbers, 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, $\$C000$, gets the high bit set whenever a key is pressed.

Up until now we've always tested by comparing the value returned from $\$C000$ to # $\$80$, such as in this listing:

```

1 *****
2 * AL10-BPL KEYTEST PROGRAM 1 *
3 *****
4 *
5 *
6 *      OBJ $300
7 *      ORG $300
8 *
9 KYBD   EQU $C000
10 STROBE EQU $C010
11 *
12 CHECK LDA KYBD
13      CMP #80
14      BCC CHECK          ; NO KEYPRESS
15 *
16 CLR   STA STROBE
17 END   RTS
0300: AD 00 C0
0303: C9 80
0305: 90 F9
0307: 8D 10 C0
030A: 60
```

This program will stay in a loop until a key is pressed. The keypress is detected by the value returned from $\$C000$ being equal to or greater than # $\$80$. A more elegant method is to use the BPL command:

```

1 *****
2 * AL10-BPL KEYTEST PROGRAM 2 *
3 *****
4 *
5 *
6 *      OBJ $300
7 *      ORG $300
```

```

      8 *
      9 KYBD EQU $C000
     10 STROBE EQU $C010
     11 *
0300: AD 00 C0 12 CHECK LDA KYBD
0303: 10 FB 13 BPL CHECK ; NO KEYPRESS
     14 *
0305: 8D 10 C0 15 CLR STA STROBE
0308: 60 16 END RTS

```

In this case, as long as the high bit stays clear (i.e. no keypress), the BPL will be taken and the loop continued. As soon as a key is pressed, bit 7 will be set to 1 and the BPL will fail. The strobe is then cleared and the return done.

A similar technique is used for detecting whether a paddle pushbutton has been pressed.

```

      1 *****
      2 * AL10-BPL BUTTON TEST *
      3 *****
      4 *
      5 *
      6 * OBJ $300
      7 * ORG $300
      8 *
      9 PB0 EQU $C061
     10 *
     11 *
0300: AD 61 C0 12 CHECK LDA PB0
0303: 10 FB 13 BPL CHECK ; NO BUTTON PUSH
     14 *
0305: 60 15 END RTS

```

DOS and Disk Access

August 1981

One of the more useful applications of assembly language is in accessing the disk directly to store or retrieve data. You might do this to modify information already on the disk, such as when you're making custom modifications to DOS, or to deal with data within files on the disk, such as when you're patching or repairing damaged or improperly written files.

To cover DOS well requires more than a few chapters such as this. My intent here, then, is to supply you with enough information to allow you to access any portion of a disk and to have enough basic understanding of the overall layout of DOS and disks to make some sense of what you find there.¹

Here's what we'll cover in this chapter. First, we'll paint a general overview of what DOS is and how the data on the diskette is arranged. Then you'll learn a general access utility with which you can read and write any single block of data from a disk. With these, you'll have a starting point for your own explorations of this aspect of your Apple computer.

The Overview: DOS

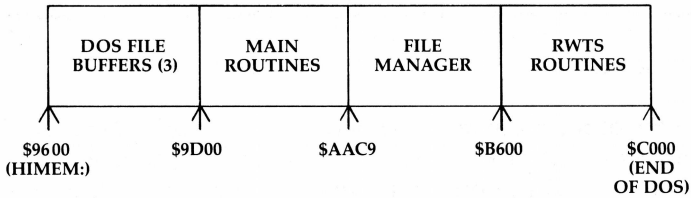
An Apple without a disk drive has no way of understanding commands like CATALOG or READ. These new words must enter its vocabulary from somewhere. When an Apple with a disk drive attached is first turned on or a PR#6 is done, this information is loaded into the computer by a process known as booting.

During the booting process, a small amount of machine-language code on the disk interface card reads in data from a small portion of the disk. This data contains the code necessary to read another 10K of machine language referred to as DOS. This block of routines is responsible for all disk-related operations in the computer. It normally resides in the upper 10K or so of memory, from \$9600 to \$BFFF.

After booting, the organization of the memory used by DOS looks something like the figure shown on the next page.

¹ For a detailed look at DOS, I recommend the book *Beneath Apple DOS*, by Dan Worth and Pieter Lechner (Reseda, CA: Quality Software, 1981).

[CT] For hints on implementing the code in ProDOS, see *Beneath Apple ProDOS*, by Dan Worth and Pieter Lechner (Quality Software, 1984).



The first area contains the three buffers set aside for the flow of data to and from the disk. A buffer is a block of memory reserved to hold data temporarily while it's being transferred. The `MAXFILES` command (a legal DOS command; see your manual if you haven't encountered it before) can alter the number of buffers reserved and thus change the beginning address from `$9600` to other values. As it happens, three buffers are almost never needed so, in a pinch for memory, you can usually set `MAXFILES` to 2, and often just to 1.

For example, if you had opened a text file called `TEXTFILE`, the data being read or written would be transferred via buffer 1. If, while this file was still open, you did a catalog, buffer 2 would be put in use. If, instead, you opened two other files, say `TEXTFILE1` and `TEXTFILE2`, and then tried to do a `CATALOG`, you would get a `NO BUFFERS AVAILABLE` error (assuming `MAXFILES` was set at three). Buffer 1 starts at `$9AA6`, buffer 2 at `$9853`, and buffer 3 at `$9600`. If `MAXFILES` is set at 3 as in a normal system, it's occasionally useful to use the dead space of the unused buffer 3 for your own routines.

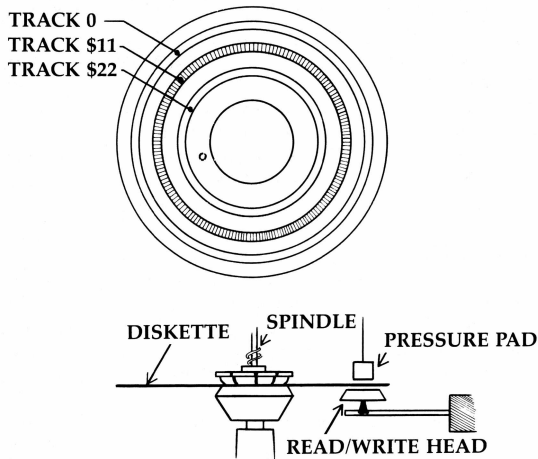
The main DOS routines starting at `$9D00` are the ones responsible for the interpreting commands such as `CATALOG` and, in general, for allowing DOS to talk to BASIC via statements prefixed with `<CTRL>D`.

The file manager is a set of routines that actually execute the various commands sent via the main routines and that makes sure files are stored in a logical (well, almost) manner on the disk. It takes care of finding a file you name, checking to see whether it's unlocked before a write, finding empty space on the disk for new data, and countless other tasks required to store even the simplest file.

When the file manager gets ready to read data from or write data to the disk, it makes use of the remaining routines, called the RWTS routines. This stands for Read/Write Track Sector. To understand fully what this section does, though, it will be necessary now to look at the general organization of the disk itself.

Diskette Organization

Physically, a diskette is coated with a material very similar to that on magnetic-recording tape. Small portions of the surface are individually magnetized to store the data in the form of ones and zeros.



But the diskette is more analogous to a vinyl record than to a continuous strip of tape. Arranged in concentric circles, there are thirty five individual *tracks*, each of which is divided into sixteen segments called *sectors*.²

Tracks are numbered from 0 to 34 (\$00 to \$22), starting with Track 0 at the outer edge of the diskette and track 34 nearest the center. Sectors are numbered from 0 to 15 (\$00 to \$0F) and are interleaved for fastest access. This means that sector 1 is not physically next to sector 0 on the diskette. Rather, the order is:

0-D-B-9-7-5-3-1-E-C-A-8-6-4-2-F

By the time DOS has read in and processed one sector, it doesn't have sufficient time to read the next physically-contiguous sector properly. If the sectors were arranged sequentially, DOS would have to wait for another entire revolution of the diskette to read the next sector. By examining the sequence you can see that after reading sector 0, DOS can let as many as six other sectors go by and still have time to start looking for sector 1. This alternation of sectors is sometimes called the *skew factor* or just *sector interleaving*.

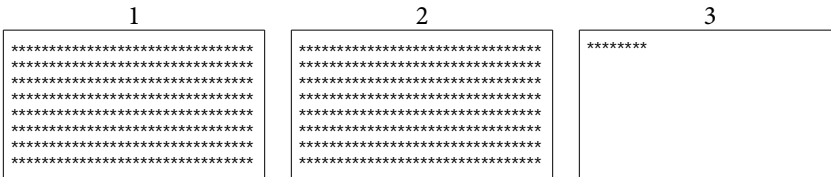
Looking for a given sector is done with two components. The first is a physical one, wherein the read/write head is positioned at a specific distance from the center of the diskette to access a given track. The sector is located via software by looking for a specific pattern of identifying bytes. In addition to the 256 bytes of actual data within a sector, each sector is preceded by a group of identifying and

² Throughout this discussion we will assume you are using DOS 3.3, which uses sixteen sectors per track. DOS 3.2 has only thirteen sectors per track but is rapidly becoming obsolete. If you're using DOS 3.2, the correction from sixteen to thirteen should be made in the topics throughout.

error-checking bytes. These include, for example, something like \$00 \$03 \$FE for track \$00, sector \$03, volume \$FE. By continuously reading these identification bytes until a match with the desired values occurs, a given sector may be located.

This software method of sector location is usually called *soft-sectoring*, and it's somewhat unique to the Apple. Most other microcomputers use *hard-sectoring*. Hard-sectoring means that hardware locates the sector as well as the track; sectors are found by means of indexing holes located around the center hole of a disk. Even Apple diskettes have this center hole, along with one to sixteen indexing holes in the media itself, but these aren't actually used by the disk drive. Because the Apple doesn't need these holes to index, using the second side of a disk is just a matter of properly notching the edge of the disk jacket to create another write-enable notch. We'll not go into the pros and cons of using the second side but will leave that to you. It's one of those topics guaranteed to be worth twenty to thirty minutes of conversation at any gathering of two or more Apple owners.

Each sector holds 256 (\$100) bytes of data. This data must always be written or read as a single block. Large files are therefore always made up of multiples of 256 bytes. Thus a 520-byte file takes up three entire sectors, even though most of the third sector is wasted space:



Certain tracks and sectors are reserved for specific information. Track 17 (\$11), for example, contains the directory. This gives each file a name, and also tells how to find out which sectors on the disk contain the data for each file. Track 17, sector 0 contains the Volume Table of Contents (VTOC), which is a master table of which sectors currently hold data, and which are available for storing new data. If all of track 17 is damaged, it may be nearly impossible to retrieve any data from the disk even though the files themselves might still be intact.

The other main reserved area is on tracks 0 through 2. These tracks hold the DOS that will be loaded when the disk is booted. If any of these tracks are damaged it will not be possible to boot the diskette...or if the disk does boot, DOS may not function properly.

As a variation on this theme, by making certain controlled changes to DOS directly on the disk you can create your own custom version of DOS to enhance what Apple originally had in mind. These enhancements will become part of

your system whenever you boot your modified diskette. Some modifications of this type are discussed below.

To gain access to a sector to make these changes, however, we need to be able to interface with the routines already in DOS to do our own operations. This is most easily done by using the RWTS routines mentioned earlier. Fortunately, Apple has made them fairly easy to use from the user's assembly-language program.

To use RWTS, you must do three general operations:

1. Specify the track and sector you wish to access.
2. Specify where the data is to be loaded to or read from (that is, give the buffer address).
3. Finally, call RWTS to do the read/write operation.

If the operation is to be a read, then we would presumably do something with the data in the buffer after the read is complete. If a write is to be done, then the buffer should be loaded before calling RWTS with the appropriate data. Usually, the way all this works is to read in a sector first, then make minor changes to the buffer, and then write the sector back out to the diskette.

Steps 1 and 2 are actually done in essentially the same operation, by setting up the IOB table ("Input/Output and control Block"). This is described in detail (along with the sector organization) in the *Apple DOS Manual*, but here's enough information to "make you dangerous," as the saying goes.

The IOB table is a table you make up and place at a location of your choice. (You can also make use of the one already in memory that is used in DOS operations.) Most people I know seem to prefer to make up their own, but my personal preference is to use the one in DOS. Since most people I know aren't at this keyboard right now, I'll explain how to use the table already set up in DOS.

The table is made up of seventeen bytes and starts at \$B7E8. It's organized like this:

Location	Code	Purpose
\$B7E8	\$01	IOB type indicator, must be \$01
B7E9	\$60	Slot number times sixteen ³
B7EA	\$01	Drive number
B7EB	\$00	Expected volume number
B7EC	\$12	Track number
B7ED	\$06	Sector number
B7EE	\$FB	Low-order byte of device characteristic table (DCT)
B7EF	\$B7	High-order byte of DCT

³ Notice that this calculation, like multiplying by ten in decimal, means just moving the hex digit to the left one place.

Location	Code	Purpose
B7F0	\$00	Low-order byte of data buffer starting address
B7F1	\$20	High-order byte of data buffer starting address
B7F2	\$00	Unused
B7F3	\$00	Unused
B7F4	\$02	Command code; \$02 = write
B7F5	\$00	Error code (or last byte of buffer read in)
B7F6	\$00	Actual volume number
B7F7	\$60	Previous slot number accessed
B7F8	\$01	Previous drive number accessed

Because DOS has already set this table up for you, it isn't necessary to load every location with the appropriate values. In fact, if you're willing to continue using the last accessed disk drive, you need only specify the track and sector, set the command code, and then clear the error and volume values to #00. However, for complete accuracy, the slot and drive values should also be set so you know for sure what the entry conditions are.

Once the IOB table has been set up, the Y-Register and Accumulator must be loaded with the low- and high-order bytes of the IOB table, and then the JSR to RWTS must be done. Although RWTS actually starts at \$B7B5, the call is usually done as JSR \$3D9 when DOS first boots. The advantage of calling here is that if Apple ever changes the location of RWTS, only the vector address at \$3D9 will be changed and a call to \$3D9 will still work.

A *vector* is the general term used for a memory location that holds the information for a second memory address. A vector is used so that a jump to a single place in memory can be routed to a number of other memory locations, usually the beginnings of various subroutines. A vector is rather like a telephone switchboard: even though the user always calls the same address, the program flow can be directed to any number of different places simply by changing two bytes at the vector location.

For example, suppose at location \$3F5 we were to put these three bytes:

```
3F5: 4C 00 03
```

Listed from the Monitor, this would disassemble as:

```
03F5- 4C 00 03    JMP    $0300
```

Now whenever you do a call to \$3F5, either by a CALL 1013 or 3F5G, the program will end up calling a routine at \$300. It would now be a simple matter to write a switching program that would rewrite the two bytes at \$3F6 and \$3F7 so that a call to \$3F5 would go anywhere we wanted.

As it happens, \$3F5 is used in just such a fashion by the ampersand (&) function of Applesoft. The *Applesoft II BASIC Programming Reference Manual* provides more information on this feature.

The best way to finish explaining how to use the IOB table and RWTS is to present the following utility to access a given track and sector using RWTS. We'll then step through the program and learn why the various steps are done to use RWTS successfully.

```

1 *****
2 *
3 * AL11-GENERAL PURPOSE RWTS *
4 * DOS UTILITY *
5 *
6 *****
7 *
8 *
9 * OBJ $300
10 * ORG $300
11 *
12 CTRK EQU $06
13 CSCT EQU $07
14 UDRIV EQU $08
15 USLOT EQU $09
16 BP EQU $0A ; BUFFER PTR.
17 UERR EQU $0C
18 UCMD EQU $E3
19 * USER SETS THIS TO HIS CMD
20 *
21 RWTS EQU $3D9
22 *
23 * BELOW ARE LOCS IN IOB
24 SLOT EQU $B7E9
25 DRIV EQU $B7EA
26 VOL EQU $B7EB
27 TRACK EQU $B7EC
28 SECTOR EQU $B7ED
29 BUFR EQU $B7F0
30 CMD EQU $B7F4
31 ERR EQU $B7F5
32 OSLOT EQU $B7F7
33 ODRIV EQU $B7F8
34 *
35 READ EQU $01
36 WRITE EQU $02
37 *
38 *
39 *
40 *****
41 * ENTRY CONDITIONS: SET *
42 * TRACK, SECTOR, SLOT, DRIVE, *
43 * BUFFER, AND COMMAND. *
44 *****
45 *
46 *
```

```

      47 *
0300: A9 00 48 CLEAR LDA #$00
0302: 8D EB B7 49 STA VOL
      50 *
0305: A5 09 51 LDA USLOT
0307: 8D E9 B7 52 STA SLOT
      53 *
030A: A5 08 54 LDA UDRIV
030C: 8D EA B7 55 STA DRIV
      56 *
030F: A5 06 57 LDA CTRK
0311: 8D EC B7 58 STA TRACK
      59 *
0314: A5 07 60 LDA CSCT
0316: 8D ED B7 61 STA SECTOR
      62 *
0319: A5 E3 63 LDA UCMD
031B: 8D F4 B7 64 STA CMD
      65 *
031E: A5 0A 66 LDA BP
0320: 8D F0 B7 67 STA BUFR
0323: A5 0B 68 LDA BP+1
0325: 8D F1 B7 69 STA BUFR+1
      70 *
0328: A9 B7 71 LDA #$B7
032A: A0 E8 72 LDY #$E8
032C: 20 D9 03 73 JSR RWTS
032F: 90 05 74 BCC EXIT
      75 *
0331: AD F5 B7 76 ERRHAND LDA ERR
0334: 85 0C 77 STA UERR
      78 *
0336: 60 79 EXIT RTS
      80 *
0337: CD 81 CHK

```

When this program runs, it assumes the user has set the desired values for the track and sector wanted, which slot and drive to use, where the buffer is, and whether to read or write.

Starting with the first functional line, line 48, the byte for the volume number in the IOB table (VOL) is stuffed with a 0. A value of 0 here tells RWTS any volume number is acceptable during the access. If we wanted to access only a particular volume number, a value from \$01 to \$FE would be used instead of \$00.⁴

In the next four sets of operations, the user values for the slot, drive, track, and sector numbers are put into the IOB table. Notice that, to have this work properly, you must set USLOT (\$09) to sixteen times the value for the slot you wish to use. For example, to access slot 5 you would store a #\$50 (80 decimal) in location \$09 just before calling this routine.

⁴ [CT] \$FF is not a valid DOS volume number.

The next pair of statements take the user command UCMD and put that in the table. If you want to read a sector, set UCMD = \$01. A write is UCMD = \$02. A few other options are seldom used. These are described in more detail in the DOS 3.3 manual in the section on RWTS.

Next, the buffer pointer is set to the value given by the user in locations \$0A and \$0B. The required space is 256 bytes (\$100) and can be put anywhere that won't conflict with data already in the computer. Convenient places are the number three DOS file buffer (\$9600), the input buffer itself (\$200), or an area of memory below \$9600 protected by setting HIMEM to an appropriate value.⁵ In the examples that follow, I'll use the area from \$1000 to \$10FF because no BASIC program will be running and \$1000 is a nice number. In this case, \$0A and \$0B will be loaded with #\$00 and #\$10, respectively.

Last of all, the Y-Register is loaded with #\$E8 and the Accumulator is loaded with #\$B7, the low-order and high-order bytes of the IOB table address.

After the call to RWTS via the vector at \$3D9, the carry flag is checked for an error. If the carry is clear, there was no error and the routine returns via the RTS. If an error is encountered, the code will be transferred from the IOB table to the user location. The possible error codes are:

Code Condition

\$10	Disk write-protected, and cannot be written to.
\$20	Volume mismatch error. Volume number found was different than specified.
\$40	Drive error. An error other than the three described here is happening (I/O error, for example).
\$80	Read error. RWTS will try forty-eight times to get to a good read; if it still fails, it will return with this error code.

DOS Modifications

The ERR byte of the IOB table is somewhat unusual in that it does not remain at 0 even if the read/write operation was successful. In actual operation, if an error does not occur, the ERR byte will contain the last byte of the sector just accessed.

It is important therefore to always use the carry flag to detect whether an error has occurred. In fact, as your experience grows, you will notice that a great many subroutines use the carry flag as an indicator of the results of the operation. In the case of RWTS, the carry will be cleared if the access was successful and

⁵ Note: The input buffer can be used only temporarily during your own routine. If you return to BASIC, or do any input or DOS commands, data in this area will be destroyed. Other than that, it's a handy place to use.

set if an error occurred. It is not necessary to condition the carry before calling RWTS.

One of the best ways to grasp this routine is to use it to modify the DOS on a sample disk and observe the differences. Before proceeding with the examples, boot an Apple master disk, then INIT a blank disk. This will be our test piece, so to speak. Do not try these experiments on a disk already containing important data. If done correctly the changes won't hurt, but if an error were to occur you could lose a good deal of work!

Disk-Volume Modification

First install the sector-access routine at \$300. Now insert the sample diskette. Enter the Monitor with CALL -151 and type in:

```
*06: 02 02 01 60 00 10
*E3: 01
```

This assumes your diskette is in drive 1, slot 6. Now enter:

```
*300G
```

The disk drive motor should come on. When it stops type in:

```
*10AFL
```

You should get something like this:

```
10AF- A0 C5      LDY  #$C5
10B1- CD D5 CC   CMP   $CCD5
10B4- CF         ???
10B5- D6 A0      DEC   $A0,X
10B7- CB         ???
10B8- D3         ???
10B9- C9 C4      CMP   #$C4
```

This apparent nonsense is the ASCII data for the words "DISK VOLUME" in reverse order. This is loaded in when the disk is booted and is used in all subsequent catalog operations.

The data was retrieved from track 2, sector 2, and put in a buffer starting at \$1000. The sequence we're interested in starts at byte \$AF in that sector. To modify that, type in:

```
*10AF: A0 D4 D3 C5 D4 A0 AD
*E3: 02
*300G
```

The first line rewrites the ASCII data there, the E3:02 changes the command to "write," and the 300G puts it back on the disk.

Now reboot the disk and then type in CATALOG. When the catalog prints to the screen, the new characters "DISK - TEST 254" should appear.⁶ By using the ASCII character chart in Appendix E, you can modify this part of the diskette to say anything you wish within the twelve-character limit.

Catalog Keypress Modification

Reinstall the sector access utility, put the sample disk in the drive again, and type in:

```
*06: 01 0D 01 60 00 10
*E3: 01
*300G
```

This will read track 1, sector \$0D, into the buffer. Type in:

```
*1039L
```

The first line listed should be:

```
1039- 20 0C FD JSR $FD0C
```

Change this to:

```
*1039: 4C DF BC (JMP $BCDF)
```

And rewrite to the disk:

```
*E3: 02
$300G
```

Now read in the section corresponding to \$BCDF (track 0, sector 6) by typing:

```
*06: 00 06
*E3: 01
*300G
```

And alter this section with:⁷

```
*10DF: 20 0C FD C9 8D D0 03 4C 2C AE 4C 3C AE
*E3: 02
*
*300G
```

⁶ [CT] The disk volume number (254) is still printed.

⁷ [CT] The Monitor listing looks like this:

```
10DF- 20 0C FD JSR $FD0C
10E2- C9 8D CMP #8D
10E4- D0 03 BNE $10E9
10E6- 4C 2C AE JMP $AE2C
10E9- 4C 3C AE JMP $AE3C
```

As it happens, this part of the disk isn't used and provides a nice place to put this new modification.

When you reboot after making this change, place a disk with a long catalog on it in the drive and type in CATALOG. When the listing pauses after the first group of names, press <RETURN>. The listing should stop, leaving the names just shown on the screen. If instead of pressing <RETURN> you press any other key, the catalog will continue just as it normally would, going on to the next group of names.

Both of these modifications will go into effect whenever you boot the sample disk. These features can also be propagated to other disks by booting the sample disk and using the new DOS to INIT fresh disks.

Many modifications to the existing DOS can be made this way, and we haven't even started to talk about storing binary data in general.

Bell Modification and Drive Access

(1) The first time you call the access utility from the Monitor, it will return with just the asterisk prompt. After that, unless you hit RESET or do a CATALOG, it will return with the asterisk and a beep. This is because the status storage byte for the Monitor (\$48) gets set to a nonzero value by RWTS. If the beep annoys you, modify the access utility to set \$48 back to #0 before returning.

(2) If you set the slot/drive values to something other than your active drive, the active drive will still be the one accessed when you do, for example, the next CATALOG. This is because DOS doesn't actually look at the last-slot/drive-accessed values when doing a CATALOG. Instead, it looks at \$AA66 for the volume number (usually #0), at \$AA68 for the drive number, and at \$AA6A for the slot number (times sixteen). If you have BASIC or assembly-language programs where you want to change the active drive values without having to do a CATALOG or give another command, then just POKE or STA the desired values in these three locations.

Have fun!

Shift Operators and Logical Operators

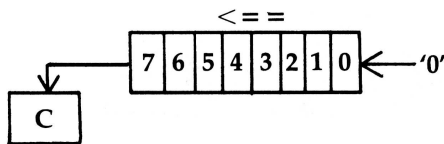
September 1981

Shift Operators

Here I'd like to cover two main groups of assembly-language commands: *shift operators* and *logical operators*. Shifts are easier to understand, so we'll do them first.

You'll recall that the Accumulator holds a single eight-bit value, and that in previous programs it has been possible to test individual bits by examining flags in the Status Register. An example of this was used in testing bit 7 after an LDA operation. If the Accumulator is loaded with a value from \$00 to \$7F, bit 7 is clear and only BPL tests will succeed, since the sign flag remains clear. If, however, a value from \$80 to \$FF is loaded, BMI will succeed since bit 7 would be set; hence the sign flag will also be a one.

The shift commands greatly extend our ability to test individual bits by giving us the option of shifting each bit in the Accumulator one position to the left or right. There are two direct shift commands, 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 0. In addition to the carry, the sign and zero flags are also affected. Some examples appear in the following table.

Value	Binary	Result	Binary	(C) Carry	(N) Sign	(Z) Zero
\$00	0000 0000	\$00	0000 0000	0	0	1
\$01	0000 0001	\$02	0000 0010	0	0	0
\$80	1000 0000	\$00	0000 0000	1	0	1
\$81	1000 0001	\$02	0000 0010	1	0	0
\$FF	1111 1111	\$FE	1111 1110	1	1	0

In the first case, there's no net change to the Accumulator, although the carry and sign flags are cleared and the zero flag is set. The 0 at each bit position was replaced by a 0 to its right.

However, in the case of \$01, the value in the Accumulator doubles to become \$02 as the 1 in bit 0 moves to the bit 1 position. In this case, all three flags will be cleared.

When the starting value is \$80 or greater, the carry will be set. In the case of \$80 itself, the Accumulator 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 in the Accumulator moves into position 7. Remember that in some schemes, bit 7 is used to indicate a negative number.

ASL has the effect of doubling the byte being operated on. This can be used as an easy way to multiply by two. In fact, by using multiple ASLs, you can multiply by two, four, eight, sixteen, and so on, depending on how many you use. In the discussion of DOS and RWTS in chapter 11, you might remember that the IOB table required the slot number byte in the table to be sixteen times the true value. If you didn't want to do the multiplication ahead of time, you could do it in your access program, as below.

```

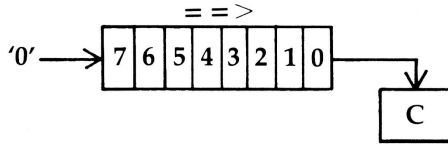
*
*
*
A5 09    LDA  USLOT
0A       ASL
0A       ASL
0A       ASL
0A       ASL
8D E9 B7 STA  SLOT
*
*
*
```

USLOT holds the value from one to seven that you pass to the routine and SLOT is the location in the IOB table in which the value for USLOT*16 should be placed. Even though the four ASLs look a bit redundant, notice that they only took four bytes. In fact, the LDA/STA steps consumed more bytes (five) than the four ASLs.

In general, then, ASL is used for these types of operations:

1. Multiply by two, four, eight, and so on.
2. Set or clear the carry *for free* while shifting for some other reason.
3. Test bits through 6. Note: This can be done, but it's usually done this way only for bit 6; there are, in general, better ways of testing specific bits, which we'll describe shortly.

The complement of the ASL command is LSR. It behaves identically except that the bits all shift to the right and bit 7 becomes a 0.

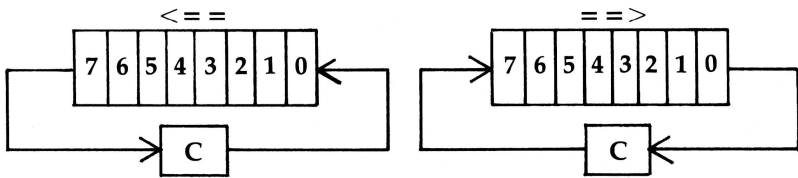


LSR – Logical Shift Right

LSR can be used to divide by multiples of two. It's also a nice way to test whether a number is even or odd. Even numbers always have bit 0 clear. Odd numbers always have it set. By doing an LSR followed by BCC or BCS, you can test for this. Whether a number is odd or even is sometimes called its *parity*. An even number has a parity of 0, and an odd number a parity of 1.

LSR also conditions the sign and zero flags.

In both LSR and ASL, one end or the other always gets forced to a 0. Sometimes this is not desirable. The solution to this is the *rotate* commands, ROL and ROR (ROtate Left, ROtate Right).



ROL – Rotate One Bit Left

ROR – Rotate One Bit Right

In 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 do turn up occasionally in math functions such as multiply and divide routines.

So far, all the examples have used the Accumulator as the byte to be shifted. As it happens, either the Accumulator or a memory location may be shifted. Addressing modes also include Zero Page,X and Absolute,X. The Y-Register cannot be used as an index in any of the shift operations.

Logical Operators

Logical operators are, to the uninitiated, some of the more esoteric of the assembly-language commands. As with everything we've done before, though, with a little explanation they'll become quite useful.

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.

As we've seen, each byte is made up of eight bits. Let's take just the left-most bit, bit 7, and see what kind of ideas can be played with. Normal text output on the Apple is always done with the high bit set. That is, all characters going out through COUT (\$FDED) should be equal to or greater than #\$80 (1000 0000 binary). Likewise, when watching the keyboard for a keypress, we wait until \$C000 has a value equal to or greater than #\$80.

Suppose we had a program wherein we would print characters to the screen only when a key was pressed and a standard character was being sent through the system. What we're saying is to print characters on the screen *only* when both the character *and* the keyboard buffer show bit 7 set to 1.

We can draw a simple chart that illustrates all the possibilities (and you know how fond computer people are of charts).

		Character Bit 7	
		0	1
Keyboard	0	0	0
Bit 7	1	0	1

The chart shows four possibilities. If the character's bit 7 is 0 (a non-standard character) and the keyboard bit is 0 (no keypress), then the character is not printed (a 0 result). Likewise, if only one of the conditions is being met but not the other, then the result is still 0, and the character is still not printed. Only when both desired conditions exist will we be allowed to print, as shown by the one as the result.

Taken to its extreme, what we end up with is a new mathematical function, AND. In the case of a single binary digit (or perhaps we should call it a bigit), the possibilities are few, and the answers are given as a simple 0 or 1.

What about larger numbers? Does the term 5 AND 3 have meaning? It turns out that it does, although the answer in this case will not be 8, and it is now that we must be cautious not to let our daily use of the word "addition" be confused with our new meaning.

As we look at these numbers on a binary level, how to get the result of 5 AND 3 will be more obvious.

$$\begin{array}{r} x = 5 \quad 0101 \\ y = 3 \quad 0011 \\ \hline x \text{ AND } y \quad 0001 = 1 \end{array}$$

If we take the chart created earlier and apply it to each set of matching bits in x and y , we obtain the result shown. Starting on the left, two 0s give 0 as a result. For the next two bits, only a single 1 is present in each case, still giving 0 as a result. Only in the last position do we get the necessary 1s in bit 0 of *both* numbers to yield a 1 in the result.

Thus 5 AND 3 does have meaning, and the answer is 1. (Try that at parties!)

Don't be discouraged if you don't see the immediate value in this operation; you should guess by now that everything is good for something!

AND is used for a variety of purposes. These include:

- (1) To force zeros in certain bit positions.
- (2) As a mask to let through only ones in certain positions.

When an AND operation is done, the contents of the Accumulator are AND'd with another specified value. The result of this operation is then put back in the Accumulator. The other value may be either given by way of the immediate mode or held in a memory location. These are some possible ways of using AND:

```
LDA #$80
AND #$7F
AND $06
AND $300,X
AND ($06),Y
```

To understand better how AND is used, we should clarify some other ideas. One of these is the nature of assembly-language programs in general. I believe that, at any given point in a program, one of two kinds of work will be going on. One is the *operational mode*, where some specific task, such as clicking a speaker or reading a paddle, is taking place. At these moments, data as such does not exist. In the other case, the *processing mode*, data has been obtained from an operational mode and the information is processed and/or passed to some other routine or location in memory.

A given routine rarely is entirely in just one mode or the other, but any given step usually falls more into one category than the other.

These ideas are important because, in general, all of the logical operators are used during the processing phases of a program. At those times, some kind of data is being carried along in a register or memory location. Part of the processing that occurs is often done with the logical operators.

In the case of the two modes of use, operational and processing, we are really just talking about two different ways of looking at the same operation. To illustrate this, examine this partial disassembly of the Monitor starting at \$FDED:¹

```
*FDEDL
FDED- 6C 36 00    JMP  ($0036)
FDF0-  C9 A0     CMP  #$A0
FDF2-  90 02     BCC  $FDF6
FDF4-  25 32     AND  $32
FDF6-  84 35     STY  $35
FDF8-  48        PHA
FDF9-  20 78 FB   JSR  $FB78
FDFC-  68        PLA
FDFD-  A4 35     LDY  $35
FDFE-  60        RTS
```

For normal text output on the Apple, the Accumulator is loaded with the ASCII value for the character to be printed, the high bit is set, and a JMP to COUT (\$FDED) is done. From looking at the listing, you can see that at \$FDED there is an indirect jump based on the contents of \$36, \$37 (called a vector).

If this seems a little vague, then consider for a moment what I call the *flow of control* in the computer. This means that the computer is always executing a program somewhere. Even when there's nothing but a flashing cursor on the screen, the computer is still in a loop programmed to get a character from the keyboard. When you call your own routines, the computer is just temporarily leaving its own activities to do your tasks until it hits that last RTS. It then goes back to what it was doing before; usually, that's waiting for your next command.

When characters are printed to the screen, disk, printer, or anywhere else, there's a flow of control that carries along the character to be printed. For virtually every character printed, the 6502 scans through this region as it executes the code necessary to print the character.

Normally, \$36, \$37 points to \$FDF0 (at least before DOS is booted). This may seem a little absurd until you realize that a great deal of flexibility is created by the vector. For instance, a PR#1, such as you do when turning on a printer, redirects \$36, \$37 to point to the card, which in turn, after printing a character, usually returns to where \$36, \$37 used to print.

The card thus borrows the flow of control long enough to print the character, after which it gives control back to the screen print routine. Likewise, when DOS is booted, \$36, \$37 gets redirected from \$FDF0 to \$9EBD, which is where phrases preceded by a <CTRL>D are detected. If no <CTRL>D is found, the output is returned to \$FDF0.

Now, back to what AND is used for. Normally when the routine enters at \$FDF0, the Accumulator will hold a value between \$80 and \$DF². The characters

¹ [CT] This is for the Apple II Plus. Results on the Apple II or Apple //e will be different.

² [CT] Between \$80 and \$FF for computers with lowercase support.

from \$80 to \$9F are all control characters and are passed through by the BCC following the first CMP. Characters passing this test will be the usual alphabetic, numeric, and special characters shown in Appendix E. You'll notice at this point an AND with the contents of \$32 is done. Location \$32 is called INVFLG and usually holds either \$FF, \$7F, or \$3F depending on whether the computer is in the NORMAL, FLASHING, or INVERSE text mode. Let's assume that the Accumulator is holding the value for a normal A. Look at the following table to see what happens when an AND is done with each of these values.

Example 1:	Hex	Binary	ASCII
Accumulator:	\$C1	1100 0001	A
INVFLAG:	\$FF	1111 1111	-
Result:	\$C1	1100 0001	A

Example 2:	Hex	Binary	ASCII
Accumulator:	\$C1	1100 0001	A
INVFLAG:	\$7F	0111 1111	-
Result:	\$41	0100 0001	A (flashing)

Example 3:	Hex	Binary	ASCII
Accumulator:	\$C1	1100 0001	A
INVFLAG:	\$3F	0011 1111	-
Result:	\$01	0000 0001	A (inverse)

In the first example, ANDing with \$FF yields a result identical to the original value. The result is identical because, with each bit set to 1, the resulting bit will always come out the same as the corresponding bit in the Accumulator. (Can you guess what the result of ANDing with \$00 would always yield?) This means that the character comes out in its original form.

In the second case, ANDing with \$7F has the effect of forcing a 0 in bit 7 of the result. Examining the chart in Appendix E, we can see that \$41 corresponds to a flashing A.

The Apple uses the leading two bits to determine how to print the character. If the leading two bits are *off*, then the character will be in inverse. If bit 7 is 0 and bit 6 is 1, then the character will be printed in flashing mode. If bit 7 is set, then the character will be displayed in normal text.

Using the AND operator forces a 0 in the desired positions and lets the remaining bit pattern through.

In general, then, the way to use AND is to set a memory location (or the immediate value) equal to a value whose bits are all set to 1 except for those that you wish to force to 0 in the Accumulator.

You can also think of **AND** as acting rather like a screen that lets only certain parts of the image through. When **INVFLG** is set to **\$3F**, the leading bits will always be 0, regardless of whether they were set at entry; hence, the expression *mask*.

Sometimes figuring exactly what value you should use for the desired result is tricky. As a general formula, first decide what bits you want to force to 0 and then calculate the number with all other positions set to ones. This will give the proper value to use in the mask. For example, to derive the inverse display mask value:

1. Determine which bits to force to 0:

```
00xxxxxx
```

2. Calculate with the remaining positions set to ones:

```
00111111 = $3F (63)
```

Try this with the desired result of forcing only bit 7 to 0 and see if you get the proper value for **INVFLG** of **\$7F**.

Apple DOS Tool Kit users should note that when shifting the Accumulator, Apple's assembler requires the addition of the A operand (Example: **LSR A**). This applies to **ASL**, **LSR**, **ROR**, and **ROL**. Most other assemblers do not require the A operand, and that is the syntax used in this book.

BIT

The command somewhat related to **AND** is **BIT**. This is provided to allow the user to determine easily the status of specific bits in a given byte. When **BIT** is executed, quite a number of things happen. First of all, bits 6 and 7 of the memory location are transferred directly to the sign and overflow bits of the Status Register. Since we've not discussed the overflow flag, let me say briefly that its related commands, **BVC** and **BVS**, may be used just as **BPL** and **BMI** are used to test the status of the sign flag. If **V** (the overflow flag) is clear, **BVC** will succeed. If **V** is set, **BVS** will work.

Most important, though, is the conditioning of the zero flag. If one or more bits in the memory location match bits set in the Accumulator, the zero flag will be cleared (**Z** = 0). If no match is made, **Z** will be set (**Z** = 1). This is done by **AND**-ing the Accumulator and the memory location and conditioning **Z** appropriately. The confusing part is that this may seem somewhat 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.

Note that one of the main advantages of **BIT** is that the Accumulator is unaffected by the test.

Here are examples of how BIT might be used:

Example 1: To test for bits 0 and 2, set:

```
LDA  #05    ; 0000 0101
BIT  MEM
BNE  OK     ; (1 OR MORE BITS MATCH)
```

Example 2: To test for bit 7, set in memory:

```
CHECK BIT  $C000 ; (KEYBOARD)
BPL  CHECK ; (BIT 7 CLR, NO KEY PRESSED)
BIT  $C010 ; (ACCESS $C010 TO CLR STROBE)
```

If you want to test for *all* of a specific set of bits being on, the AND command must be used directly.

Example 3: To test for both bits 6 and 7 being on:

```
LDA  CHAR
AND  #$C0    ; 1100 0000
CMP  #$C0
BEQ  MATCH   ; BOTH BITS "ON"
```

This last example is somewhat subtle, in that the result in the Accumulator will only equal the value with which it was AND'd if each bit set to 1 in the test value has an equivalent bit on the Accumulator.

ORA and EOR

These last two commands bring up an interesting error of sorts in the English language, and that is the difference between an *inclusive OR* and the *exclusive OR*. What all this is about is the phenomenon that saying something like "I'll go to the store if it stops raining *or* if a bus comes by" has two possible interpretations. The first is that if either event happens, and even if both events occur, then the result will happen. This is called an *inclusive OR* 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 purest form of an *OR* statement. It is either night or day, but never both. This is 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. The table below shows the charts for both functions.

ORA	Accumulator		EOR	Accumulator	
	0	1		0	1
Memory 0	0	1	Memory 0	0	1
Memory 1	1	1	Memory 1	1	0

First, consider the table for ORA. If either or both corresponding bits in the Accumulator and the test value match, then the result will be a one. Only when neither bit is 1 does a 0 value result for that bit. The main use for ORA is to force a one at a given bit position. In this manner, it's something of the complement to the use of the AND operator to force zeros.

The following table presents some examples of the effect of the ORA command.

	Example 1:	Example 2:
Accumulator:	\$80 1000 0000	\$83 1000 0011
Value:	\$03 0000 0011	\$0A 0000 1010
Result:	\$83 1000 0011	\$8B 1000 1011

Use of ORA conditions the sign and zero flags, depending on the result, which is automatically put into the Accumulator.

The EOR command is somewhat different in that the bits in the result are set to 1 only if one or the other of the corresponding bits in the Accumulator and test value is 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 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. See the examples in the table below.

Accumulator:	\$80 1000 0000	\$83 1000 0011
Value:	\$03 0000 0011	\$0A 0000 1010
Result:	\$83 1000 0011	\$89 1000 1001
Accumulator:	\$83 1000 0011	\$89 1000 1001
Value:	\$03 0000 0011	\$0A 0000 1010
Result:	\$80 1000 0000	\$83 1000 0011

This flipping phenomenon is used extensively in hi-res graphics to allow one image to overlay another without destroying the image below. EOR also can be used to reverse specific bits: Simply place ones in the positions you wish to reverse.

You might find it quite rewarding to write your own experimental routine that will EOR certain ranges of memory with given values. Then make the second pass to verify that the data has been restored. This is especially interesting when done either on the hi-res screen or blocks of ASCII data such as on the text screen.

It would be a shame if you've stayed with this chapter long enough to read through all this and didn't get a program for your efforts, so I offer the demonstration program that follows. It provides a way of visually experimenting with the different shifts and logical operators. Assemble the assembly-language program listed and save it to disk under the name AL12.OPERATOR.

```

1 *****
2 * AL12-BINARY FUNCTION DISPLAY *
3 *           UTILITY           *
4 *****
5 *
6 *
7 *           OBJ   $300
8 *           ORG   $300
9 *
10 NUM       EQU   $06
11 MEM       EQU   $07
12 RSLT      EQU   $08
13 STAT      EQU   $09
14 *
15 YSAV1     EQU   $35
16 COUT1     EQU   $FDF0
17 CVID      EQU   $FDF9
18 COUT      EQU   $FDED
19 PRBYTE    EQU   $FDDA
20 *
21 *
0300: A9 00 22 OPERATOR LDA  #$00
0302: 48      23           PHA
0303: 28      24           PLP
0304: A5 06 25           LDA  NUM
0306: 25 07 26           AND  MEM           ; <= ALTER THIS
0308: 85 08 27           STA  RSLT
030A: 08      28           PHP
030B: 68      29           PLA
030C: 85 09 30           STA  STAT
030E: 60      31           RTS
32 *
030F: A9 A4 33 PRHEX   LDA  #$A4           ; '$'
0311: 20 ED FD 34           JSR  COUT
0314: A5 06 35           LDA  NUM
0316: 4C DA FD 36           JMP  PRBYTE
37 *
0319: A5 06 38 PRBIT    LDA  NUM
031B: A2 08 39           LDX  #$08
031D: 0A      40 TEST     ASL
031E: 90 0D 41           BCC  PZ
0320: 48      42 P0      PHA
0321: A9 B1 43           LDA  #$B1           ; '1'
0323: 20 ED FD 44           JSR  COUT
0326: A9 A0 45           LDA  #$A0           ; 'SPC'
0328: 20 ED FD 46           JSR  COUT
032B: B0 0B 47           BCS  NXT
48 *

```

```

032D: 48      49  PZ      PHA
032E: A9 B0   50      LDA #B0    ; '0'
0330: 20 ED FD 51      JSR COUT
0333: A9 A0   52      LDA #A0    ; 'SPC'
0335: 20 ED FD 53      JSR COUT
          54      *
0338: 68      55  NXT     PLA
0339: CA      56      DEX
033A: D0 E1   57      BNE TEST
          58      *
033C: 60      59  EXIT    RTS
          60      *
033D: EA      61      NOP
033E: EA      62      NOP
033F: EA      63      NOP
          64      *
0340: C9 80   65  CSHOW  CMP #80    ; STAND CHAR?
0342: 90 10   66      BCC CONT
0344: C9 8D   67      CMP #8D    ; <C/R>
0346: F0 0C   68      BEQ CONT
0348: C9 A0   69      CMP #A0    ; 'SPC'
034A: B0 08   70      BCS CONT
          71      *
034C: 48      72      PHA
034D: 84 35   73      STY YSAV1
034F: 29 7F   74      AND #7F    ; FORCE '0' IN BIT 7
0351: 4C F9 FD 75      JMP CVID
          76      *
0354: 4C F0 FD 77  CONT    JMP COUT1
          78      *
0357: 00      79  EOF     BRK
          80      *
          81      *
0358: 87      82      CHK

```

Then enter the accompanying Apple program and save it under the name AL12.OPERATOR.A.³

```

1  IF PEEK (768) <> 169 THEN PRINT CHR$ (4);"BLOAD AL12.OPERATOR,A$300"
2  REM IF DOS 3.3 THEN SET UP CSW VECTOR
3  IF PEEK(1002) = 76 THEN POKE 54,64: POKE 55,3: CALL 1002: GOTO 10
4  REM IF PRODOS, SET UP OUTPUT LINK AT $BE30,31
5  POKE 48688,64: POKE 48689,3
10 REM LOGICAL OPERATOR PROGRAM
15 OP = 774: F = 768: PH = 783: PB = 793
20 TEXT: HOME: GOTO 1000
100 KEY = PEEK ( -16384): IF KEY > 127 THEN 1000
110 A = PDL(0):A = PDL(0)
120 M = PDL(1):M = PDL(1)
125 POKE 6,A: POKE 7,M
130 CALL F: REM EVALUATE FUNCTION

```

³ [CT] Spaces and dashes were cleaned up to make the screen display more readable. In addition, for ProDOS we manually change the output vector at \$BE30, \$BE31 to point to CSHOW (\$340). See footnote 1 in chapter 29 for more discussion.

```

140 R = PEEK (8): S = PEEK (9)
200 VTAB 11: HTAB 1: PRINT "OPCODE:"; POKE 6,OC: GOSUB 500: VTAB 11: HTAB
32: PRINT " ";0$;" "
210 VTAB 14: PRINT "ACC:"; POKE 6,A: GOSUB 500: HTAB 30: PRINT " "; HTAB
30: PRINT CHR$ (A); VTAB 14: HTAB 33: PRINT "(P0)": POKE 1742,A: IF
A = 13 OR A = 141 THEN VTAB 14: HTAB 30: INVERSE : PRINT "M": NORMAL
215 IF 01 = 7 THEN VTAB 16: PRINT "MEMORY:"; POKE 6,M: GOSUB 500: HTAB
30: PRINT " "; HTAB 30: PRINT CHR$ (M); VTAB 16: HTAB 33: PRINT
"(P1)": POKE 1998,M: IF M = 13 OR M = 141 THEN VTAB 16: HTAB 30:
INVERSE : PRINT "M": NORMAL
220 IF 0$ < > "BIT" THEN VTAB 18: PRINT "RESULT:"; POKE 6,R: GOSUB 500:
HTAB 30: PRINT " "; HTAB 30: PRINT CHR$ (R): POKE 1270,R: IF R = 13
OR R = 141 THEN VTAB 18: HTAB 30: INVERSE : PRINT "M": NORMAL
230 VTAB 20: PRINT "STATUS:"; POKE 6,S: GOSUB 500: PRINT
240 VTAB 22: HTAB 10: PRINT "N V - B D I Z C"
250 GOTO 100
499 END
500 REM PRINT BITS & HEX
510 HTAB 10: CALL PB: HTAB 26: CALL PH: RETURN
1000 REM SELECT FUNCTION
1010 T = PEEK(-16368):FC = FC + 1 -(KEY = 136) * 2: IF FC > 8 THEN FC = 1
1011 IF KEY = 193 THEN FC = 1: REM 'A'=AND
1012 IF KEY = 194 THEN FC = 3: REM 'B'=BIT
1013 IF KEY = 197 THEN FC = 4: REM 'E'=EOR
1014 IF KEY = 204 THEN FC = 5: REM 'L'=LSR
1015 IF KEY = 207 THEN FC = 6: REM 'O'=ORA
1016 IF KEY = 210 THEN FC = 7: REM 'R'=ROL
1019 IF FC < 1 THEN FC = 8
1020 FOR I = 1 TO FC: READ 0$,OC,01: NEXT I: RESTORE
1025 IF KEY = 155 THEN PRINT CHR$ (4);"PR#0": END : REM <ESC>
1030 POKE OP,OC: POKE OP + 1,01: HOME
1050 ON FC GOSUB 1100,1200,1300,1400,1500,1600,1700,1800
1055 POKE 32,0
1060 A = -1: GOTO 100
1100 REM 'AND'
1110 POKE 32,9
1140 VTAB 2
1145 PRINT " AND 0 1 "
1150 PRINT " -----"
1155 PRINT " 0 ! 0 ! 0 !"
1160 PRINT " -----"
1165 PRINT " 1 ! 0 ! 1 !"
1170 PRINT " -----"
1175 PRINT : HTAB 7: PRINT "'AND'"
1180 VTAB 23: PRINT " ^ ^"
1185 RETURN
1200 REM 'ASL'
1220 VTAB 1: HTAB 9: PRINT "-----<==-----"
1225 HTAB 4: PRINT "----- 7 6 5 4 3 2 1 0 <-- '0'"
1230 HTAB 4: PRINT "! -----"
1235 HTAB 3: PRINT "----"
1240 HTAB 3: PRINT "!C!"
1245 HTAB 3: PRINT "----"
1250 VTAB 7:HTAB 16:PRINT "'ASL'": HTAB 8: PRINT "(ARITHMETIC SHIFT LEFT)"
1280 VTAB 23: HTAB 10: PRINT " ^ ^ ^"
1285 RETURN

```



```

1300 REM 'BIT'
1310 POKE 32,9
1340 VTAB 2
1345 PRINT "AND/BIT  0      1  "
1350 PRINT "      -----"
1355 PRINT "    0 ! 0 ! 0 !"
1360 PRINT "      -----"
1365 PRINT "    1 ! 0 ! 1 !"
1370 PRINT "      -----"
1375 PRINT : HTAB 7: PRINT "'BIT'"
1380 VTAB 23: PRINT "M M      ^": PRINT "7 6";
1385 RETURN
1400 REM 'EOR'
1410 POKE 32,9
1440 VTAB 2
1445 PRINT " EOR  0      1  "
1450 PRINT "      -----"
1455 PRINT "    0 ! 0 ! 1 !"
1460 PRINT "      -----"
1465 PRINT "    1 ! 1 ! 0 !"
1470 PRINT "      -----"
1475 PRINT : HTAB 7: PRINT "'EOR'"
1480 VTAB 23: PRINT " ^      ^"
1485 RETURN
1500 REM 'LSR'
1520 VTAB 1: HTAB 9: PRINT "-----==>-----"
1525 HTAB 2: PRINT "'0' --> 7 6 5 4 3 2 1 0 -----"
1530 VTAB 3: HTAB 9: PRINT "-----          !"
1535 HTAB 29: PRINT "----"
1540 HTAB 29: PRINT " !C!"
1545 HTAB 29: PRINT "----"
1550 VTAB 7: HTAB 15: PRINT "'LSR'": HTAB 8: PRINT "(LOGICAL SHIFT RIGHT)"
1580 VTAB 23: HTAB 10: PRINT "0      ^ ^"
1585 RETURN
1600 REM 'ORA'
1610 POKE 32,9
1640 VTAB 2
1645 PRINT " ORA  0      1  "
1650 PRINT "      -----"
1655 PRINT "    0 ! 0 ! 1 !"
1660 PRINT "      -----"
1665 PRINT "    1 ! 1 ! 1 !"
1670 PRINT "      -----"
1675 PRINT : HTAB 7: PRINT "'ORA'"
1680 VTAB 23: PRINT " ^      ^"
1685 RETURN
1700 REM 'ROL'
1720 VTAB 1: HTAB 9: PRINT "-----<==-----"
1725 HTAB 4: PRINT "<---- 7 6 5 4 3 2 1 0 <----"
1730 HTAB 4: PRINT " !  -----          !"
1735 HTAB 4: PRINT " !  ---          !"
1740 HTAB 4: PRINT "----->!C!-----"
1745 HTAB 16: PRINT "----"
1750 VTAB 8: HTAB 15: PRINT "'ROL'": HTAB 9: PRINT "(ROTATE ONE BIT LEFT)"
1780 VTAB 23: HTAB 10: PRINT " ^      ^ ^"
1785 RETURN

```

```

1800 REM 'ROR'
1820 VTAB 1: HTAB 9: PRINT "-----=>-----"
1825 HTAB 4: PRINT "----> 7 6 5 4 3 2 1 0 ---->"
1830 HTAB 4: PRINT "! ----- !"
1835 HTAB 4: PRINT "! --- !"
1840 HTAB 4: PRINT "----- C <-----"
1845 HTAB 16: PRINT "----"
1850 VTAB 8:HTAB 15: PRINT "'ROR': HTAB 9: PRINT "(ROTATE ONE BIT RIGHT)"
1880 VTAB 23: HTAB 10: PRINT "^ ^"
1885 RETURN
2000 DATA AND,37,7, ASL,10,234, BIT,36,7, EOR,69,7, LSR,74,234, ORA,5,7,
    ROL,42,234, ROR,106,234
32000 REM COPYRIGHT (C) 1981
32010 REM ROGER R. WAGNER

```

The basic theory of operation for the program is to rewrite locations \$306 and \$307 with the appropriate values to create the different operators. These values are contained in the data statement on line 2000 of the Applesoft program. In addition, there are routines to print the value in location \$06 in both binary and hex formats. Also, there is a routine to show control characters in inverse. You may wish to examine each of these to determine the logic, if any, behind their operation.

The Applesoft program itself operates by getting a value for the Accumulator and the memory location from paddles 0 and 1. The double reads in lines 110 and 120 minimize the interaction between the two paddles. Pressing any key advances the display to the next function; the left arrow backs up. Pressing A, B, E, L, O, or R will jump to the selected function.

The screen display shows the hex and binary values for each number and also what character would be printed via a PRINT CHR\$(X) statement (control

EOR	0	1	
0	!	0	!
1	!	1	!

'EOR'

```

OPCODE:  0 1 0 0 0 1 0 1 $45 'EOR'
ACC:     1 1 0 0 0 0 0 1 $C1 A (P0) A
MEMORY:  1 1 0 1 1 0 1 1 $DB C (P1) C
RESULT:  0 0 0 1 1 0 1 0 $1A Z      Z
STATUS:  0 0 1 0 0 0 0 0 $20
          N V - B D I Z C
          ^ ^

```

characters are shown in inverse). To the far right is the character obtained when the value is poked into the screen display part of memory.

I suppose if I were a purist the entire thing would have been written in assembly language. Oh well, maybe next time.

13

I/O Routines

October 1981

In chapter 11 I discussed how to access the disk using the RWTS routine. There is another way to read the disk that is more similar to the procedure used in BASIC. The advantage of this system is that we need not be concerned about what track and sector we're using, since DOS will handle the files just as it does in a *normal* program. The disadvantage is that we must have the equivalent of PRINT and INPUT statements to use in our programs to send and receive the data. So, before going any further, let's digress to input/output routines.

Print Routines

I have two favorite ways of simulating the PRINT statement. The first was described in earlier chapters and looks like this:

```
1 *****
2 *   AL13-DATA-TYPE PRINT 1   *
3 *****
4 *
5 *
6 *       OBJ   $300
7 *       ORG   $300
8 *
9 COUT    EQU   $FDED
10 *
0300: A2 00 11 ENTRY   LDX   #$00
0302: BD 0E 03 12 LOOP   LDA   DATA,X
0305: F0 06 13         BEQ   DONE
0307: 20 ED FD 14         JSR   COUT
030A: E8 15         INX
030B: D0 F5 16         BNE   LOOP
17 * (ALWAYS UP TO 255 CHRS)
18 *
030D: 60 19 DONE    RTS
20 *
030E: 84 21 DATA   HEX   84
030F: C3 C1 D4 22         ASC   "CATALOG"
0312: C1 CC CF C7
0316: 8D 00 23         HEX   8D00
24 *
0318: 00 25 EOF     BRK
```

This type of routine uses a defined data block to hold the ASCII values for the characters we wish to print. The printing is accomplished by loading the X-Register with \$00 and stepping through the data table until a \$00 is encountered. Each byte loaded is put into the Accumulator and printed via the JSR to COUT (\$FDED). When the \$00 is finally reached, the BEQ on line 13 is taken and we return from the routine via the RTS at DONE.

The new item of interest in this listing is the use of the \$84 as the first character printed. This will be printed as a <CTRL>D, and the word CATALOG that follows will be executed as a DOS command.

The essence of this chapter's message, along with the routines, is that any DOS command can be executed from assembly language exactly the same way it's done from BASIC. One need only precede the command with a <CTRL>D and terminate the command with a carriage return. Because DOS looks at all characters being output, it will see the <CTRL>D character and behave accordingly. (READ and WRITE are something of an exception to this technique but can still be done with only minor adjustments.)

You'll also notice the new assembler directive: ASC. This directive allows you to put an ASCII string directly into a listing without having to use the HEX command, which would necessitate a lot of mental conversions.

Try entering this program and then calling it with either a 300G from the Monitor or a CALL 768 from BASIC. Remember, the routine cannot be BRUN.

When running this program, you may notice a difference between a CALL 768 and the 300G. When called from the Monitor with the 300G, strange characters are printed out after the CATALOG is done. It is important to note here that any DOS command will overwrite the input buffer (\$200+) when executed. Because the Monitor expects to look for commands after your 300G, it maintains an internal pointer to which character in the input buffer it is currently evaluating. For example, it normally would be perfectly legal to execute the command: 300G 200.210.

The problem is, it wouldn't work with this program. Let's see why. When you enter 300G<RETURN>, the input buffer holds five characters: 3-0-0-G-<C/R>. When \$300 is called, the character pointer is at the <RETURN> character. When the DOS command CATALOG, is issued, the input buffer is overwritten with the characters ^D-C-A-T-A-L-O-G-<C/R>, where a ^D represents the <CTRL>D character. After the CATALOG, the Monitor will resume its interpretation of the input buffer on the fifth character, which now instead of the carriage return, is the second A of the word CATALOG. Thus, after the CATALOG command is done and control returns from the routine at \$300, you get the same result as if you had typed in ALOG, which would be to disassemble the code starting at location \$0A (AL), followed by a beep for a syntax error for OG<C/R>. To avoid this problem, routines that involve DOS commands should be called only from a running BASIC

program, or should exit via a `JMP $3D0`, as mentioned earlier in the section on the `COUT` routine.

This next print routine is more involved but does offer some advantages. One advantage is that the `HEX` or `ASC` data for what you want to print can immediately follow the `JSR` print statement, which parallels `BASIC` a little more closely and avoids construction of the various data blocks. The disadvantage is that the overall code is longer for short programs such as this. The general rule of thumb is to use the data-type print routine when you have only to print once or twice during the program, and to use the following type of routine when printing many times.

The logic behind the operation of this second method is slightly more complex than the previous routine, but I think you'll find it quite interesting.

Here's the new method:

```

1 *****
2 *   AL13-SPECIAL PRINT 2   *
3 *****
4 *
5 *           OBJ $300
6 *           ORG $300
7 *
8 PTR       EQU $06
9 COUT      EQU $FDED
10 *
0300: 20 0E 03 11 ENTRY   JSR PRINT
0303: 84      12 E0      HEX 84
0304: C3 C1 D4 13          ASC "CATALOG"
0307: C1 CC CF C7
030B: 8D 00   14          HEX 8D00
030D: 60     15 DONE    RTS
16 *
030E: 68     17 PRINT   PLA
030F: 85 06   18          STA PTR
0311: 68     19          PLA
0312: 85 07   20          STA PTR+1
0314: A0 01   21          LDY #$01           ; PTR HOLDS E0-1 HERE
22 *
0316: B1 06   23 P0      LDA (PTR),Y
0318: F0 06   24          BEQ FNSH
031A: 20 ED FD 25          JSR COUT
031D: C8     26          INY
031E: D0 F6   27          BNE P0           ; (MOST ALWAYS)
28 *
0320: 18     29 FNSH   CLC
0321: 98     30          TYA
0322: 65 06   31          ADC PTR
0324: 85 06   32          STA PTR
0326: A5 07   33          LDA PTR+1
0328: 69 00   34          ADC #$00
032A: 48     35          PHA
032B: A5 06   36          LDA PTR
032D: 48     37          PHA

```

```

032E: 60      38  EXIT    RTS
          39  * WILL  RTS TO DONE INSTEAD OF
          40  * E0!
          41  *

```

This one is rather interesting in that it uses the stack to determine where to start reading the data. You'll recall that when a JSR is done, the return address minus one is put on the stack. Upon entry to the PRINT routine, we use this fact to put that address in PTR, PTR+1. By loading the Y-Register with #E01 and indexing PTR to fetch the data, we can scan through the string to be printed until we encounter \$00, which indicates the end of the string.

When the end is reached, the BEQ FNSH will be taken. After that happens, the Y-Register (the length of the string printed) is transferred to the Accumulator and added to the address in PTR, PTR+1, and the result pushed back onto the stack. Remember that the old return address was E0-1 until it was pulled off.

Now when the RTS is encountered, the program will be fooled into returning to DONE instead of to E0 as it otherwise would have done.

To summarize, then:

1. Any DOS command can be executed from assembly language just as it is done in BASIC by doing the equivalent of printing a <CTRL>D followed by the command and a <RETURN>.
2. A data-type print routine uses ASCII characters in a labeled block, which is then called by name using the X-Register in a direct indexed addressing mode. The string to be printed should have the high bit set (ASCII value + \$80), and the string must be terminated by a 0 (at least when using the routine given here).
3. A JSR to a special print routine can also be done. In this case the ASCII data should immediately follow the JSR. Again, have the high bit set and end with \$00.

Input Routines

The other side of the coin is, of course, the INPUT routine. You might be surprised by the number of times I get calls from people saying, "If only the input in such-and-such program would accept quotes, commas, etc." The solution is actually quite simple and is presented here.

In its simplest form, the routine looks like this:

```

1  *****
2  * AL13-INPUT ROUTINE FOR BINARY*
3  *****
4  *
5  * STORES STRING AT PTR LOC
6  *
7  *          OBJ $300

```

```

      8          ORG  $300
      9          *
     10 GETLN    EQU  $FD6F
     11 BUFF     EQU  $200
     12 PTR      EQU  $06
     13          *
     14          *
0300: A2 00    15 ENTRY   LDX  #$00
0302: 20 6F FD 16        JSR  GETLN
     17          *
0305: 8A      18 CLEAR   TXA          ; X=LEN OF STRING
0306: A8      19        TAY
0307: A9 00    20        LDA  #$00
0309: 91 06    21        STA  (PTR),Y ; PUT END-OF-STRING MARKER
030B: 88      22        DEY          ; Y-1 FOR PROPER INDEXING
030C: B9 00 02 23 C2    LDA  BUFF,Y
030F: 29 7F    24        AND  #$7F   ; CLEAR HIGH BIT
0311: 91 06    25        STA  (PTR),Y ; PUT IN NEW LOC
0313: 88      26        DEY
0314: C0 FF    27        CPY  #$FF
0316: D0 F4    28        BNE  C2
     29          *
0318: 60      30 DONE   RTS

```

The heart of this routine is a call to the Monitor's GETLN routine, which gets a line of text from the keyboard or current input device and puts it in the keyboard buffer (\$200-\$2FF).

This saves our having to write a routine ourselves. The beauty of this method is also that all the <ESCAPE> and left/right arrow keys are recognized. When the routine returns from GETLN, the entered line is sitting at \$200+. The length is held in the X-Register.

At this point we presumably could just return from our routine as well but, as it happens, all the data now in the buffer has the high bit set—that is, #\$80 has been added to the ASCII value of each character. Because Applesoft in particular, and many other routines in general, don't expect this, the high bit should be cleared before returning. Also \$200+ will hold only one string at a time, so there should be some provision for relocating the string to some final destination.

Both are accomplished in the CLEAR section of this routine. First the length of the string is transferred via the TXA, TAY to the Y-Register. My preference is then to mark the end of the string. The subtle part here is that even though the Y-Register holds the length value, this actually points to the position immediately after the last character entered into the input buffer. For example, if you entered the word TEST, X would be returned as \$04. Now the characters TEST occupy bytes \$200-\$203. Thus when the length (\$04) is put in the Y-Register, STA \$200, Y will put a 0 in the fifth character position. Thus a DEY is then needed to get ready for the continuation to C2.

Next, C2 begins a loop that loads each character into the buffer, does an AND with #\$7F, and then stores the result at a location pointed to by PTR, PTR+1 plus the Y-Register offset.

The AND #\$7F has the effect of clearing the high bit by forcing bit 7 to 0.

The Y-Register is then decremented and the loop repeated until the DEY forces Y to an \$FF. This will indicate that the last value was \$00, and we have thus completed scanning the buffer.

This routine will work fine as long as you're willing to manage the string entirely by yourself once it gets to the PTR, PTR+1 location. As noble as it might be to write programs entirely in assembly language, I usually prefer to write in both Applesoft and assembly language. This is because unless speed is required, Applesoft does offer some advantages in terms of program clarity and ease of modification. After all, if there were no advantage to Applesoft, why would somebody have written it in the first place?

So, to that end, here are two new listings, the first in Applesoft:

```
5 PRINT CHR$(4);"BLOAD AL13.INPUTFP"
10 IN$ = "X"
20 PRINT "ENTER THE STRING: ";
30 CALL 768: IN$ = MID$(IN$,1)
40 IF IN$ = "END" THEN END
50 PRINT IN$: PRINT: GOTO 20
```

and the second in assembly language:

```
1 *****
2 * AL13-INPUT ROUTINE FP BASIC *
3 *****
4 *
5 * IN$="" MUST BE FIRST VARIABLE
6 * DEFINED IN PROGRAM!
7 *
8 * OBJ $300
9 * ORG $300
10 *
11 GETLN EQU $FD6F
12 VARTAB EQU $69
13 BUFF EQU $200
14 *
15 *
0300: A2 00 16 ENTRY LDX #$00
0302: 20 6F FD 17 JSR GETLN
0305: A0 02 18 LDY #$02
0307: 8A 19 TXA
0308: 91 69 20 STA (VARTAB),Y
21 * STORE 'X-REG = LEN OF IN$'
22 * IN LEN BYTE OF IN$
23 *
030A: C8 24 INY ; Y = 3
030B: A9 00 25 LDA #$00
030D: 91 69 26 STA (VARTAB),Y
```

```

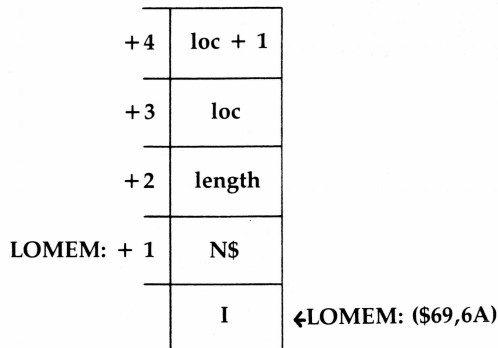
030F: C8      27          INY          ; Y = 4
0310: A9 02   28          LDA  #$02
0312: 91 69   29          STA  (VARTAB),Y
          30 * SET LOCATION PTR OF IN$ TO
          31 * $200 (INPUT BUFFER)
          32 *
0314: 8A      33 XFER      TXA
0315: A8      34          TAY          ; Y-REG = LEN NOW
0316: B9 00 02 35 X2      LDA  BUFF,Y
0319: 29 7F   36          AND  #$7F
031B: 99 00 02 37          STA  BUFF,Y
031E: 88      38          DEY
031F: C0 FF   39          CPY  #$FF
0321: D0 F3   40          BNE  X2
          41 *
0323: 60      42 DONE     RTS
0324: 62      43          CHK

```

The important difference to notice here is that IN\$ has been defined as the first variable in the Applesoft program, and that the assembly-language routine uses this fact to transfer the string to Applesoft.

The way this is done begins at XFER. When an Applesoft string variable is stored, the name, length, and location of the string are put in a table whose beginning is pointed to by locations \$69, \$6A (VARTAB, VARTAB+1).

Since IN\$ was the first variable defined, we know that its name and pointer will start at wherever VARTAB points. The name is held in positions \$00 and \$01, the length in \$02, and the location in \$03 and \$04.



By loading the Y-Register with #\$02, we can store the length of the entered string in the proper place. The location of IN\$ is then set to \$200 by putting the appropriate bytes into positions \$03 and \$04. Now Applesoft is temporarily fooled into thinking that IN\$ is at \$200—right where our input string is held!

The routine finishes by clearing the high bit, as before, and then returning with the RTS.

When the RTS is done, line 30 of the Applesoft program immediately assigns IN\$ to itself in such a way as to force Applesoft to move IN\$ from where it was in the input buffer to a new location up in its usual variable storage area. The net result can be obtained in various other ways besides the MID\$ statement, but the way shown is the least intrusive in terms of affecting other variables. (You could use A\$=IN\$: IN\$=A\$, but then you'd need a second variable in your program—no problem, just more names to keep track of.)

Make sure the input routine is loaded at \$300 before running the Applesoft program. Note that you can enter commas, quotes, <CTRL>C's, etc. Only entering END or pressing RESET should be able to exit this routine.

Reading and Writing Files on Disk

November 1981

Reading and Writing Data Files

This chapter will challenge your devotion to the cause of learning assembly-language programming. Up until now the source listings have been very short and easily typed in a few minutes' time. Unfortunately, the listings for this chapter are a bit longer than usual. But chin up! The result will be worth it! I've received quite a number of requests for information on how to read and write files on the disk. The programs listed will combine many of the techniques and routines you've learned so far into a single mini-database program.¹

The first program saves and loads the data by means of a simple BSAVE/BLOAD operation. This is fast and very straightforward. Here's the listing:

```

1  *****
2  * AL14-NAME FILE DEMO PROGRAM *
3  *****
4  *
5  *
6          ORG  $6000
7  *
8  HOME    EQU  $FC58
9  COUT    EQU  $FDED
10 RDKEY   EQU  $FD0C
11 GETLN   EQU  $FD75
12 BUFF    EQU  $200
13 VTAB    EQU  $FC22
14 CH      EQU  $24
15 CV      EQU  $25
16 CTR     EQU  $08
17 PTR     EQU  $06
18 REENTRY EQU  $3D0
19 *
20 *
6000: A9 00 21 ENTRY   LDA  #$00
6002: 85 06 22        STA  PTR

```

¹ [CT] These two programs will work only in DOS, not ProDOS. According to *Beneath Apple ProDOS* (Don Worth and Pieter Lechner, Quality Software, 1984, p. 6-61), <CTRL>D does not work with ProDOS commands from assembly code. Instead, you must place the command string in the GETLN input buffer at \$200 and then call the BASIC Interpreter (BI) at \$BE03. This is left as an exercise for the reader.

```

6004: A9 10 23 LDA #$10
6006: 85 07 24 STA PTR+1
6008: A9 B1 25 LDA #$B1
600A: 85 08 26 STA CTR
      27 *
600C: A0 00 28 CLR LDY #$00
600E: 91 06 29 STA (PTR),Y
6010: C8 30 INY
6011: A9 A0 31 LDA #$A0
6013: 91 06 32 STA (PTR),Y
6015: A9 00 33 LDA #$00
6017: C8 34 INY
6018: 91 06 35 STA (PTR),Y
601A: E6 07 36 INC PTR+1
601C: E6 08 37 INC CTR
601E: A5 08 38 LDA CTR
6020: C9 B6 39 CMP #$B6
6022: 90 E8 40 BCC CLR
      41 *
      42 * PUTS '#1-5,SPC,00' IN BUFFER
      43 *
6024: 20 58 FC 44 MENU JSR HOME
6027: A9 02 45 P1 LDA #$02
6029: 85 25 46 STA CV ; VTAB 3
602B: 20 22 FC 47 JSR VTAB
602E: 20 C2 61 48 JSR PRINT
6031: B1 A9 A0 49 ASC "1) INPUT NAMES"
6034: C9 CE D0 D5 D4 A0 CE C1
603C: CD C5 D3
603F: 8D 00 50 HEX 8D00
      51 *
6041: A9 04 52 P2 LDA #$04
6043: 85 25 53 STA CV
6045: 20 22 FC 54 JSR VTAB ; VTAB 5
6048: 20 C2 61 55 JSR PRINT
604B: B2 A9 A0 56 ASC "2) PRINT NAMES"
604E: D0 D2 C9 CE D4 A0 CE C1
6056: CD C5 D3
6059: 8D 00 57 HEX 8D00
      58 *
605B: A9 06 59 P3 LDA #$06
605D: 85 25 60 STA CV
605F: 20 22 FC 61 JSR VTAB ; VTAB 7
6062: 20 C2 61 62 JSR PRINT
6065: B3 A9 A0 63 ASC "3) SAVE NAMES"
6068: D3 C1 D6 C5 A0 CE C1 CD
6070: C5 D3
6072: 8D 00 64 HEX 8D00
      65 *
6074: A9 08 66 P4 LDA #$08
6076: 85 25 67 STA CV
6078: 20 22 FC 68 JSR VTAB ; VTAB 9
607B: 20 C2 61 69 JSR PRINT
607E: B4 A9 A0 70 ASC "4) LOAD NAMES"
6081: CC CF C1 C4 A0 CE C1 CD
6089: C5 D3

```

```

608B: 8D 00      71          HEX 8D00
        72      *
608D: A9 0A      73 P5        LDA #$0A
608F: 85 25      74          STA CV
6091: 20 22 FC    75          JSR VTAB      ; VTAB 11
6094: 20 C2 61   76          JSR PRINT
6097: B5 A9 A0   77          ASC "5) END PROGRAM"
609A: C5 CE C4 A0 D0 D2 CF C7
60A2: D2 C1 CD
60A5: 8D 00      78          HEX 8D00
        79      *
60A7: A9 0C      80 P6        LDA #$0C
60A9: 85 25      81          STA CV
60AB: 20 22 FC    82          JSR VTAB      ; VTAB 13
60AE: 20 C2 61   83          JSR PRINT
60B1: D7 C8 C9   84          ASC "WHICH DO YOU WANT? "
60B4: C3 C8 A0 C4 CF A0 D9 CF
60BC: D5 A0 D7 C1 CE D4 BF A0
60C4: 00          85          HEX 00
        86      *
60C5: 20 0C FD    87 M1        JSR RDKEY
60C8: C9 B1      88          CMP #$B1      ; '1'
60CA: D0 06      89          BNE M2
60CC: 20 FD 60   90          JSR INPUT
60CF: 4C 24 60   91          JMP MENU
60D2: C9 B2      92 M2        CMP #$B2      ; '2'
60D4: D0 09      93          BNE M3
60D6: 20 42 61   94          JSR DSPLY
60D9: 20 0C FD    95          JSR RDKEY
60DC: 4C 24 60   96          JMP MENU
60DF: C9 B3      97 M3        CMP #$B3      ; '3'
60E1: D0 06      98          BNE M4
60E3: 20 78 61   99          JSR SAVE
60E6: 4C 24 60   100         JMP MENU
60E9: C9 B4      101 M4       CMP #$B4      ; '4'
60EB: D0 06      102         BNE M5
60ED: 20 A0 61   103         JSR LOAD
60F0: 4C 24 60   104         JMP MENU
60F3: C9 B5      105 M5       CMP #$B5      ; '5'
60F5: D0 03      106         BNE M6
60F7: 4C D0 03   107         JMP REENTRY
60FA: 4C 24 60   108 M6       JMP MENU
        109      *
        110     *
60FD: 20 42 61   111 INPUT    JSR DSPLY      ; SHOW WHAT'S THERE
        112     *
6100: A9 00      113 I0       LDA #$00
6102: 85 06      114         STA PTR
6104: A9 10      115         LDA #$10
6106: 85 07      116         STA PTR+1    ; SET PTR=$1000
        117     *
6108: A9 00      118         LDA #$00
610A: 85 08      119         STA CTR
610C: 18         120 ILOOP    CLC
610D: A5 08      121         LDA CTR
610F: 65 08      122         ADC CTR

```

```

6111: 85 25      123      STA  CV
6113: 20 22 FC   124      JSR  VTAB
6116: A9 00      125      LDA  $$$00
6118: 85 24      126      STA  CH
611A: A8        127      TAY
611B: 20 29 61   128      JSR  IP
611E: E6 07      129      INC  PTR+1
6120: E6 08      130      INC  CTR
6122: A9 04      131      LDA  $$$04
6124: C5 08      132      CMP  CTR
6126: B0 E4      133      BCS  ILOOP      ; GET 5 NAMES
        134      *
6128: 60        135      IFIN  RTS
        136      *
6129: A2 00      137      IP    LDX  $$$00
612B: 20 75 FD   138      JSR  GETLN
612E: 8A        139      TXA
612F: F0 10      140      BEQ  IPFIN      ; EXIT IF <CR> ONLY
6131: A8        141      TAY
6132: A9 00      142      LDA  $$$00
6134: 99 00 02   143      STA  BUFF,Y
6137: B9 00 02   144      IPLOOP LDA  BUFF,Y
613A: 91 06      145      STA  (PTR),Y   ; MOVE DATA TO PTR
        146      *                ; BLOCK
613C: 88        147      DEY
613D: C0 FF      148      CPY  #$FF
613F: D0 F6      149      BNE  IPLOOP
6141: 60        150      IPFIN  RTS
        151      *
6142: 20 58 FC   152      DSPLY JSR  HOME
6145: A9 00      153      LDA  $$$00
6147: 85 08      154      STA  CTR
        155      *
6149: 85 06      156      STA  PTR
614B: A9 10      157      LDA  $$10
614D: 85 07      158      STA  PTR+1
614F: 18        159      D0    CLC
6150: A5 08      160      LDA  CTR
6152: 65 08      161      ADC  CTR
6154: 85 25      162      STA  CV          ; VTAB (2*CTR)+1
6156: 20 22 FC   163      JSR  VTAB
6159: A9 00      164      LDA  $$$00
615B: 85 24      165      STA  CH          ; HTAB 1
615D: A8        166      TAY
        167      *
615E: B1 06      168      D1    LDA  (PTR),Y
6160: F0 06      169      BEQ  D1FIN
6162: 20 ED FD   170      JSR  COUT
6165: C8        171      INY
6166: D0 F6      172      BNE  D1          ; (ALWAYS)
        173      *
6168: A9 8D      174      D1FIN LDA  $$8D
616A: 20 ED FD   175      JSR  COUT        ; END WITH <CR>
616D: E6 07      176      INC  PTR+1
616F: E6 08      177      INC  CTR
6171: A9 04      178      LDA  $$$04

```

```

6173: C5 08      179          CMP   CTR
6175: B0 D8      180          BCS   D0          ; PRINT 5 NAMES
        181      *
6177: 60         182 DSFIN   RTS
        183      *
        184      *
6178: A9 8D      185 SAVE    LDA   $$8D
617A: 20 ED FD   186          JSR   COUT          ; CLEAR OUTPUT BUFFER
617D: 20 C2 61   187 OPEN    JSR   PRINT
6180: 84         188          HEX   84
6181: C2 D3 C1   189          ASC   "BSAVE DEMOFILE,A$1000,L$500"
6184: D6 C5 A0 C4 C5 CD CF C6
618C: C9 CC C5 AC C1 A4 B1 B0
6194: B0 B0 AC CC A4 B5 B0 B0
619C: 8D 00      190          HEX   8D00
        191      *
619E: 60         192 SFIN    RTS
        193      *
        194      *
619F: A9 8D      195 LOAD    LDA   $$8D
61A1: 20 ED FD   196          JSR   COUT
        197      *
61A4: 20 C0 61   198          JSR   PRINT
61A7: 84         199          HEX   84
61A8: C2 CC CF   200          ASC   "BLOAD DEMOFILE,A$1000"
61AB: C1 C4 A0 C4 C5 CD CF C6
61B3: C9 CC C5 AC C1 A4 B1 B0
61BB: B0 B0
61BD: 8D 00      201          HEX   8D00
        202      *
61BF: 60         203          RTS
        204      *
        205      *
        206      *
61C0: 68         207 PRINT   PLA
61C1: 85 06      208          STA   PTR
61C3: 68         209          PLA
61C4: 85 07      210          STA   PTR+1
61C6: A0 01      211          LDY   $$01
61C8: B1 06      212 P0      LDA   (PTR),Y
61CA: F0 06      213          BEQ   PFIN
61CC: 20 ED FD   214          JSR   COUT
61CF: C8         215          INY
61D0: D0 F6      216          BNE   P0          ; (ALWAYS)
        217      *
61D2: 18         218 PFIN    CLC
61D3: 98         219          TYA
61D4: 65 06      220          ADC   PTR
61D6: 85 06      221          STA   PTR
61D8: A5 07      222          LDA   PTR+1
61DA: 69 00      223          ADC   $$00
61DC: 48         224          PHA
61DD: A5 06      225          LDA   PTR
61DF: 48         226          PHA
61E0: 60         227 EXIT    RTS
        228      *

```



```

                229 *
61E1: 00        230 EOF      BRK
                231 *
                232 *
61E2: 89        233          CHK

```

To understand how it works, consider these conditions:

Data will be stored in the area from \$1000-\$14FF. This area is called a *buffer*. A total of five strings will be stored, each beginning at an exact page boundary (\$1000, \$1100, \$1200, etc.). It is assumed that no string will be longer than 255 bytes—a fairly safe assumption since the INPUT routine won't allow this either.

A zero-page pointer (cleverly labeled PTR) will be used to control which range in the buffer is currently being accessed for a particular string.

The basic routines used to make the overall idea work are as follows:

1. An INPUT routine using the Monitor (\$FD6F=GETLN).²
2. A PRINT routine using a JSR and a stack manipulation. (Not the DATA type.)
3. A single-key input routine present in the Monitor used to get the command key (\$FD0C=RDKEY).
4. The execution of DOS commands from assembly language by preceding phrases with a <CTRL>D.

To use the program, call it directly from BASIC with a CALL 24576. A menu will appear with these choices:

- 1) INPUT NAMES
- 2) PRINT NAMES
- 3) SAVE NAMES
- 4) LOAD NAMES
- 5) END PROGRAM

To try the routine out, use option 1 to enter five sample names. Then use option 2 to view the data you've entered. You may then use option 3 to save the data as a binary file on a diskette. Then rerun the program, and verify that only the numbers 1 through 5 exist in the buffer (option 2). Then retrieve your data by using the LOAD command (option 4), and view again to confirm a successful load.

In detail, this is how the program works:

At entry, PTR is set to point to \$1000, where the name buffer begins. The Accumulator is then loaded with the ASCII value for the character 1, and the CLR routine entered.

² [CT] Technically, our program is using NXTCHAR (\$FD75) instead of GETLN (\$FD6A), to avoid printing out the prompt character.

CLR puts the characters 1 through 5 into each of the string spaces. Each digit is followed by a <SPACE>, and then a \$00. I used \$00 as an end-of-string marker, but the choice is somewhat arbitrary.

MENU clears the screen and presents the user with the available choices. Points of interest here are the VTAB operation and the PRINT routine. To VTAB to a given line from assembly language, one of the easiest ways is to load CV with the line you wish to go to, and then JSR to the Monitor's VTAB routine (\$FC22). Normally, we might also wish to either print a carriage return, or set CH to 0. Note that CV and CH are the computer's vertical and horizontal cursor position bytes, as used by the Monitor. You can always tell the cursor position by examining these bytes, and CH may be forced to a desired value to accomplish the same as an HTAB in BASIC.

The PRINT routine is the one described in chapter 13, and is useful because the JSR PRINT can be followed immediately with the data to print. This is more similar to the BASIC PRINT statement, and also avoids setting up a lot of specific data tables to do the printing.

Once the menu is printed on the screen, line 87 of the source file does the JSR to RDKEY. This gets the command key from the user, which is then tested by the M1 to M6 series of checks.

After calling RDKEY the keyboard value was returned in the Accumulator, and we can directly test to see which key was pressed. The key is then compared with each of the five desired responses. If no match is found, the program jumps back to MENU to repeat the display and command input. Other than RESET, option 5 is the only way to exit the program.

Let's examine the menu options:

If you enter 1, control is directed to the section labeled INPUT. The first thing done there is to JSR to DSPLY. At this point, it's necessary only to understand that DSPLY just clears the screen and shows the five strings currently in memory.

After DSPLY, PTR is initialized to point to the beginning of the buffer (\$1000), and the counter is set to 0. The main INPUT loop comes next. Here CTR is used to calculate what line (vertical position) to put the cursor on. (DSPLY used the same algorithm to display the current data.) After VTAB, the equivalent of HTAB is done, followed by the jump to the actual input routine, here labeled IP. This is the routine from the previous chapter that gets a line and then moves it to a location indicated by PTR.

There are a few subtle items in the IP routine that should be noted. The first is line 140. If <RETURN> alone is entered (i.e. no new data), the routine immediately returns without rewriting the old string. This is to allow editing of a single entry by skipping the entries not of interest. Try it to see how it works.

The second item is the characteristic of this particular input routine to put the trailing zero at the end of the line. This is done on lines 141-143.

When it returns from IP, the counter is incremented and checked to see if it exceeds #04. If not, ILOOP repeats until five strings have been input. After the fifth string is entered, the program returns to the menu.

If option 2 is entered, the DSPLY routine is called. The sole purpose of this section is to clear the screen and print the five names in memory. At entry to DSPLY, a JSR \$FC58 does a HOME and the CTR is initialized to 0. As in the INPUT section, CTR is then used to calculate the VTAB position to print each line.

D1 is the part that actually prints each line by scanning (and outputting through COUT) all of the bytes at each range indicated by PTR. Note that as a safety check, if a 0 did not happen to be present due to some other error, eventually the Y-Register will pass #FF and the program will fall through to DIFIN.

DIFIN provides an ending carriage return to the string and then increments CTR until all five strings have been printed.

The load/save operations are quite simple. Knowing where the buffer is located, the entire block is accessed by doing either a BLOAD or BSAVE. Remember that disk commands are done from assembly language just as they would be done from BASIC. The program need only output a <CTRL>D followed by a legal DOS command and a <RETURN>. Again the PRINT routine is used to facilitate this.

If option 5 is entered, then the JMP to the DOS BASIC entry vector is executed to end the program.

Reading and Writing Text Files

This second listing is basically a modification of the first program. If you wish, rather than retype the entire file, you can just edit the first listing to add lines 20–29 and 194–228.

```

1 *****
2 * AL14-NAME FILE DEMO PROGRAM 2*
3 *****
4 *
5 *
6 *      OBJ  $6000
7 *      ORG  $6000
8 *
9 HOME   EQU  $FC58
10 COUT  EQU  $FDED
11 RDKEY EQU  $FD0C
12 GETLN EQU  $FD75
13 BUFF  EQU  $200
14 VTAB  EQU  $FC22
15 CH    EQU  $24
16 CV    EQU  $25
17 CTR   EQU  $08
18 PTR   EQU  $06
19 *
20 PROMPT EQU  $33

```

```

21  CURLIN  EQU  $75
22  LANG   EQU  $AAB6
23  REENTRY EQU  $3D0
24  *
6000: A9 40 25  ENTRY   LDA  #$40
6002: 8D B6 AA 26          STA  LANG      ; LANG = FP
6005: 85 76 27          STA  CURLIN+1  ; RUNNING PROG
6007: A9 06 28          LDA  #$06
6009: 85 33 29          STA  PROMPT   ; NOT DIRECT MODE
600B: A9 00 30          LDA  #$00
600D: 85 06 31          STA  PTR
600F: A9 10 32          LDA  #$10
6011: 85 07 33          STA  PTR+1
6013: A9 B1 34          LDA  #$B1
6015: 85 08 35          STA  CTR
36  *
6017: A0 00 37  CLR     LDY  #$00
6019: 91 06 38          STA  (PTR),Y
601B: C8 39          INY
601C: A9 A0 40          LDA  #$A0
601E: 91 06 41          STA  (PTR),Y
6020: A9 00 42          LDA  #$00
6022: C8 43          INY
6023: 91 06 44          STA  (PTR),Y
6025: E6 07 45          INC  PTR+1
6027: E6 08 46          INC  CTR
6029: A5 08 47          LDA  CTR
602B: C9 B6 48          CMP  #$B6
602D: 90 E8 49          BCC  CLR
50  *
51  * PUTS '#1-5,SPC,00' IN BUFFER
52  *
602F: 20 58 FC 53  MENU   JSR  HOME
6032: A9 02 54  P1     LDA  #$02
6034: 85 25 55          STA  CV          ; VTAB 3
6036: 20 22 FC 56          JSR  VTAB
6039: 20 0A 62 57          JSR  PRINT
603C: B1 A9 A0 58          ASC  "1) INPUT NAMES"
603F: C9 CE D0 D5 D4 A0 CE C1
6047: CD C5 D3
604A: 8D 00 59          HEX  8D00
60  *
604C: A9 04 61  P2     LDA  #$04
604E: 85 25 62          STA  CV
6050: 20 22 FC 63          JSR  VTAB      ; VTAB 5
6053: 20 0A 62 64          JSR  PRINT
6056: B2 A9 A0 65          ASC  "2) PRINT NAMES"
6059: D0 D2 C9 CE D4 A0 CE C1
6061: CD C5 D3
6064: 8D 00 66          HEX  8D00
67  *
6066: A9 06 68  P3     LDA  #$06
6068: 85 25 69          STA  CV
606A: 20 22 FC 70          JSR  VTAB      ; VTAB 7
606D: 20 0A 62 71          JSR  PRINT
6070: B3 A9 A0 72          ASC  "3) SAVE NAMES"

```

```

6073: D3 C1 D6 C5 A0 CE C1 CD
607B: C5 D3
607D: 8D 00      73      HEX 8D00
           74      *
607F: A9 08      75      P4      LDA #08
6081: 85 25      76      STA CV
6083: 20 22 FC 77      JSR VTAB      ; VTAB 9
6086: 20 0A 62 78      JSR PRINT
6089: B4 A9 A0 79      ASC "4) LOAD NAMES"
608C: CC CF C1 C4 A0 CE C1 CD
6094: C5 D3
6096: 8D 00      80      HEX 8D00
           81      *
6098: A9 0A      82      P5      LDA #0A
609A: 85 25      83      STA CV
609C: 20 22 FC 84      JSR VTAB      ; VTAB 11
609F: 20 0A 62 85      JSR PRINT
60A2: B5 A9 A0 86      ASC "5) END PROGRAM"
60A5: C5 CE C4 A0 D0 D2 CF C7
60AD: D2 C1 CD
60B0: 8D 00      87      HEX 8D00
           88      *
60B2: A9 0C      89      P6      LDA #0C
60B4: 85 25      90      STA CV
60B6: 20 22 FC 91      JSR VTAB      ; VTAB 13
60B9: 20 0A 62 92      JSR PRINT
60BC: D7 C8 C9 93      ASC "WHICH DO YOU WANT ? "
60BF: C3 C8 A0 C4 CF A0 D9 CF
60C7: D5 A0 D7 C1 CE D4 BF A0
60CF: 00      94      HEX 00
           95      *
60D0: 20 0C FD 96      M1      JSR RDKEY
60D3: C9 B1      97      CMP #B1      ; '1'
60D5: D0 06      98      BNE M2
60D7: 20 08 61 99      JSR INPUT
60DA: 4C 2F 60 100     JMP MENU
60DD: C9 B2      101     M2      CMP #B2      ; '2'
60DF: D0 09      102     BNE M3
60E1: 20 4D 61 103     JSR DSPLY
60E4: 20 0C FD 104     JSR RDKEY
60E7: 4C 2F 60 105     JMP MENU
60EA: C9 B3      106     M3      CMP #B3      ; '3'
60EC: D0 06      107     BNE M4
60EE: 20 83 61 108     JSR SAVE
60F1: 4C 2F 60 109     JMP MENU
60F4: C9 B4      110     M4      CMP #B4      ; '4'
60F6: D0 06      111     BNE M5
60F8: 20 C7 61 112     JSR LOAD
60FB: 4C 2F 60 113     JMP MENU
60FE: C9 B5      114     M5      CMP #B5      ; '5'
6100: D0 03      115     BNE M6
6102: 4C D0 03 116     JMP REENTRY
6105: 4C 2F 60 117     M6      JMP MENU
           118     *
           119     *
6108: 20 4D 61 120     INPUT   JSR DSPLY      ; SHOW WHAT'S THERE

```

```

121 *
610B: A9 00 122 I0 LDA #000
610D: 85 06 123 STA PTR
610F: A9 10 124 LDA #10
6111: 85 07 125 STA PTR+1 ; SET PTR=$1000
126 *
6113: A9 00 127 LDA #000
6115: 85 08 128 STA CTR
6117: 18 129 ILOOP CLC
6118: A5 08 130 LDA CTR
611A: 65 08 131 ADC CTR
611C: 85 25 132 STA CV
611E: 20 22 FC 133 JSR VTAB
6121: A9 00 134 LDA #000
6123: 85 24 135 STA CH
6125: A8 136 TAY
6126: 20 34 61 137 JSR IP
6129: E6 07 138 INC PTR+1
612B: E6 08 139 INC CTR
612D: A9 04 140 LDA #04
612F: C5 08 141 CMP CTR
6131: B0 E4 142 BCS ILOOP ; GET 5 NAMES
143 *
6133: 60 144 IFIN RTS
145 *
6134: A2 00 146 IP LDX #000
6136: 20 75 FD 147 JSR GETLN
6139: 8A 148 TXA
613A: F0 10 149 BEQ IPFIN ; EXIT IF <CR> ONLY
613C: A8 150 TAY
613D: A9 00 151 LDA #000
613F: 99 00 02 152 STA BUFF,Y
6142: B9 00 02 153 IPLOOP LDA BUFF,Y
6145: 91 06 154 STA (PTR),Y ; MOVE DATA TO PTR
155 * ; BLOCK
6147: 88 156 DEY
6148: C0 FF 157 CPY #FF
614A: D0 F6 158 BNE IPLOOP
614C: 60 159 IPFIN RTS
160 *
614D: 20 58 FC 161 DSPLY JSR HOME
6150: A9 00 162 LDA #000
6152: 85 08 163 STA CTR
164 *
6154: 85 06 165 STA PTR
6156: A9 10 166 LDA #10
6158: 85 07 167 STA PTR+1
615A: 18 168 D0 CLC
615B: A5 08 169 LDA CTR
615D: 65 08 170 ADC CTR
615F: 85 25 171 STA CV ; VTAB (2*CTR)+1
6161: 20 22 FC 172 JSR VTAB
6164: A9 00 173 LDA #000
6166: 85 24 174 STA CH ; HTAB 1
6168: A8 175 TAY
176 *

```

```

6169: B1 06      177 D1      LDA (PTR),Y
616B: F0 06      178      BEQ D1FIN
616D: 20 ED FD   179      JSR COUT
6170: C8          180      INY
6171: D0 F6      181      BNE D1          ; (ALWAYS)
                182 *
6173: A9 8D      183 D1FIN   LDA $$8D
6175: 20 ED FD   184      JSR COUT          ; END WITH <CR>
6178: E6 07      185      INC PTR+1
617A: E6 08      186      INC CTR
617C: A9 04      187      LDA $$04
617E: C5 08      188      CMP CTR
6180: B0 D8      189      BCS D0          ; PRINT 5 NAMES
                190 *
6182: 60          191 DSFIN   RTS
                192 *
                193 *
6183: A9 8D      194 SAVE   LDA $$8D
6185: 20 ED FD   195      JSR COUT          ; CLEAR OUTPUT BUFFER
6188: 20 0A 62   196 OPENW JSR PRINT
618B: 84          197      HEX 84          ; <CTRL>D
618C: CF D0 C5   198      ASC "OPEN DEMOTEXTFILE"
618F: CE A0 C4 C5 CD CF D4 C5
6197: D8 D4 C6 C9 CC C5
619D: 8D 84      199      HEX 8D84
619F: D7 D2 C9 200 WRITE ASC "WRITE DEMOTEXTFILE"
61A2: D4 C5 A0 C4 C5 CD CF D4
61AA: C5 D8 D4 C6 C9 CC C5
61B1: 8D 00      201      HEX 8D00
                202 *
61B3: 20 4D 61   203 SVLOOP JSR DSPLY          ; PRINT NAMES TO DISK
                204 *
61B6: 20 02 62   205 CLOSEW JSR PRINT
61B9: 8D 84      206      HEX 8D84
61BB: C3 CC CF    207      ASC "CLOSE"
61BE: D3 C5
61C0: 8D 00      208      HEX 8D00
61C2: 60          209 SVFIN   RTS
                210 *
                211 *
61C3: A9 8D      212 LOAD   LDA $$8D
61C5: 20 ED FD   213      JSR COUT
                214 *
61C8: 20 02 62   215 OPENR  JSR PRINT
61CB: 84          216      HEX 84
61CC: CF D0 C5   217      ASC "OPEN DEMOTEXTFILE"
61CF: CE A0 C4 C5 CD CF D4 C5
61D7: D8 D4 C6 C9 CC C5
61DD: 8D 84      218      HEX 8D84
61DF: D2 C5 C1  219 READ  ASC "READ DEMOTEXTFILE"
61E2: C4 A0 C4 C5 CD CF D4 C5
61EA: D8 D4 C6 C9 CC C5
61F0: 8D 00      220      HEX 8D00
                221 *
61F2: 20 0B 61   222 RDLOOP JSR I0          ; GET NAMES FROM DISK
                223 *

```

```

61F5: 20 02 62 224 CLOSER JSR PRINT
61F8: 8D 84 225 HEX 8D84
61FA: C3 CC CF 226 ASC "CLOSE"
61FD: D3 C5
61FF: 8D 00 227 HEX 8D00
6201: 60 228 RDFIN RTS
        229 *
        230 *
        231 *
6202: 68 232 PRINT PLA
6203: 85 06 233 STA PTR
6205: 68 234 PLA
6206: 85 07 235 STA PTR+1
6208: A0 01 236 LDY #$01
620A: B1 06 237 P0 LDA (PTR),Y
620C: F0 06 238 BEQ PFIN
620E: 20 ED FD 239 JSR COUT
6211: C8 240 INY
6212: D0 F6 241 BNE P0 ; (ALWAYS)
        242 *
6214: 18 243 PFIN CLC
6215: 98 244 TYA
6216: 65 06 245 ADC PTR
6218: 85 06 246 STA PTR
621A: A5 07 247 LDA PTR+1
621C: 69 00 248 ADC #$00
621E: 48 249 PHA
621F: A5 06 250 LDA PTR
6221: 48 251 PHA
6222: 60 252 EXIT RTS
        253 *
        254 *
6223: 00 255 EOF BRK
        256 *
        257 *
6224: A1 258 CHK

```

The theory to this second program is fairly simple. If you think about it, the INPUT and DSPLY sections are essentially equivalent to a FOR I=1 TO 5/NEXT I type loop that respectively inputs and prints five strings. In a BASIC program, all that would be required to access a text file would be to precede the execution of those routines with the OPEN, READ and the OPEN, WRITE commands. (I'm assuming you're familiar with the normal access of Apple DOS text files. If not, read your manual!)

If you examine the new save and load routines you'll notice two changes. First, rather than printing BSAVE or BLOAD, the files are OPENed and the READ or WRITE command output. Notice that each command begins with a <CTRL>D and ends with a carriage return. Second, after the command is printed, a JSR is done to the IP or DSPLY routine as is appropriate. Last of all, a CLOSE is output before returning to the menu.

According to what we know so far, these should be the only changes necessary to access text files. There is one last catch though.

Apple DOS complicates things by not allowing the user to OPEN text files from the immediate mode. When a machine language program is running, DOS thinks we're still in the immediate mode and won't let us access the text files. What's needed is a way to fool DOS into thinking we're running a program.

This is done by using three internal management locations in the Apple. LANG (\$AAB6) is what DOS uses to keep track of which language is currently running. CURLIN (\$75, \$76) is Applesoft's register for the bytes of the program line number currently being executed. In the immediate mode, the high-order byte (\$76) defaults to #\$FF. Applesoft can tell if a program is running by looking for a non-#\$FF value in this location. The other way it knows a program is running is to check location \$33, which holds the ASCII value for the prompt character. In the immediate mode of Applesoft, this is #\$DD, equivalent to the '[' character. In a running program, this changes to #\$06.

To fool DOS, all we need to do is load these three locations appropriately at the beginning of the routine. Finally, when exiting the program, rather than using a simple RTS, the JMP \$3D0 is executed to do a soft reentry to BASIC. This will restore the bytes we've altered to fool DOS and also return us to the current language.³

Try these programs out. You'll find they make an excellent summary of many of the ideas and routines discussed so far, and they also provide a valuable model for your own programs.

³ Some people have also inquired as to whether the check for a write-protect label can be defeated by modifying DOS. The answer is yes and no. Yes, the part of the code that generates the error can be eliminated, but because the write-protect switch is physically wired into the recording head's write system, you cannot defeat it without actually removing or altering the switch itself.

Special Programming Techniques

December 1981

It has long been my feeling that it is not enough just to know an arbitrary selection of options or commands when using any tool, program, or programming language. Equally important are the techniques with which the options are combined to achieve the desired results.

With time and practice you will develop your own skills at creating efficient assembly-language routines, but that process can be assisted by examining the techniques that others have developed in previous programming efforts.

I have tried in this book to provide a reasonable mix of programming techniques, along with the usual ration of new commands.

Relocatable versus Non-relocatable Code

In chapter 13 I presented two print subroutines for the output of text to the screen or disk text file. The disadvantage of both routines is that they are not *relocatable*. To see what this means, consider the following program:

```

1 *****
2 *AL15-NON-RELOCATABLE PRINT DEMO
3 *****
4 *
5 *
6 *      OBJ  $300
7      ORG  $300
8  COUT   EQU  $FDED
9 *
10 *
0300: 20 0D 03 11  ENTRY   JSR  PRNT
12 *
0303: 4C 0C 03 13  DONE    JMP  EXIT
14 *
0306: D4 C5 D3 15  DATA   ASC  "TEST"
0309: D4
030A: 8D 00      16          HEX  8D00
17 *
030C: 60        18  EXIT    RTS
19 *
030D: A2 00     20  PRNT   LDX  #$00
030F: BD 06 03 21  LOOP   LDA  DATA,X
0312: F0 EF     22          BEQ  DONE
0314: 20 ED FD 23          JSR  COUT
0317: E8        24          INX

```

```

0318: D0 F5      25          BNE LOOP
031A: 60          26  FIN     RTS

```

This program, as written, can run only at the location specified by the `ORG` statement, in this case `$300`. Thus it is called *non-relocatable* code. Machine code becomes non-relocatable through the use of any statements which involve absolute addressing. The most common examples are the `JMP` and `JSR` commands, and the use of data statements, usually in print routines.

The first statement of this type occurs on line 11. The `JSR` to `PRNT` (`$30D`) will work only so long as `PRNT` is at `$30D`. If the routine were to be loaded into memory at `$400` (instead of `$300`), the routine would take the `JSR` to a block of nonexistent code at `$30D`.

Likewise, the `JMP` on line 13 has the same difficulty, as does the `DATA,X` statement on line 21. Any attempt to run the code at an address other than `$300` will result in disaster.

It should be noted, however, that not all `JSRs` and `JMPs` are universally troublesome. The `JSR COUT` (`$FDED`) will execute properly no matter where the object code is located since the reference is to a location outside of the object code block.

The general rule then is that any code which makes reference to absolute addresses within itself will not be relocatable, whereas code that does not suffer from this limitation can be run anywhere in memory.

The problem of relocatability may seem slight since any given routine is usually designed to be put at a definite location (usually either at `$300` or at the top of memory) and then protected via the Applesoft `HIMEM:` statement. However, as the number of routines you use increases, you will encounter more and more conflicts between routines originally written to occupy the same memory ranges. In addition, it also is occasionally desirable to directly append machine code to the end of Applesoft programs, where they will float up and down in memory at the end of the BASIC portion of the listing, being automatically moved as lines are added or deleted.

For these reasons, it is better in the long run to write code to run anywhere in memory when possible, thus avoiding future headaches about where to put everything.

The remainder of this chapter will discuss the various ways of avoiding the use of absolute addressing, thus creating code that can be used anywhere in memory regardless of the `ORG` statement used at assembly time.

JMP Commands

This is an example of a common use of the `JMP` command to jump over a range of memory, here represented by the fill section. At the destination, `EXPT`,

the BELL routine is called as a trivial example of where a subroutine might be executed.

```

1 *****
2 * AL15-NON-RELOCATABLE JMP DEMO*
3 *****
4 *
5 *
6 *          OBJ $300
7 *          ORG $300
8 BELL      EQU $FF3A
9 *
10 *
0300: 4C 04 03 11 ENTRY   JMP   EXPT
12 *
0303: EA      13 FILL    NOP
14 *
0304: 20 3A FF 15 EXPT    JSR   BELL
16 *
0307: 60      17 DONE    RTS

```

An alternative to this is the use of a forced branch statement, as shown in this example:

```

1 *****
2 * AL15-RELOCATABLE JMP 1 *
3 *****
4 *
5 *
6 *          OBJ $300
7 *          ORG $300
8 BELL      EQU $FF3A
9 *
10 *
0300: 18      11 ENTRY   CLC
0301: 90 01   12         BCC   EXPT
13 *
0303: EA      14 FILL    NOP
15 *
0304: 20 3A FF 16 EXPT    JSR   BELL
17 *
0307: 60      18 DONE    RTS

```

Notice that by clearing the carry and then immediately executing the BCC, the same result is obtained as when the JMP command was used in the earlier listing.

The main caution to observe is that the forced branch cannot be made over a distance of greater than 127 bytes, although most assemblers will give an error at assembly time if this is attempted. In addition, since the carry is cleared to force the branch, routines that set or clear the carry to indicate certain conditions may have compatibility problems with this approach.

Both limitations can be solved by slight modifications to this listing. The first is by using the overflow flag, often represented by a V. You should remember that the Status Register of the 6502 contains certain flags that are conditioned by various operations. These flags can be checked and appropriate responses can be made depending on their status. Examples of flags already covered are the carry (C) and zero (Z) flags.

The overflow flag is another bit in the Status Register which is set either by the BIT command (the overflow flag is set to bit 6 of the memory location), or by an ADC command. The overflow will be set whenever there is a carry from bit 6 to bit 7 as a result of an ADC operation.

These details are mentioned only in passing at this point, and you need not be concerned if it is not entirely clear. The main reason for bringing it up is that the overflow flag is used much more infrequently than the carry, and thus it is a slightly more desirable flag to use when creating a forced branch.

To make jumps over distances greater than 127 bytes, a *stepping* technique can be used. This is done by creating a series of the branch commands throughout the code to facilitate the program flow from one part to another. It is generally not too difficult to find breaks between routines to insert the branch statements required for the stepping action. Both techniques are illustrated here:

```

1 *****
2 *   AL15-RELOCATABLE JMP 2   *
3 *****
4 *
5 *
6 *           OBJ $300
7           ORG $300
8 BELL      EQU $FF3A
9 *
10 *
0300: B8   11 ENTRY      CLV
0301: 50 01 12           BVC STEP
13 *
0303: EA   14 FILL1     NOP
15 *
0304: 50 01 16 STEP     BVC EXPT
17 *
0306: EA   18 FILL2     NOP
19 *
0307: 20 3A FF 20 EXPT   JSR BELL
21 *
030A: 60   22 DONE     RTS

```

Although only one step is shown here, any number may be used, depending on what is needed to span the required distance.

Determining Code Location

Solving the JMP problem is only the beginning of the task. Very often it is important to know just where in memory the code is currently being run. One example of this is the code present on the disk controller cards. Since the card can be put in one of seven slots, and since each slot occupies a unique memory range, some technique is required to answer the question, “Where are we?”

```

1 *****
2 *          AL 15-LOCATOR 1          *
3 *****
4 *
5 *          OBJ  $300
6 *          ORG  $300
7 *
8 PTR      EQU  $06
9 RTRN     EQU  $FF58
10 STCK    EQU  $100
11 *
0300: 20 58 FF 12 ENTRY   JSR  RTRN
0303: BA      13         TSX
0304: BD 00 01 14         LDA  STCK,X
0307: 85 07   15         STA  PTR+1
0309: CA      16         DEX
030A: BD 00 01 17         LDA  STCK,X
030D: 85 06   18         STA  PTR
030F: 60      19  DONE   RTS

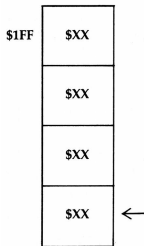
```

The success of this routine is based entirely on both the predictable nature of the stack and its function when a JSR is executed.

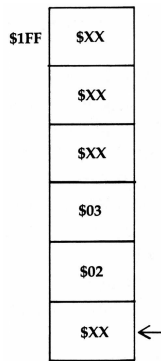
The stack was briefly described in chapter nine. At this point a little greater detail is necessary. The stack is a reserved part of memory from \$100 to \$1FF. It is used as a temporary holding buffer for various kinds of information required by the 6502 microprocessor. Information put on the stack is always retrieved in the opposite order from which it was deposited. This is often called LIFO (“Last-In First-Out”). The analogy of a stack of plates was used earlier, but the time has come to examine what actually occurs.

Whenever a JSR is done, the stack is used to hold the address to which the return should be made when the expected RTS is encountered. The diagrams on the next page illustrate this. Location \$FF58 is a simple RTS in the Monitor ROM which will be used to set up a *dummy* return address. Before the JSR, the Stack Pointer is set to some arbitrary position in the stack. Upon executing the JSR, the return address of \$302 is put on the stack and the Stack Pointer is decremented two bytes. Note that the stack stores the data from the top down, advancing the pointer as new data is added. When the RTS is encountered (immediately in the case of \$FF58), the Stack Pointer is returned to its original position and the return made.

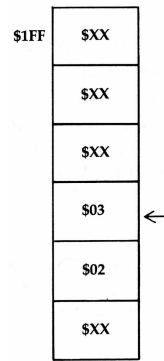
Before JSR \$FF58



During JSR \$FF58



After RTS from \$FF58



The arrow points to the current Stack Pointer S, which is a one-byte pointer to the next available position on the stack (not the last stored byte).

Note that the address stored, \$302, is the last byte of the JSR command—or, put another way, one byte less than the address of the next immediate command following the JSR.

Upon return from the JSR, the Stack Pointer is transferred to the X-Register with the TSX command on line 13. Because the Stack Pointer is at the next available byte on the stack, this will also point at the high-order byte of the return address still left in memory there. This is retrieved with the LDA STCK,X on line 15 and put in a temporary pointer location PTR+1 (\$07). The X-Register is then decremented and the low-order byte retrieved and put in PTR (\$06).

The final RTS of the routine returns control to the caller, at which point \$06, \$07 may be examined to verify the successful determination of the address \$302. You may wish to run this routine at several different locations in memory to verify that in each case PTR is properly set to ENTRY+2. What you have then is a short routine which can determine where in memory it is currently being run. The only disadvantage to this routine is that the high-order byte is retrieved first, thus complicating things if we want to add some offset value to the return address. The desirability of this will be shown shortly. In the meantime, consider this altered version of the Locator 1 routine:

```

1 *****
2 *      AL 15-LOCATOR 2      *
3 *****
4 *
5 *      OBJ $300
6 *      ORG $300
7 *
8 PTR    EQU $06
9 RTRN   EQU $FF58
10 STCK  EQU $100

```

```

11 *
0300: 20 58 FF 12 ENTRY JSR RTRN
0303: BA 13 TSX
0304: CA 14 DEX
0305: BD 00 01 15 LDA STCK,X
0308: 85 06 16 STA PTR
030A: E8 17 INX
030B: BD 00 01 18 LDA STCK,X
030E: 85 07 19 STA PTR+1
0310: 60 20 DONE RTS

```

What I've done here is decrement the X-Register (line 14) immediately after the TSX statement so that the low-order byte of the address can be retrieved first. The INX is then later used to go back and get the high-order byte. The advantage of this system is that it makes adding an offset much easier.

To show what we can now do, look at this revised print routine:

```

1 *****
2 * AL15-RELOCATABLE PRINT 1 *
3 *****
4 *
5 * OBJ $300
6 * ORG $300
7 *
8 PTR EQU $06
9 COUT EQU $FDED
10 RTRN EQU $FF58
11 STCK EQU $100
12 *
13 *
0300: 20 58 FF 14 ENTRY JSR RTRN
0303: B8 15 CLV
0304: 50 06 16 BVC CONT
17 *
0306: D4 C5 D3 18 DATA ASC "TEST"
0309: D4
030A: 8D 00 19 HEX 8D00
20 *
030C: BA 21 CONT TSX
030D: CA 22 DEX
030E: 18 23 CLC
030F: BD 00 01 24 LDA STCK,X
0312: 69 04 25 ADC #$04
0314: 85 06 26 STA PTR
0316: E8 27 INX
0317: BD 00 01 28 LDA STCK,X
031A: 69 00 29 ADC #$00
031C: 85 07 30 STA PTR+1
31 *
031E: A0 00 32 PRNT LDY #$00
0320: B1 06 33 LOOP LDA (PTR),Y
0322: F0 06 34 BEQ FIN
0324: 20 ED FD 35 JSR COUT
0327: C8 36 INY

```



```

0328: D0 F6    37          BNE  LOOP      ; ALWAYS UNTIL 255
          38      *
032A: 60      39      FIN      RTS
032B: 28      40          CHK

```

After calling the dummy return statement, a forced branch over the data section is done. This will have no effect on the address remaining on the stack. At CONT, we take the general procedure used in Locator 2 and add the CLC and ADC statements needed to add an offset to the address on the stack. What we need is the starting address of the ASCII data to be printed. Since the data starts at \$306 and the address on the stack is \$302 (see earlier examples) the offset needed is #\$04.

This may seem arbitrary but the value to add will always be #\$04 if you always do the CLV, BVC \$XXXX branch immediately after the return. Then follow that with the data to be printed.

Once the actual address of the ASCII data has been calculated, it is printed in the PRNT section by use of the indexed pointer at LOOP.

JSR Simulations

You might get the impression from the above example that a tremendous code expansion takes place to accomplish the relocatability of a program. This is somewhat true, but it depends on how you write the program. The use of CLV, BVC \$XXXX takes only three bytes where the JMP \$XXXX it was replacing also used three bytes.

The stack operations just discussed take a small number of bytes to implement but could become rather large if used many times. What is needed is a way to put the stack operations in a subroutine. Unfortunately, JSR is one of the non-relocatable commands.

```

1      *****
2      * AL15-NON-RELOCATABLE JSR DEMO*
3      *****
4      *
5      *          OBJ  $300
6      *          ORG  $300
7      *
8      BELL      EQU  $FF3A
9      *
10     *
0300: 20 04 03 11  ENTRY   JSR  TEST
12     *
0303: 60      13  DONE    RTS
14     *
0304: EA      15  TEST    NOP
16     *
0305: 20 3A FF 17  EXPT    JSR  BELL
18     *
0308: 60      19  FIN     RTS

```

```

20 *
21 * WILL RETURN TO DONE
22 *

```

This routine is very similar to the non-relocatable JMP demo presented earlier, with the exception that the call to the BELL routine has been made a subroutine itself, headed by the label TEST. In this listing, TEST is followed by a dummy NOP statement, but we'll fill that in shortly.

This program, as written, can run only at the address specified in the ORG statement. Here is an improved version, using a simulation of the JSR command:

```

1 *****
2 *AL15-RELOCATABLE JSR SIMULATION
3 *****
4 *
5 *           OBJ $300
6           ORG $300
7 *
8 PTR       EQU $06
9 BELL     EQU $FF3A
10 RTRN    EQU $FF58
11 STCK    EQU $100
12 *
13 *
0300: 20 58 FF 14 ENTRY   JSR  RTRN
0303: B8      15         CLV
0304: 50 01   16         BVC  TEST
17 *
0306: 60     18 DONE    RTS
19 *
0307: BA     20 TEST    TSX
0308: CA     21         DEX
0309: 18     22         CLC
030A: BD 00 01 23         LDA  STCK,X
030D: 69 03   24         ADC  #$03
030F: 85 06   25         STA  PTR
0311: E8     26         INX
0312: BD 00 01 27         LDA  STCK,X
0315: 69 00   28         ADC  #$00
0317: 85 07   29         STA  PTR+1
30 *
0319: 20 3A FF 31 EXPT    JSR  BELL
32 *
031C: A5 07   33 FIX    LDA  PTR+1
031E: 48     34         PHA
031F: A5 06   35         LDA  PTR
0321: 48     36         PHA
0322: 60     37 FIN    RTS
38 *
39 * WILL RETURN TO DONE
40 *

```

This program is very similar to the Print 1 program, with two exceptions. First, #\$03 is added instead of #\$04 to the address on the stack. This is a subtle

point worth mentioning, and you should review the listings until you feel comfortable with what is being done. Remember that the return address for a JSR/RTS is always *one less* than the address you want to return to. In the case of the DATA statement, we needed to know the exact address of the first character of the string to be printed. Hence the difference in the offset value used in each case.

Once the offset has been added and the proper return address calculated, the FIX section uses the PHA commands to put these on the stack. Thus when the RTS is encountered, the program returns to DONE. Notice that we have seemingly violated two general rules of assembly-language programming. The first is using the PHA commands without corresponding PLA statements, and the second is the use of an RTS without a calling JSR.

Upon further thought, however, it should become apparent that the two counteracted each other, and that an RTS is really equivalent to two PLAs.

The converse of this is using two PLAs within a routine called by a JSR to avoid returning to the calling address. This is equivalent to using a POP command in a BASIC subroutine called by a GOSUB.

Having thus simulated the JSR command, let's put it all together into a rewrite of the Print 1 routine that uses calls to subroutines to minimize the extra code required to make the routine relocatable:

```

1 *****
2 * AL15-RELOCATABLE PRINT 2 *
3 *****
4 *
5 * OBJ $300
6 * ORG $300
7 *
8 PTR EQU $06
9 COUT EQU $FDED
10 RTRN EQU $FF58
11 STCK EQU $100
12 *
13 *
0300: 20 58 FF 14 ENTRY JSR RTRN
0303: B8 15 CLV
0304: 50 15 16 BVC PRINT
17 *
0306: D4 C5 D3 18 DATA1 ASC "TEST1"
0309: D4 B1
030B: 8D 00 19 HEX 8D00
20 *
030D: 20 58 FF 21 L2 JSR RTRN
0310: B8 22 CLV
0311: 50 08 23 BVC PRINT
24 *
0313: D4 C5 D3 25 DATA2 ASC "TEST2"
0316: D4 B2
0318: 8D 00 26 HEX 8D00

```

```

27 *
031A: 60 28 DONE RTS
29 *
031B: BA 30 PRINT TSX
031C: CA 31 DEX
031D: 18 32 CLC
031E: BD 00 01 33 LDA STCK,X
0321: 69 04 34 ADC #$04
0323: 85 06 35 STA PTR
0325: E8 36 INX
0326: BD 00 01 37 LDA STCK,X
0329: 69 00 38 ADC #$00
032B: 85 07 39 STA PTR+1
40 *
032D: A0 00 41 PRNT LDY #$00
032F: B1 06 42 LOOP LDA (PTR),Y
0331: F0 06 43 BEQ FIX
0333: 20 ED FD 44 JSR COUT
0336: C8 45 INY
0337: D0 F6 46 BNE LOOP ; ALWAYS UNTIL 255
47 *
0339: 18 48 FIX CLC
033A: 98 49 TYA
033B: 65 06 50 ADC PTR
033D: 85 06 51 STA PTR
033F: A5 07 52 LDA PTR+1
0341: 69 00 53 ADC #$00
0343: 48 54 PHA
0344: A5 06 55 LDA PTR
0346: 48 56 PHA
0347: 60 57 FIN RTS
58 *
59 * WILL RTS TO L2/DONE
60 *
0348: AC 61 CHK

```

This routine has the advantage of allowing the PRINT statements to be used very much as though they were in the non-relocatable version given in chapter 13. The extra bytes required for the stack calculations are confined to one place, and there are only three extra bytes per line to be printed, compared to the chapter 13 routine.

The return to the end of each printed string is accomplished by using the Y-Register in FIX. At entry to FIX, the Y-Register will hold the length of the string printed, which is then added to PTR to calculate the proper address to return to. Again we use the two PHAs followed by an RTS to accomplish the return.

Self-Modifying Code

Ah, here is an area to make the strongest heart quiver—the idea that a program rewrites itself to accomplish its given task. The possibilities are endless, but for now we'll just look at a way of coping with statements like LDA \$ADDR,X. It

was this type of statement in the very first program of this chapter that contributed to its non-relocatability. Here's the new mystery program:

```

1 *****
2 * AL15-RELOCATABLE PRINT 3 *
3 *****
4 *
5 * OBJ $300
6 * ORG $300
7 *
8 PTR EQU $06
9 COUT EQU $FDED
10 RTRN EQU $FF58
11 STCK EQU $100
12 *
13 *
0300: 20 58 FF 14 ENTRY JSR RTRN
0303: B8 15 CLV
0304: 50 14 16 BVC PRINT
17 *
0306: D4 C5 D3 18 DATA ASC "TEST"
0309: D4
030A: 8D 00 19 HEX 8D00
20 *
030C: A2 00 21 PRNT LDX #$00
030E: BD 06 03 22 LOOP LDA DATA,X
0311: F0 06 23 BEQ DONE
0313: 20 ED FD 24 JSR COUT
0316: E8 25 INX
0317: D0 F5 26 BNE LOOP ; ALWAYS UNTIL 255
27 *
0319: 60 28 DONE RTS
29 *
031A: BA 30 PRINT TSX
031B: CA 31 DEX
031C: 18 32 CLC
031D: BD 00 01 33 LDA STCK,X
0320: 69 04 34 ADC #$04
0322: 85 06 35 STA PTR
0324: E8 36 INX
0325: BD 00 01 37 LDA STCK,X
0328: 69 00 38 ADC #$00
032A: 85 07 39 STA PTR+1
40 *
41 *
032C: A0 09 42 FIX LDY #$09 ; LEN OF $ + 5
032E: A5 06 43 LDA PTR
0330: 91 06 44 STA (PTR),Y
0332: C8 45 INY
0333: A5 07 46 LDA PTR+1
0335: 91 06 47 STA (PTR),Y ; REWRITE DATA ADDR
0337: B8 48 CLV
0338: 50 D2 49 BVC PRNT
50 *
033A: 4E 51 CHK

```

This program will actually rewrite the address present on line 22 for the LDA DATA,X statement. The method uses the address on the stack to calculate the address for the beginning of the ASCII string to be printed. It is this address that we will want eventually to put into the code at \$30F, \$310 to rewrite the data statement.

After calculating the address in lines 30–39, the result is stored in PTR. The FIX section then adds the length of the printed string plus five and uses this as the Y-Register offset to finally point to \$30F. The low- and high-order bytes are then written to the code and a return done to the actual PRNT routine.

This example comes with many cautions. The value on line 42 must be appropriate to the length of the string being printed. Also, the order of the ENTRY, DATA, and PRNT routines was deliberately chosen to make the rewrite as easy as possible. Extreme care must be taken whenever constructing a program that alters itself, but the results can be very powerful.

If you are inclined to pursue this, study this example well until you are very sure why each step was done. To verify its versatility, you should assemble the code for this example and then run it at several different memory locations. After each run, list the code from the Monitor and see how the statement on line 22 has been rewritten. It's really quite fascinating!

Indirect Jumps

To round out this chapter, one more technique will be discussed. Although the stepping method using forced branching can be used to span large distances, it can get rather inconvenient to have to keep inserting stepping points throughout your code. An alternate technique is to use the indirect JMP command.

In the indirect jump, a two-byte pointer is created which indicates where the jump should be made to. The added advantage of this command is that the pointer need not be created on the zero page, which already is in high demand for numerous other uses.¹ The basic syntax for the indirect jump is:

```
0300: 6C FF FF 99 J1 JMP ($FFF)
```

Here is a sample program showing how this can be combined with the stack operation to create a relocatable jump command:

```

1 *****
2 * AL15-RELOCATABLE JMP 3 *
3 *****
4 *
5 * OBJ $300
6 * ORG $300
7 *
8 PTR EQU $06
9 BELL EQU $FF3A
```

¹ [CT] See Appendix F for the list of available zero-page locations.

```

          10 RTRN    EQU  $FF58
          11 STCK   EQU  $100
          12 *
          13 *
0300: 20 58 FF 14 ENTRY  JSR  RTRN
          15 *
0303: BA      16 CALC   TSX
0304: CA      17        DEX
0305: 18      18        CLC
0306: BD 00 01 19        LDA  STCK,X
0309: 69 17   20        ADC  #$17
030B: 85 06   21        STA  PTR
030D: E8      22        INX
030E: BD 00 01 23        LDA  STCK,X
0311: 69 00   24        ADC  #$00
0313: 85 07   25        STA  PTR+1
0315: 6C 06 00 26        JMP  (PTR)      ; TO 'EXPT'
          27 *
0318: EA      28 FILL   NOP
          29 *
0319: 20 3A FF 30 EXPT   JSR  BELL
          31 *
031C: 60      32 DONE   RTS

```

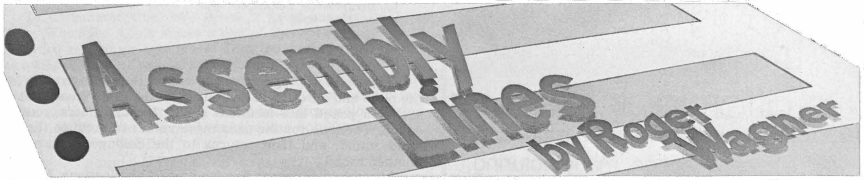
The system is fairly simple, basically just using the stack to get a base address and then adding whatever the distance is between the end of the JSR RTRN statement and the destination of the JMP(). As with some of the other systems, though, this distance will change as code is added or deleted between the two points. You may thus have to change the values on lines 20 and 24 rather frequently to keep up with your code changes.

However, it does avoid the problems associated with many stepping points sprinkled throughout your code, as would be necessary using the other alternative.

There is one bug in the use of the indirect jump that should be mentioned. It is present in the 6502 microprocessor itself, and occurs whenever the indirect pointer straddles a page boundary.² For example, if you used the statement JMP (\$06), the destination would be retrieved from locations \$06 and \$07. However, if you were to use JMP (\$3FF), the destination would be retrieved from \$3FF and \$300. The high-order byte is not properly incremented by the 6502. This is usually not a concern, though, since there are generally many alternate locations for the destination pointer.

In conclusion then, certain techniques can be used to produce code which is not restricted to running at a particular address in memory. Although a bit harder to construct initially, and slightly larger in terms of final memory requirements, the product is generally much more versatile in its applications.

² [CT] This bug was fixed in the 65C02.



Volume 2

Passing Data from Applesoft BASIC

January 1982

One useful application of assembly-language programming is in the enhancement of your existing Applesoft programs. Some people are inclined to write all their programs in assembly language, but it may be more efficient on occasion to write “hybrids”—programs that are a combination of Applesoft and assembly language. In this way, particular functions can be done by the language best suited to the particular task.

If you had to write a short program to store ten names, it would be best to do it in Applesoft:

```
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 neither speed nor program size is a concern, Applesoft is a completely acceptable solution.

However, if you had to sort a thousand names, speed would become a concern and it would be worth considering whether the job could best be done in assembly language.

If you have ever done a CALL in one of your BASIC programs, then you have already combined Applesoft with machine code. For example:

```
10 HOME
20 PRINT "THIS IS A TEST"
30 PRINT "THIS IS STILL A TEST"
40 GET A$
50 VTAB 1: HTAB 5: CALL -958
```

In this program, a line of text is printed on the screen. After you press a key, all text on the screen after the first word “THIS” is cleared.

Now although it might be possible to accomplish the same effect in Applesoft by printing many blank lines, it would not be as fast or as efficient in terms of code as the CALL -958.

In executing the above program, the Applesoft interpreter goes along carrying out your instructions until it reaches the CALL statement. At that point a JSR is done to the address indicated by the CALL. When the final RTS is encountered,

control returns to the BASIC program. In between, however, you can do anything you'd like!

CALLing routines is hardly complicated enough to warrant an entire chapter on the subject. The real questions are, how do you pass data back and forth between the two programs, and how can the problem of handling that data be made easier for the assembly-language program?

Simple Interfacing

The easiest way to pass data to an assembly-language routine is simply to POKE the appropriate values into unused memory locations and then retrieve them when you get to your assembly-language routine. To illustrate this, let's resurrect the tone routine from chapter eight.

To use this, assemble the code and place the final object code at \$300. Then enter the accompanying Applesoft program.

```

1 *****
2 *   AL 16-SOUND ROUTINE 3A   *
3 *****
4 *
5 *
6 *       OBJ  $300
7 *       ORG  $300
8 *
9 PITCH  EQU  $06
10 DURATION EQU  $07
11 SPKR   EQU  $C030
12 *
0300: A6 07 13 ENTRY  LDX  DURATION
0302: A4 06 14 LOOP   LDY  PITCH
0304: AD 30 C0 15     LDA  SPKR
0307: 88      16 DELAY DEY
0308: D0 FD 17     BNE  DELAY
030A: CA     18 DRTN  DEX
030B: D0 F5 19     BNE  LOOP
030D: 60     20 EXIT  RTS

```

This Applesoft program is used to call it:

```

10 INPUT "PITCH, DURATION? ";P,D
20 POKE 6,P: POKE 7,D
30 CALL 768
40 PRINT
50 GOTO 10

```

The Applesoft 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 CALLED. The tone routine uses these values to produce the desired sound and then returns to the CALLing program for another round.

This technique works fine for limited applications. Having to POKE all of the desired parameters into various corners of memory is not flexible, however, and strings are nearly impossible. There must be an alternative.

The Internal Structure of Applesoft

If you've been following along, you've no doubt figured out by now that I'm a great believer in using routines already present in the Apple, where possible, to accomplish a particular task. Since routines already exist in Applesoft for processing variables directly, why not use them?

To answer this, we must take a brief detour to outline how Applesoft actually "runs" a program. Consider this simple program:

```
10 HOME: PRINT "HELLO"
20 END
```

After you've entered this into the computer, typing LIST should reproduce the listing given here. 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.

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:

```
0067- 01
0068- 08
00AF- 18
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. The second pair is the program end pointer, and they show that it ends at \$818. Using this information let's examine the program data by typing in:

```
801L
```

You should get:

```
0801- 10 08      BPL  $080B
0803- 0A        ASL
0804- 00        BRK
0805- 97        ???
0806- 3A        ???
0807- BA        TSX
0808- 22        ???
0809- 48        PHA
080A- 45 4C     EOR  $4C
```

```

080C- 4C 4F 22 JMP $224F
080F- 00 BRK
0810- 16 08 ASL $08,X
0812- 14 ???
0813- 00 BRK
0814- 80 ???
0815- 00 BRK
0816- 00 BRK
0817- 00 BRK
0818- F9 A2 00 SBC $00A2,Y
081B- 86 FE STX $FE

```

This obviously is not directly executable code. Now type in:

```
801.818
```

This will give:

```

0801- 10 08 0A 00 97 3A BA
0808- 22 48 45 4C 4C 4F 22 00
0810- 16 08 14 00 80 00 00 00
0818- 8C

```

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*. Thus the word PRINT is stored as a \$BA. This does wonders for conserving memory. In addition, there is some overhead associated with packaging the line: a byte to signify the end of the line, a few bytes at the beginning of each line to hold information related to its length, and the line number itself.

To be more specific:

```

0801- 10 08 0A 00 97 3A BA
0808- 22 48 45 4C 4C 4F 22 00
0810- 16 08 14 00 80 00 00 00
0818- 8C

```

The first two bytes of every line of an Applesoft 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 that this is the end of the BASIC listing. It is important to realize that the \$00 00 end of the Applesoft program usually, *but not always*, corresponds to the contents of \$AF, \$B0. It is possible to hide machine-language code between the end of the line data and the actual end as indicated by \$AF, \$B0—but more on that later.

The next information within a line is the line number itself:

```

0801- 10 08 0A 00 97 3A BA
0808- 22 48 45 4C 4C 4F 22 00
0810- 16 08 14 00 80 00 00 00
0818- 8C

```

The $\$0A\ 00$ is the two-byte form of the decimal number 10, the line number of the first line of the Applesoft 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.

```
0801- 10 08 0A 00 97 3A BA
0808- 22 48 45 4C 4C 4F 22 00
0810- 16 08 14 00 80 00 00 00
0818- 8C
```

All bytes with a value of $\$80$ or greater are Applesoft keywords in token form. Bytes less than $\$80$ represent normal ASCII data (letters of the alphabet, for example). Examining the data here we see a $\$97$ followed by $\$3A$. $\$97$ is the token for HOME, and $\$3A$ the colon. Next, $\$BA$ is the token for PRINT. This is followed by the quote ($\$22$), 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 twenty, the $\$80$ is the token for END. As before, the line is terminated with a 00 .

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 6502 microprocessor and hence is much faster, since no table lookups are required.

In Applesoft, a syntax error is generated whenever a series of tokens is encountered that is not consistent with what the interpreter expects to find.

Passing Variables

So, back to the point of all this. The key to passing variables to your own assembly-language routines is to work with Applesoft in terms of routines already present in the machine. One of the simplest methods was described in chapter 13, wherein a given variable is the very first one defined in your program (see the input routine). This is okay, but rather restrictive. A better way is to name the variable you're dealing with right in the CALL statement.

The important points here are two components of the Applesoft interpreter: TXTPTR and CHRGET (and related routines).

TXTPTR is the two-byte pointer ($\$B8$, $\$B9$) that points to the next token to be analyzed. CHRGET ($\$B1$) is a very short routine that resides on the zero page and that reads a given token into the Accumulator. In addition to its occasionally being called directly, many other routines use CHRGET to process a string of data in an Applesoft program line.

Here then is the revised tone routine :

```

1 *****
2 *   AL 16-SOUND ROUTINE 3B   *
3 *****
4 *
5 *
6 *           OBJ  $300
7 *           ORG  $300
8 *
9 PITCH      EQU  $06
10 DURATION  EQU  $07
11 SPKR      EQU  $C030
12 *
13 COMBYTE   EQU  $E74C
14 *
0300: 20 4C E7 15 ENTRY      JSR  COMBYTE
0303: 86 06      16          STX  PITCH
0305: 20 4C E7 17          JSR  COMBYTE
0308: 86 07      18          STX  DURATION
19 *
030A: A6 07      20 BEGIN    LDX  DURATION
030C: A4 06      21 LOOP     LDY  PITCH
030E: AD 30 C0 22          LDA  SPKR
0311: 88          23 DELAY   DEY
0312: D0 FD      24          BNE  DELAY
0314: CA          25 DRTN    DEX
0315: D0 F5      26          BNE  LOOP
0317: 60          27 EXIT     RTS

```

The Applesoft calling program would then be revised to read:

```

10 INPUT "PITCH DURATION? ";P,D
20 CALL 768,P,D
30 PRINT
40 GOTO 10

```

This is a much more elegant way of passing the values and also requires no 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 the routine COMBYTE (\$E74C). This is an Applesoft routine which checks for a comma and then returns a value between \$00 and \$FF (0-255) in the X-Register.

It is normally used for evaluating POKEs, HCOLOR=, and so forth, but 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 by 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 will get a SYNTAX ERROR, since upon return Applesoft's

TXTPTR will be on the first comma after the CALL, and the phrase “,P,D” is not a legal Applesoft expression.

What about two-byte quantities? To deal with them, a number of other routines are used. For example, this routine will do the equivalent of a two-byte pointer POKE. Suppose for instance you wanted to store the bytes for the address \$9600 at locations \$1000, \$1001. Normally in Applesoft you would do it like this:

```
*
*
50 POKE 4096,0: POKE 4097,150
*
*
```

where 4096 and 4097 are the decimal equivalents of \$1000 and \$1001 and 0 and 150 are the low-order and high-order bytes for the address \$9600 (\$96 = 150, \$00 = 0).

A more convenient approach might be like this:

```
*
*
50 CALL 768, 4096, 38400
*
*
```

or perhaps:

```
*
*
50 CALL 768, A, V
*
*
```

The routine for this would be:

```
1 *****
2 * AL16-POINTER SETUP ROUTINE *
3 *****
4 *
5 *
6 *          OBJ $300
7          ORG $300
8 *
9 CHKCOM EQU $DEBE
10 FRMNUM EQU $DD67
11 GETADR EQU $E752
12 LINNUM EQU $50          ; ($50,$51)
13 *
14 PTR EQU $3C
15 *
0300: 20 BE DE 16 ENTRY JSR CHKCOM
0303: 20 67 DD 17 JSR FRMNUM ; EVAL FORMULA
0306: 20 52 E7 18 JSR GETADR ; PUT FAC INTO LINNUM
0309: A5 50 19 LDA LINNUM
```



```

030B: 85 3C    20          STA  PTR
030D: A5 51    21          LDA  LINNUM+1
030F: 85 3D    22          STA  PTR+1
                23      *
0311: 20 BE DE  24          JSR  CHKCOM
0314: 20 67 DD  25          JSR  FRMNUM
0317: 20 52 E7  26          JSR  GETADR
                27      *
031A: A0 00    28          LDY  #$00
031C: A5 50    29          LDA  LINNUM
031E: 91 3C    30          STA  (PTR),Y
0320: C8      31          INY
0321: A5 51    32          LDA  LINNUM+1
0323: 91 3C    33          STA  (PTR),Y
                34      *
0325: 60      35  DONE   RTS
0326: 09      36          CHK

```

The special items in this routine include CHKCOM, a syntax-checking routine that serves two purposes. First it verifies that a command follows the CALL address, and secondly it advances TXTPTR to point to the first byte of the expression immediately following the comma. If a comma is not found, a SYNTAX ERROR is generated.

FRMNUM is a routine that evaluates any expression and puts the real floating-point number result into Applesoft's *floating-point Accumulator*, usually called FAC. This is a six-byte pseudo register (\$97-\$9C) used to hold the floating-point representation of a number. It includes such niceties as the exponential magnitude of the number and the equivalent of the digits of the logarithm of the number stored.

At this stage you'd have to be something of a masochist to want to deal with the number in its current form, so the next step is used to convert it into a two-byte integer.

GETADR does this by putting the two-byte result into LINNUM, LINNUM+1 (\$50, \$51).

Even if this is not exactly an in-depth explanation of all the most precise details of the operation, the bottom line is that the three JSRs (CHKCOM, FRMNUM, and GETADR) will always result in the low-order and high-order bytes of whatever expression follows a comma being stored in LINNUM and LINNUM+1.

These simple subroutines should be quite adequate for many applications. Next chapter, however, we'll look at string passing, some other useful routines, and how to pass data back to the CALLing Applesoft program.

More Applesoft Data Passing

February 1982

In the previous chapter we began a discussion of how to pass variables back and forth between Applesoft and assembly-language programs. This chapter we'll complete the discussion with more information about how all types of variables are handled and how data can be passed back to the calling Applesoft program.

Applesoft Variables

There are six types of variables in Applesoft BASIC. These are *real*, *integer*, and *string* variables, and their array counterparts. To understand fully how to use these variables we must first take a moment to examine the differences between them as well as how the variables are actually stored in the computer.

Real variables are number values between 10^{38} and -10^{38} , which are very large positive and negative numbers. In addition, the values need not be whole numbers; a value such as 1.25 is allowed. *Integer* variables, on the other hand, are limited in magnitude to the range of -32767 to $+32767$. They are also limited to whole number values, such as 1, 2, 3, and so on. Values such as 1.25 are not allowed.

Real variables are indicated in BASIC by an alphabetic character (A to Z) followed by a letter or number (A to Z or 0 to 9). Any characters after the first two are ignored when Applesoft looks up the value for the variable. Integer variables are similar, but the name is suffixed by a percent sign (%). Thus A would represent the real variable, whereas A% would represent an integer variable.

When passing data such as a memory address or a single-byte value to put in memory, integer variables would be quite adequate and, additionally, would require no conversion in the assembly-language routine. However, it is generally more convenient to the BASIC programmer not to have to put the % sign in the variable name and, instead, to convert the value using the Applesoft routine FRMNUM (\$DD67) as described in the previous chapter. For the record, though, I will present an example shortly on how to retrieve an integer variable from a calling BASIC program.

String variables consist of a series of any legal ASCII characters, with a maximum length of 255 characters. Strings are indicated by a \$ suffix to the variable name.

Any of these variables may be present either singly or in an *array*. Arrays are groupings of variables that use a common name and then a delimiting *subscript* to identify each individual element. Array variables are indicated by a pair of parentheses following the variable name between which a number or expression may be used to specify the desired element.

You probably are already somewhat familiar with the general points mentioned so far; they're raised not so much to teach you about Applesoft variable types as such but rather to set the stage for what is to follow, namely how each of these variable types is stored within the memory of the Apple computer.

Memory Maps

In chapter one we presented a graphic representation of the memory usage of the computer. We'd like to revive the topic in the interest of our current subject.

A memory map is used to show the relative placement of data within the available memory locations in the computer. Recall that there are a total of 65536 locations available, which we identify with hexadecimal addresses of \$0000 to \$FFFF.

The chart in Table 17-1 shows a typical Apple memory map, with DOS booted and an arbitrary Applesoft program in memory.

In previous chapters, the areas shown have been described in varying degrees of detail. You'll recall that the area from \$C000 to \$CFFF is reserved for the interface card addressing, and that Applesoft BASIC is stored in ROM beginning at \$D000. The Monitor ROM begins at \$F800.

A normal Applesoft program starts at \$800, with the highest available address usually just below \$9600, which is identified with the lower boundary of the Disk Operating System (DOS).

The area from \$300 to \$3CF is available for user assembly-language programs. \$3D0 to \$3FF is reserved for Apple system vectors, such as the DOS entry vectors. Zero page, the stack, and the input buffer also have been discussed in some detail.

Since our main concern is in the area of Applesoft variables, let's consider a revised map emphasizing Applesoft programs.

Table 17-2 shows that when an Applesoft program is RUN, simple (non-array) variables are placed immediately after the end of the BASIC program, fol-

\$00	\$100	\$200	\$300	\$400	\$800	...	\$9600	\$C000	\$D000	\$F800
Zero Page	Stack	Input Buffer	User Page	Screen Display	FP Program	Free	DOS	Slots	FP BASIC	F8 ROM

Table 17-1: Apple Memory Map

\$00	\$800	\$XX	\$XX	\$XX	\$XX	\$9600
	FP Program	Simple Variables	Array Variables	Free	String Data	DOS
	\$67, \$68-\$AF, \$B0	\$69, \$6A LOMEM:	\$6B, \$6C	\$6D, \$6E	\$6F, \$70	\$73, \$74 HIMEM:

Table 17-2: Applesoft Memory Layout

lowed by the array variables. Because the data for each string variable is ever-changing in length, string data is stored dynamically at the top of memory, working down. The space in between these converging areas is the so-called free space of the system.

HIMEM: and LOMEM: are used by the BASIC programmer to set the upper and lower bounds of variable storage. If not specifically declared within the program, these default to the bottom of DOS and the end of the Applesoft program, respectively. They *do not*, however, always have to be restricted to these locations. It is possible to move LOMEM: up, or HIMEM: down, so as to set aside a portion of memory in the computer that won't be affected by the running program. This is done for one or both of two reasons: first to protect either or both of the hi-res display pages from variable table encroachment; or, second, to provide a protected area for a user's assembly-language program.

Now that we know where the information for each variable is stored in the computer, let's examine the format of the information for each variable. Within the areas indicated, a variable table is constructed that contains both the name of the given variable and its value if the variable is a real or integer. If the variable is a string, a pointer is stored that indicates the string's starting location at the top of memory and its corresponding length (0 to 255 characters).

Figure 17-3 summarizes the details of the format for these tables.

Each time a variable is first encountered in a running Applesoft program, an entry is made for it in the variable table. For simple variables, Applesoft looks to the pointer at \$6B, \$6C to see where the end of the current simple variable table is. It then opens up seven bytes for the new variable and puts a block of data similar to that shown in Figure 17-3, as is appropriate to the type of variable defined.

Real variables store the value in a logarithmic form, where each value is indicated by the exponent and four mantissas. Integer variables require only that the high- and low-order bytes of the value be stored. The remaining three positions are unused, with dummy 0 values placed in the table. It's important to note here that for integer variables, the two-byte representation of the value is reversed from what we would normally expect. That is, the high-order byte is placed first, followed by the low-order byte.

Assembly Lines

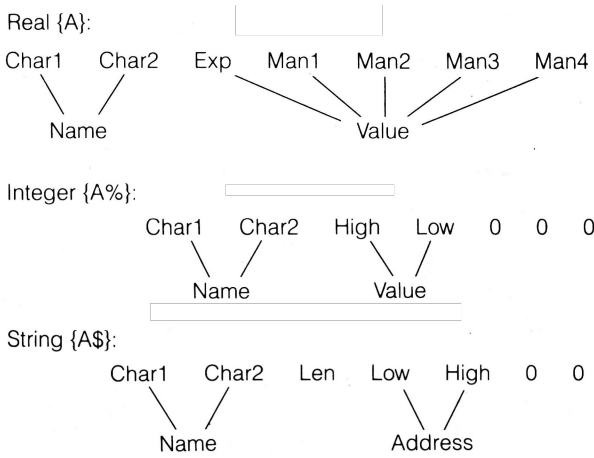


Figure 17-3: Simple Variable Storage Format

For strings only three bytes of information are required, namely the length and address data mentioned earlier. Again, the last two positions are filled with dummy zeros.

It should be evident from this table that the same amount of memory is allocated for all simple variable types: there is no advantage in specifying integer variables versus reals to save memory. This will not be the case with arrays.

Notice that there are two distinct parts to each seven-byte variable entry. The first two bytes define the name, where, incidentally, the high-order byte is used in each character to indicate to which of the three variable types (real, integer, or string) that entry corresponds. The last five bytes make up the actual data for each variable and consist of either the required numeric information or, in the case of a string, the length and address information.

The reason to stress this distinction is that, in examining arrays, we notice that it is this five-byte block that gets repeated a large number of times, depending on the total number of elements in the array. For arrays, a much larger table needs to be constructed, and this is created starting at the address indicated by \$6B, \$6C. Whenever a new array is defined, the pointer at \$6D, \$6E is examined to determine the end of the current array table and a new entry is made according to the format shown in Figure 17-4.

In this format, the entry is given a header that lists the variable name, followed by an offset value used to determine the address of the next array entry if one is present. The offset is encoded in the usual two-byte manner. Following the offset is a byte indicating the number of dimensions in the array, after which is listed a byte for each dimension stating its size. Although not shown in the dia-

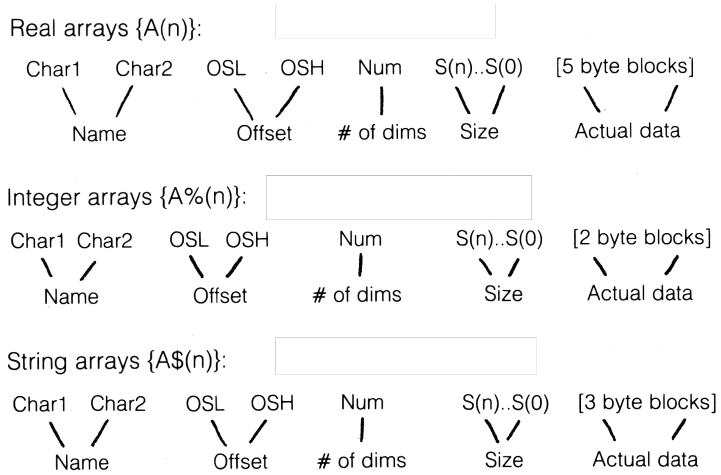


Figure 17-4: Array Variable Storage Format

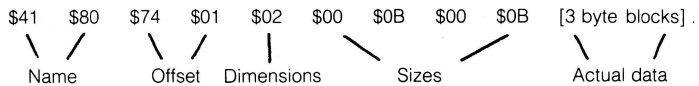
gram, each size indicator is a two-byte pair, although in this case the high byte is always given first.

Immediately after the header are the actual data blocks, each block consisting of five, two, or three bytes per array element, depending on which variable type is involved. Note that, in this case, integer variable arrays do take much less memory than an equivalent real array.

As an example, if you were to dimension an array with this statement:

```
DIM A$(10, 10)
```

the header block would look like this:



where \$41, \$80 are the ASCII values for an A followed by a null. The high bit is off in the first character, and on in the second—indicating a string. The next array variable would be found at the address of the first name character plus \$174. There are two dimensions to the array, as indicated by the \$02. The \$00 \$0B indicates *eleven* elements in each dimension of the array. This should not be surprising when you recall that ten plus the zeroth position makes eleven elements.

Following this header we would find 121 three-byte blocks, each indicating the length and address of a string array element, if present. $11 \times 11 = 121$; $(121 \times 3) + 9$ [for the header] = 372 = \$174.

Passing Variables to Assembly Language

At this point you may well think that we have strayed very far from the topic of assembly-language programming and have become overly involved with the structure of Applesoft. Upon a little reflection, however, it should become apparent that we must have some familiarity with how these variables are stored if we are to interact successfully with them.

In either reading or creating Applesoft variables, clearly we must handle effectively each component of the data. We must be able to identify the name and location of the variable we are interested in, and also to modify that information if necessary.

The temptation at this point might be to take this new-found knowledge and write our own routines to accomplish the needed operations, but such an undertaking would be quite unnecessary—not to mention likely to have you mindlessly babbling to yourself in no time. Fortunately, Applesoft already contains the routines necessary to do almost anything we wish. The main trick will be to properly identify and use the appropriate ones.

In the previous chapter I made use of a few of these to accomplish a certain degree of flexibility in passing numeric data to an assembly-language routine. Let's complete the study by formalizing the possible operations.

The first general category is passing data to a routine. We can pass any of six variable types. To minimize the confusion, let us establish a fairly simple goal: to pass the data successfully and prove so by storing the data in a non-Applesoft location.

Integer Variables

For integer variables the calling Applesoft program looks like this:¹

```
10 A% = 258
20 CALL 768, A%
30 PRINT PEEK(896), PEEK(897)
40 REM 896,897 = $380,$381
50 END
```

The machine-language routine should be assembled from this listing:

```
1 *****
2 * AL17-INTEGGER VARIABLE *
3 * READER *
4 * 2/1/82 *
5 *****
```

¹ [CT] For a more interactive program, replace lines 10–30 with the following:

```
5 PRINT CHR$(4); "BLOAD AL17.READINT"
10 INPUT "INPUT INTEGER: "; A%
20 CALL 768,A%
30 PRINT "LO: "; PEEK (896); " HI: "; PEEK (897)
```

Then try values such as 258, 1, -1, 32767, and -32767. Try -32768.

```

6      *
7      *
8      *          OBJ  $300
9      *          ORG  $300
10     *
11     CHKCOM EQU  $DEBE
12     PTRGET EQU  $DFE3
13     VARPNT EQU  $83
14     MOVFM  EQU  $EAF9
15     CHKNUM EQU  $DD6A
16     DATA  EQU  $380
17     *
0300: 20 BE DE 18     ENTRY   JSR  CHKCOM   ; CHK SYNTAX
0303: 20 E3 DF 19           JSR  PTRGET   ; FIND VARIABLE
20     * Y,A = ADDRESS OF VALUE
0306: 20 F9 EA 21           JSR  MOVFM   ; MOV VAL -> FAC
0309: 20 6A DD 22           JSR  CHKNUM  ; FAC = NUM?
030C: A0 00 23           LDY  #$00
030E: B1 83 24           LDA  (VARPNT),Y
0310: 8D 81 03 25           STA  DATA+1
0313: C8 26           INY
0314: B1 83 27           LDA  (VARPNT),Y
0316: 8D 80 03 28           STA  DATA
29     *
30     * NOTE! HIGH BYTE FIRST!
31     *
0319: 60 32     DONE   RTS
031A: F1 33           CHK

```

In this routine, CHKCOM (\$DEBE = CHecK for COMma) is used to make sure the syntax is correct (that is, a comma), and to advance TXTPTR (\$B8 = TeXT PoiNTeR) to the first byte of the variable name being evaluated. Refer to the previous chapter for a discussion of these two routines.

PTRGET (\$DFE3 = PoiNTeR GET) is now called, which is a subroutine that reads in a variable name and then locates it in the variable table. As a bonus, if the variable named does not already exist in the table, PTRGET will create an entry for it. This applies to variables of all six types. After returning from PTRGET, the address of the value for the variable is held in the Y-Register and the Accumulator (low byte, high byte). This thus indicates the location in memory of the two-to-five byte data block discussed earlier. The data in the Y-Register and the Accumulator is also duplicated in VARPNT, VARPNT+1 (\$83, \$84 = VARIable PoiNTeR), which will be used later in the program.

At this stage it would be a simple matter to use indirect addressing to retrieve the two bytes, but a little more effort will result in a much more thorough routine. It is possible that the user might have called the routine with an improper variable type following the CALL statement, such as a string. This can be checked for by the next two program steps.

MOVFM (\$EAF9 = MOVE to FAC from Memory) will move whatever data is pointed to by the Y-Register and the Accumulator into the floating-point Accu-

mulator (\$F9-A2 = FAC). The contents then can be checked for variable type by the call to CHKNUM (\$DD6A = CHeCK NUMber). The presence of a string here would yield a TYPE MISMATCH error.² Unfortunately, it is not particularly easy to test for a real variable having been mistakenly used here.

Presuming no error occurs, we will now make use of the data saved in VARPNT (since the Y-Register and Accumulator no doubt have been altered by MOVFM and CHKNUM) to actually retrieve the two-byte value passed. The indirect addressing mode is used to move the variable data into our two data bytes. The address of \$380, \$381 was arbitrarily chosen for this example.

It is important to note that special care is used in lines 25 and 28, since integer variables store the two data bytes high-order first, as mentioned earlier. This is opposite to the normal 6502 convention.

This routine will work equally well for retrieving data from simple integer variables and from integer array variables.

When you run this example, the numbers 2 and 1 should be printed out, these being the low- and high-order bytes of the number passed to the routine (258 = \$102).

Real Variables

Once in assembly language, the handling of floating-point numbers, such as represented by real variables, is somewhat involved. Additionally, the majority of the time you will be concerned only with passing an integer between 0 and 65535. Therefore, we will consider here how to use a real variable to pass a number in this range to a given subroutine.

This revision of our earlier Applesoft program will do the trick:

```
10 A = 258
20 CALL 768, A
30 PRINT PEEK(896), PEEK(897)
40 REM 896,897 = $380,$381
50 END
```

The assembly-language program for this is:

```
1 *****
2 *      AL17-REAL VARIABLE *
3 *      READER *
4 *      2/1/82 *
5 *****
6 *
7 *
8 *      OBJ $300
9 *      ORG $300
10 *
11 CHKCOM EQU $DEBE
12 FRMNUM EQU $DD67
```

² [CT] Actually, typing a string will give a ?REENTER warning message.

```

13 GETADR EQU $E752
14 LINNUM EQU $50
15 DATA EQU $380
16 *
0300: 20 BE DE 17 ENTRY JSR CHKCOM ; CHK SYNTAX
0303: 20 67 DD 18 JSR FRMNUM ; EVALUATE NUM
0306: 20 52 E7 19 JSR GETADR ; FAC -> INT
0309: A5 50 20 LDA LINNUM
030B: 8D 80 03 21 STA DATA
030E: A5 51 22 LDA LINNUM+1
0310: 8D 81 03 23 STA DATA+1
0313: 60 24 DONE RTS
0314: 2F 25 CHK

```

This is basically a repeat of the previous chapter's Pointer Setup routine, with the results being put into DATA, DATA+1. The advantage of this routine compared to the Integer Variable Reader is that not only is it shorter, but also that it will accept either integer or real variables (simple or array) and still do the string error check. This, then, is usually the preferred method.

String Variables

The goal here is to read some string data from the calling Applesoft program and then put it somewhere in memory where it presumably will be available to other portions of the assembly-language program. To illustrate this, enter the following two programs:

```

10 A$ = "TEST"
20 CALL 768, A$
30 END

```

```

1 *****
2 * AL17-STRING VARIABLE *
3 * READER *
4 * 2/1/82 *
5 *****
6 *
7 *
8 * OBJ $300
9 * ORG $300
10 *
11 CHKCOM EQU $DEBE
12 FRMEVL EQU $DD7B
13 CHKSTR EQU $DD6C
14 FACMO EQU $A0
15 FACLO EQU $A1 ; FAC+5
16 VARPNT EQU $83
17 DATA EQU $380
18 *
0300: 20 BE DE 19 ENTRY JSR CHKCOM ; CHK SYNTAX
0303: 20 7B DD 20 JSR FRMEVL ; EVALUATE
21 * (FACMO,LO) -> DESCRIPTOR
0306: 20 6C DD 22 JSR CHKSTR ; VAR = $?
23 *

```

```

0309: A0 00    24          LDY  #$00
030B: B1 A0    25          LDA  (FACMO),Y ; LEN OF $
030D: AA      26          TAX  ; SAVE LEN
030E: C8      27          INY  ; Y = 1
030F: B1 A0    28          LDA  (FACMO),Y ; ADDR LO BYTE
0311: 85 83    29          STA  VARPNT
0313: C8      30          INY  ; Y = 2
0314: B1 A0    31          LDA  (FACMO),Y ; ADDR HI BYTE
0316: 85 84    32          STA  VARPNT+1
0318: 8A      33          TXA  ; RETRIEVE LEN
0319: A8      34          TAY
                35          *
031A: 88      36          LOOP DEY
031B: B1 83    37          LDA  (VARPNT),Y ; GET CHR
031D: 99 80 03 38          STA  DATA,Y
0320: C0 00    39          CPY  #$00
0322: D0 F6    40          BNE  LOOP
                41          *
0324: 60      42          DONE RTS
0325: 4F      43          CHK

```

After running the calling program, enter the Monitor and list out the DATA region of memory with:

```
*380.383 <RETURN>
```

This should print out the following data:

```
0380- 54 45 53 54
```

This shows that the hex values for the characters “TEST” have been successfully transferred. Let’s see how it was accomplished.

The routine starts off rather like the previous ones by using CHKCOM to make sure a comma was used after the CALL and to prepare TXTPTR for reading in the data. FRMEVL (\$DD78 = FoRMula EVaLuation) is a very nice general-purpose routine that takes in virtually any numeric or string expression or literal, and places the final result in FAC. It is related to FRMNUM but is much more omnivorous. Upon returning from FRMEVL, FACMO and FACLO (\$A0, \$A1 = “...sorry, couldn’t find out where they got the names...”³) hold the address of the string’s *descriptor*, that is, the three-byte group giving the length and address of the actual string data.

Our routine uses FACMO, FACLO in the indirect addressing mode to retrieve the first byte of the descriptor, which is the length of the string. This is put into the X-Register for temporary storage. Some people prefer to push it onto the stack with a PHA command; it’s a matter of choice. Next, the address of the string data is retrieved from the descriptor and put into VARPNT, which is assumed to be

³ [CT] FACMO and FACLO are the Middle-Order and Low-Order bytes of the four-byte mantissa for the floating-point Accumulator.

not in use at the time. Last of all, we copy the data from its location, indicated by the VARPNT pointer, to our DATA address.

In experimenting, notice that the area from \$380 to \$3CF is open, but that the area starting at \$3D0 is reserved for DOS. Entering very long strings in the example may lead to some problems. In your own programs, it would be necessary to set aside a one-page area (\$100 = 256 bytes) to put the data, unless of course you can limit the length of the string before doing the call.

You may also wish to try variations in the Applesoft program by deleting line 10 and rewriting line 20 as:

```
20 CALL 768, "ABC" + "DEF"
```

or

```
20 CALL 768, LEFT$("ABCDE F")
```

or

```
10 A$(5,5) = "TEST"
20 CALL 768, A$(5,5)
```

Passing Data from Assembly Language

The converse of the techniques we've discussed so far actually is fairly simple. The key to much of it is the PTRGET routine used earlier. Because this routine will even create a variable when it's not already present, we can simply more or less reverse the process of the previous routines to pass data back to a calling Applesoft program.

Again, I'll illustrate an example for each variable type.

Integer Variables

The Applesoft program:⁴

```
10 POKE 896,2: POKE 897,1
20 CALL 768, A%
30 PRINT A%
40 END
```

⁴ [CT] For an interactive program, replace line 10 with:

```
5 PRINT CHR$(4); "BLOAD AL17.SENDINT"
6 INPUT "ENTER INTEGER LO,HI BYTES: "; A%,B%
10 POKE 896,A%: POKE 897,B%
```

Try entering "2,1", "255,127", or "1,128". Now try "0,128". Is this a legal integer value?

The assembly subroutine to be called is:

```

1 *****
2 *   AL 17-INTEGER VARIABLE   *
3 *           SENDER           *
4 *           2/1/82           *
5 *****
6 *
7 *
8 *           OBJ   $300
9 *           ORG   $300
10 *
11 CHKCOM EQU $DEBE
12 PTRGET EQU $DFE3
13 VARPNT EQU $83
14 MOVFM  EQU $EAF9
15 CHKNUM EQU $DD6A
16 DATA  EQU $380
17 *
0300: 20 BE DE 18 ENTRY   JSR  CHKCOM   ; CHK SYNTAX
0303: 20 E3 DF 19         JSR  PTRGET   ; FIND VARIABLE
20 * Y,A = ADDRESS OF VALUE
0306: 20 F9 EA 21         JSR  MOVFM   ; MOV VAL -> FAC
0309: 20 6A DD 22         JSR  CHKNUM   ; FAC = NUM?
030C: A0 00 23         LDY  #$00
030E: AD 81 03 24         LDA  DATA+1
0311: 91 83 25         STA  (VARPNT),Y
0313: C8 26         INY
0314: AD 80 03 27         LDA  DATA
0317: 91 83 28         STA  (VARPNT),Y
29 *
30 * NOTE! HIGH BYTE FIRST!
31 *
0319: 60 32 DONE   RTS
031A: F1 33         CHK

```

This program is a rather trivial exercise in that all that needs to be done is to reverse the operands of lines 24, 25 and 27, 28 from the first Integer Variable Reader program. Again, the only caution is to make sure that the bytes are transferred in the proper order, since integer data is reversed.

Real Variables

Real variables require the introduction of a few new routines. The same Applesoft calling program is used with only a minor modification.

```

10 POKE 896,2: POKE 897,1
20 CALL 768,A
30 PRINT A
40 END

```

The subroutine is entered as:

```

1 *****
2 *      AL 17-REAL VARIABLE *
3 *          SENDER          *
4 *          2/1/82          *
5 *****
6 *
7 *
8 *          OBJ $300
9 *          ORG $300
10 *
11 CHKCOM EQU $DEBE
12 PTRGET EQU $DFE3
13 CHKNUM EQU $DD6A
14 GIVAYF EQU $E2F2
15 MOVMF  EQU $EB2B
16 DATA  EQU $380
17 *
0300: 20 BE DE 18 ENTRY   JSR  CHKCOM      ; CHK SYNTAX
0303: AD 80 03 19          LDY  DATA
0306: AC 81 03 20          LDA  DATA+1
0309: 20 F2 E2 21          JSR  GIVAYF      ; DATA -> FAC
030C: 20 E3 DF 22          JSR  PTRGET      ; FIND VARIABLE
030F: 20 6A DD 23          JSR  CHKNUM      ; VAR = NUM?
24 * Y,A = ADDRESS OF VARIABLE DATA
0312: AA 25          TAX
0313: 20 2B EB 26          JSR  MOVMF      ; FAC -> MEMORY
0316: 60 27 DONE   RTS
0317: D1 28          CHK

```

The technique here is to use the routine GIVAYF (\$E2F2 = GIVE Accumulator and Y-Register to FAC) to put the two bytes of our integer number into the FAC. GIVAYF requires that the Accumulator and Y-Register be loaded with the high- and low-order bytes, respectively, for the integer number to be transferred⁵. As a bonus, the number may even be signed—that is, positive or negative. Signed binary numbers were covered in great detail in chapter 10.

Lines 19 and 20 load the appropriate registers, then, after calling GIVAYF, PTRGET and CHKNUM are used to determine the name of the variable to use in returning the data. Recall that after returning from PTRGET, the Y-Register and Accumulator will hold the low- and high-order bytes of the address of the data for the new variable digested by PTRGET.

MOVMF (\$EB2B = MOVE to Memory from FAC) is the routine we'll use to complete the process. It requires that the Y-Register and X-Register be loaded with the address of the memory location to which the contents of the FAC will be moved. Since PTRGET has just determined that for us, the only hitch is that PTRGET left the high-order byte in the Accumulator instead of in the X-Register as

⁵ [CT] The original article switched the meaning of the GIVAYF high and low bytes. The code above has been corrected and produces a value of 258.

we require. A simple TAX solves that problem, and the routine is concluded with the call to MOVMF and an RTS.

Programming Tip

Whenever a routine ends with a JSR to another routine, immediately followed by the ending RTS of the main routine, the line can be shortened one byte by changing the last JSR to a JMP. When the RTS in the last called subroutine is encountered, the RTS will cause an exit from the main routine instead. An example of this would be to rewrite the end of the program just listed as:

```

*
*
*
030F: 20 6A DD 23          JSR  CHKNUM      ; VAR = NUM?
                24 * Y,A = ADDRESS OF VARIABLE DATA
0312: AA      25          TAX
0313: 4C 2B EB 26  DONE   JMP  MOVMF      ; FAC -> MEMORY
                                AND RETURN!

```

String Variables

String variables are not much different but will require a slightly clumsy calling Applesoft program to demonstrate. Line 10 is a series of POKEs that will put the ASCII data for the string "TEST" into memory at our usual DATA (\$380) location. Additionally, a delimiter will be placed at the end of the string so that the routines we will be calling can determine the string's length. Use of a delimiter is more practical, especially in situations where you don't know the length of an incoming string until the carriage return or other delimiter shows up. The Applesoft routine we'll use will automatically determine the length by scanning the string for the delimiter.

```

10  POKE 896,84: POKE 897,69: POKE 898,83: POKE 899,84: POKE 900,0
20  REM "TEST" + NULL DELIMITER
30  CALL 768, A$
40  PRINT A$
50  END

```

The subroutine for this is:

```

1  *****
2  * AL17-STR$ VARIABLE SENDER *
3  *          2/1/82          *
4  *          R. WAGNER      *
5  *****
6  *
7  *
8  *          OBJ $300
9  *          ORG $300
10 *
11 CHKCOM EQU $DEBE

```

```

12 PTRGET EQU $DFE3
13 CHKSTR EQU $DD6C
14 FORPNT EQU $85
15 MAKSTR EQU $E3E9
16 SAVD EQU $DA9A
17 DATA EQU $380
18 *
19 *
0300: 20 BE DE 20 ENTRY JSR CHKCOM ; CHK SYNTAX
0303: 20 E3 DF 21 JSR PTRGET ; FIND VAR
0306: 20 6C DD 22 JSR CHKSTR ; VAR = $?
0309: 85 85 23 STA FORPNT
030B: 84 86 24 STY FORPNT+1 ; ADDR OF DESCR
030D: A9 80 25 LDA #$80
030F: A0 03 26 LDY #$03 ; A, Y = $380
0311: A2 00 27 LDX #$00 ; DELIMITER='00'
0313: 20 E9 E3 28 JSR MAKSTR ; DATA -> MEMORY
0316: 20 9A DA 29 JSR SAVD ; VARPNT = NEW $
0319: 60 30 DONE RTS
031A: CD 31 CHK

```

The new routines here are MAKSTR (\$E3E9 = MAKE STRing) and SAVD (\$DA9A = SAVE Descriptor). MAKSTR requires that the Accumulator and the Y-Register hold the address (low, high) of the string to be scanned and that the X-Register hold the value for the delimiting character. This example uses \$00, but another common variation would be to use a carriage return (\$8D) or a comma (\$2C). (Note that <RETURN> is almost always found in the input buffer with the high bit set, that is, \$8D versus \$0D).

After scanning for the delimiter, MAKSTR moves the data up to the string storage area at the top of memory.

SAVD is a companion routine which will take whatever string descriptor is currently pointed to by FORPNT (\$85, \$86 = FORMula PoiNTER) and match it to the data just moved by MAKSTR.

Looking at the listing, we can see that the only creative work to be done is moving the contents of the Accumulator and Y-Register to FORPNT. The Accumulator, Y-Register, and X-Registers are then prepared as was just described, and the remaining calls are done. Voila! Instant strings!

Conclusion

You'll notice that all of the routines handle arrays as well as simple variables. Additionally, certain more subtle points become apparent as you study the listings. For example, each of the last three Applesoft listings was done without defining the returned variable prior to the CALL. This was to demonstrate that PTRGET does a very nice job of creating the variable for us. In addition, in each case the data that was put into a variable and then later retrieved at DATA (and vice versa) should be consistent, thus demonstrating the accuracy of the methods.

You may also wish to experiment with using formulas or string calculations after the CALL statement to confirm that all legal Applesoft operations are acceptable.

Last but not least, I would like to give credit and thanks to Craig Peterson for his help in providing some of the information used in preparing this chapter.

Next chapter we'll look at some other applications of internal Applesoft routines within custom assembly-language programs.

Applesoft Hi-Res Graphics

March 1982

In the previous chapter we examined the techniques for passing data back and forth between Applesoft and assembly language in the form of standard Applesoft variables. This was greatly facilitated by the use of existing internal Applesoft routines. A natural extension of this idea is to use other internal Applesoft routines as may be appropriate to our given task. One of the most interesting applications of this is in the area of hi-res graphics.

There are two main reasons for doing hi-res graphics from assembly language. The first and most obvious is speed. By doing many of the operations directly in assembly language, the basic overhead (so to speak) of Applesoft is avoided, thus producing a noticeable speed increase in the overall program. Be aware however, that since we are ultimately still calling Applesoft routines, the speed increase has a certain limit. Greater speeds are obtained only by creating specialized and dedicated routines that perform only a specific function. The normal Applesoft routines are designed to be flexible and to occupy a minimum of space. Faster routines will do less and possibly be larger in terms of memory use. The trade-off must be weighed.

The second reason is simply the convenience of being able to do the same things, including graphics, from assembly language that you are able to do from BASIC. To this end, the techniques presented in this chapter should be quite adequate. In future chapters, we'll explore the creation of specialized routines that give higher speed and greater independence from the Applesoft routines.

Ground School

Before jumping into the intricate details of the various routines, we'll impose upon your patience long enough to describe briefly the model of Apple hi-res graphics used for the current discussion. This may seem unnecessary, but it will provide the common ground for the points to be made in this particular presentation. As you'll see in later chapters, there are a number of ways of looking at the total screen environment.

For the time being, we'll use an approximation of what is usually viewed as the "classical" explanation of the hi-res screen in Figure 18-1.

The array of possible points to be plotted consists of a field of 192 lines, each of which is made up of 280 points. If a mixed mode of graphics plus text is

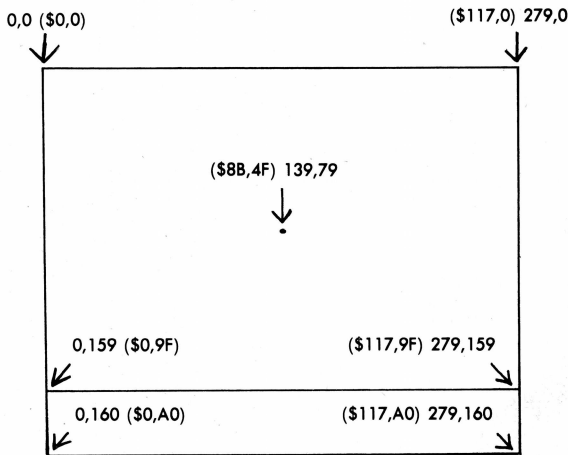


Figure 18-1: Hi-res screen coordinates

selected, only 160 graphics lines are displayed. On the majority of Apples, six colors are available: black, white, green, violet, orange, and blue.

These colors have been assigned to eight numeric values, as follows:

Set 1	Set 2
0 = Black1	4 = Black2
1 = Green	5 = Orange
2 = Violet	6 = Blue
3 = White1	7 = White2

White is created by plotting two color points right next to each other (green/violet or orange/blue). Black, when specifically plotted, is produced by turning off two adjacent color dots.

The model gets shaky when we have to tell you that things like “odd colors” (green or orange) can be plotted only at odd x-coordinates (1, 3, 5...), and that “even colors” (blue or violet) can be plotted only at even x-coordinates (0, 2, 4...). It gets even worse, but we’ll save the horror stories for chapter 20. For the time being, you’ll have many fewer headaches if you limit yourself to using the colors from only Set 1 or Set 2. Even better, stick to black and white for now, and fewer mysterious things will happen.

Landmarks and Entry Points

A number of the fundamental hi-res routine entry points are documented in various publications relating to the Apple. A brief summary is given in the following table.

Routine	Address	Description
HGR	\$F3E2	Initializes to hi-res page 1, clears screen.
HGR2	\$F3D8	Initializes to hi-res page 2, clears screen.
HCLR	\$F3F2	Clears current screen to black1.
BKGND	\$F3F6	Clears current screen to last plotted HCOLOR.
HCOLOR	\$F6F0	Sets HCOLOR to contents of X-Register (0–7).
HPOSN	\$F411	Positions hi-res “cursor” without plotting. Enter with X, Y (low, high) = horizontal position, Accumulator = vertical position.
HPLOT	\$F457	Identical to HPOSN, but plots current HCOLOR at coordinates given.
HFIND	\$F5CB	Returns current “cursor” position. Useful after a DRAW to find where you’ve been left. Coordinates returned in: \$E0, \$E1 = horizontal (low,high), \$E2 = vertical.
HLIN	\$F53A	Draws a line from last plot to point given. Accumulator, X (low, high) = horizontal, Y = vertical position.
SHNUM	\$F730	Puts address of shape number indicated by X-Register into \$1A, \$1B; returns with X, Y (low, high) also set to address of that shape-table entry.
DRAW	\$F601	Draw shape pointed to by X, Y (low, high) in current HCOLOR. Note: X, Y point to specific entry, <i>not</i> the beginning of the table. Call SHNUM first.
XDRAW	\$F65D	Erases shape just drawn (if there) by doing an exclusive OR with the screen data. Load X, Y (low, high) with address of shape to XDRAW or call SHNUM first with X-Register = shape number.

A Test Flight: Hi-Res Demo

To illustrate how these are actually put to use, assemble and run the following program:

```

1 *****
2 *      AL18-HIRES DEMO 1      *
3 *****
4 *
5 *
6 *      OBJ  $6000
7 *      ORG  $6000
8 *
9 PREAD  EQU  $FB1E
10 WAIT  EQU  $FCA8
11 PB0   EQU  $C061
12 HCOLOR EQU  $F6F0
13 HGR   EQU  $F3E2

```

```

14 HPLOT EQU $F457
15 HPOSN EQU $F411
16 HLIN EQU $F53A
17 ROT EQU $F9
18 SCALE EQU $E7
19 SHNUM EQU $F730
20 DRAW EQU $F601
21 PTR EQU $E8
22 *
6000: 4C 11 60 23 ENTRY JMP E2
6003: 01 00 04 24 TABLE HEX 010004
6006: 00 12 3F 25 HEX 00123F
6009: 20 64 2D 26 HEX 20642D
600C: 15 36 1E 27 HEX 15361E
600F: 07 00 28 HEX 0700
29 *
6011: 20 E2 F3 30 E2 JSR HGR ; CLR SCRN
6014: A2 03 31 LDX #$03 ; WHITE = 3
6016: 20 F0 F6 32 JSR HCOLOR
33 *
6019: A9 00 34 BORDER LDA #$00 ; Y = 0
601B: A8 35 TAY
601C: AA 36 TAX ; X = 0
601D: 20 57 F4 37 JSR HPLOT ; PLOT 0,0
6020: A9 17 38 LDA #$17 ;
6022: A2 01 39 LDX #$01 ; X = $117
6024: 20 3A F5 40 JSR HLIN ; HLIN TO 279,0
41 *
42 *
6027: A9 17 43 LDA #$17
6029: A2 01 44 LDX #$01 ; X = 279
602B: A0 9F 45 LDY #$9F ; Y = 159
602D: 20 3A F5 46 JSR HLIN ; HLIN TO 279,159
47 *
6030: A9 00 48 LDA #$00
6032: A2 00 49 LDX #$00 ; X = 0
6034: A0 9F 50 LDY #$9F ; Y = 159
6036: 20 3A F5 51 JSR HLIN ; HLIN TO 0,159
52 *
6039: A9 00 53 LDA #$00
603B: A2 00 54 LDX #$00 ; X = 0
603D: A0 00 55 LDY #$00 ; Y = 0
603F: 20 3A F5 56 JSR HLIN ; HLIN TO 0,0
57 *
6042: A9 03 58 SET LDA #$03
6044: 85 E8 59 STA PTR
6046: A9 60 60 LDA #$60
6048: 85 E9 61 STA PTR+1 ; SET TABLE TO $6003
62 *
604A: A2 00 63 READ LDX #$00 ; PDL(0)
604C: 20 1E FB 64 JSR PREAD
604F: 98 65 TYA
6050: D0 02 66 BNE R1
6052: A9 01 67 LDA #$01 ; FIX 0 -> 1
6054: 85 E7 68 R1 STA SCALE
6056: A9 18 69 LDA #$18

```

```

6058: 20 A8 FC 70          JSR  WAIT
605B: A2 01 71          LDX  #$01          ; PDL(1)
605D: 20 1E FB 72          JSR  PREAD
6060: 84 F9 73          STY  ROT
6062: A9 18 74          LDA  #$18
6064: 20 A8 FC 75          JSR  WAIT
        76 *
6067: A2 8B 77  DSPLY  LDX  #$8B
6069: A0 00 78          LDY  #$00          ; X = 139
606B: A9 4F 79          LDA  #$4F          ; Y = 79
606D: 20 11 F4 80          JSR  HPOSN
6070: A2 01 81          LDX  #$01          ; SHAPE #1
6072: 20 30 F7 82          JSR  SHNUM          ; FIND SHP ADDR
6075: A5 F9 83          LDA  ROT
6077: 20 05 F6 84          JSR  DRAW+4        ; USE SHNUM ENTRY PT
        85 *
607A: AD 61 C0 86  CHK   LDA  PB0
607D: 30 92 87          BMI  E2          ; BUTTON PUSHED
607F: 10 C9 88          BPL  READ        ; NO PUSH
        89 *
6081: F0 90          CHK

```

When run, this routine will draw a border around the hi-res screen, and then draw in the center of the screen the shape defined by the table. Scale and rotation values may be changed by adjusting paddles 0 and 1, respectively. Pushing button 0 will re-clear the hi-res screen of the accumulated images.

The routine starts with a jump over a data table to E2. The table is a simple shape table taken from page 95 of the *Applesoft II BASIC Programming Reference Manual*. It is a table to draw something resembling a square. The table could have been put at the beginning of the routine, but it would not then have been able to be BRUN.

Line 30 clears and displays the hi-res display page (page 1); lines 31, 32 use HCOLOR to set the color to be used to white1.

A border is then drawn in lines 34–50. HPLLOT (line 37) is used to plot the starting point (a requirement for subsequent use of HLIN, unless HPOSN is used for a “no-plot”).

Lines are drawn between the four corner points of the mixed-mode display. See Figure 18-1 to confirm the coordinates.

Once the border is done, preparation is made to use the shape table. Locations \$E8, \$E9 are used by Applesoft to point to the beginning of a shape table. SET initializes this pointer to our example table at \$6003. The table need not be part of the actual code, however, and could have been located virtually anywhere in memory. (Obvious exceptions would be the hi-res page area, \$2000–\$3FFF, and other reserved system areas.)

READ loads the X-Register with 0 to tell PREAD that we want to read paddle 0 and then puts the results (found in the Y-Register) into the SCALE parameter location (\$E7). Line 66 tests for a SCALE value of 0. Because Applesoft treats 0 as

the largest scale, this is shifted back to 1 to make the paddles more usable from a human standpoint.

Lines 69, 70 use the WAIT routine to wait a rather arbitrary amount of time before reading paddle 1. The value # $\$18$ was used as the delay value for very unscientific reasons. The larger the value, the more accurate the subsequent paddle readings, but the paddles will seem less responsive. Shorter delays give fast paddle response, but less accuracy. This effect is due to the fact that the Monitor reads the paddles by measuring the time it takes to charge a capacitor within the system. The higher the paddle setting, the longer it takes. The same capacitor is used for all paddles. When two or more paddles are read in rapid succession, the capacitor does not have time to return to its 0 value before the next read starts, and hence a false value is returned. The delay allows the system to make a better return to the desired states.

The interaction between the two paddles is most apparent when paddle 1 is set to 255 (full right). When paddle 0 is then increased from 0, the square is seen to rotate, as the scale parameter is increased. This does not happen when paddle 1 is at a low setting. One technique for minimizing paddle interaction is to read the same paddle twice when getting a reading (as we saw in chapter 12). If line 64 and 72 were duplicated in the listing, the result would be more stable. Try altering the listing and reassembling with the new technique. You'll find the distortion of paddle 0 much less pronounced than before.

The DSPLY section sets the coordinates to draw the shape at $\$8B$, $\$4F$ (139, 79). It then calls HPOSN to position the imaginary hi-res cursor at that point without actually plotting a point. SHNUM is then called, which finds the address of the first shape-table entry. SHNUM returns with the X- and Y-Registers holding the low- and high-order bytes for the entry. The Accumulator is then loaded with the ROTation value, and DRAW called.¹

Before repeating the cycle, pushbutton 0 is checked for a button press, which indicates the user wants to clear the hi-res screen.

A Minor Diversion

High-resolution graphics are generally used for two main purposes. The first is the presentation of graphical data, such as sales charts and equations. The routines presented here are adequate for that, but overall the task is probably better done directly in Applesoft anyway. Applesoft is often given a worse reputation than it deserves. It is quite versatile and, when combined with assembly-language subroutines, can perform quite admirably.

¹ [CT] Line 84 was corrected in the July 1982 *Softalk*: The DRAW routine ($\$F601$) is normally called with the X- and Y-Registers set to the address of the individual shape to be drawn. This can be automated, however, by first calling SHNUM ($\$F730$). When SHNUM is called, however, a later entry point to DRAW is used. Specifically, this should be DRAW+4 ($\$F605$). Entering at $\$F601$ by mistake can produce rather unpredictable results.

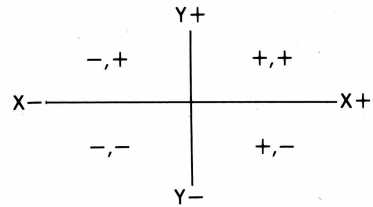
The other main area of concern is the production of screen animation, as is commonly seen in arcade-type games. This area brings up some new requirements in our expertise, because depictions of motion on the screen are really a matter of creating a computer simulation of motion, using the laws of physics to mimic the real world. (Next time somebody bugs you about writing or playing games, just tell them you're busy doing computer simulations.)

It would be impossible to present many more ideas in the area of graphics without relying on an underlying understanding of some of the principles used in creating a simulation program. Although we'll certainly not try to present a comprehensive tutorial on basic physics and computer graphics, we can get quite a bit of mileage out of one or two rather simple concepts.

Location

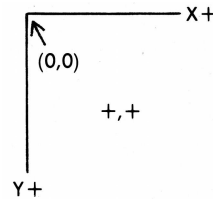
It should be fairly obvious that when specifying the coordinates of a point on the screen, we are giving information about the relative location of something. About the only thing different about the Apple screen is that the number system used is laid out somewhat differently from the Cartesian system described in junior high school math classes.

In the usual system, the point with the coordinates 0, 0 (the origin) is at the center of the display, and all possible combinations of positive and negative numbers are shown in the four quadrants.



This is more than we need to do Apple graphics though, because the screen uses only positive values, with the origin (0, 0) in the upper-left corner.

The location of objects always can be given by the number pair associated with the X and Y (horizontal and vertical) axes.

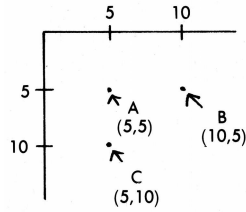


Motion

So much for discussions of elementary graphing. If you understood the first example of drawing the border on the screen, all this is already known to you. The reason we mention it is to prepare you for the next idea, the one of motion.

When something is moving, we say it has a velocity. Velocity has only two components: direction and magnitude. That is to say that the only things we have to worry about when simulating a moving object are its speed and its direction of travel. Speed is measured in units of distance per unit of time.

In the case of our screen display as shown to the right, something moving from point A to point B in one second would have a speed of +5 units per second. Likewise for something moving from point A to C. Negative values are used to indicate something moving in a direction opposite the given coordinate system. An object moving from point B to A in one second would have a speed of -5 units per second.



Now at this point you may find yourself tempted to throw up your hands and say, "I can tell where he's going and it doesn't sound fun!" You might think you're going to plunge deeper into the esoteric and rather uninteresting ramblings of a physics teacher and end up who-knows-where and for what good reason anyway?

Well, first of all, you're only going to have to wade in a very little bit deeper (the scary part comes when we try to do negative numbers in binary!). And second of all, the point of all this will be the simple goal of bouncing a little ball around on the screen. As it happens, we must know a bit about how the universe works if we are going to simulate it on our TV screen. And if you really intend to end up with spaceships careening wildly about, you'll have to show a little determination now to get the basics under your belt. So much for the halftime pep talk.

The sticky question is how to handle objects that are moving from, say, point C to A. As a case of extremely good fortune, it turns out we can consider the *components* of the motion quite easily and achieve our end result, without having to know the object's real diagonal speed.

What this means is that we can give an object both a horizontal and vertical component to its motion, and then do the appropriate calculations separately.

Speed can be rephrased as "a change in position with respect to time." On the screen, what this means is that something will appear to move consistent with the real world as long as we keep re-plotting its position in a regular manner. The timebase of the operations ends up depending on how fast we cycle through the re-plotting pattern. Since an example can work wonders, let's take a moment to examine a program in (oh no!) Applesoft:

```

10 HGR
20 X = 0: Y = 80
30 V = 1
100 REM DRAW LOOP
110 HCOLOR = 3: REM WHITE
120 HPLOT X,Y : REM DRAW OBJECT
130 HCOLOR = 0: REM BLACK
140 HPLOT X,Y : REM ERASE IT
200 REM MAKE IT MOVE!
210 X = X + V
220 IF X > 278 THEN V = V*(-1)

```

```
230 IF X < 1 THEN V = V*(-1)
240 GOTO 100
```

This program will bounce a tiny spot off the left and right sides of the screen. The important things to note are that (1) motion is simulated by adding a constant velocity factor V to the position of each cycle; (2) the object is erased from its old position before being redrawn at the new one; and (3) a bounce is basically a complete reversal of the velocity factor, that is, the value is multiplied by minus one. The speed with which everything is executed depends on the inherent speed of the programming language and how fast we can cycle through the service loop. If for some reason the loop shown was too fast, you could put a FOR-NEXT delay loop in anywhere along the line. If it was too slow, you could increase the speed factor, V , from 1 to a larger number. Larger numbers produce more jerky motion, however. The other option would be to write it in assembly language!

Before doing that, though, let's make it two-dimensional by giving the ball both horizontal and vertical components to its motion:

```
10 HGR
20 X = 140 : Y = 80
30 XV = 1 : YV = 1
100 REM DRAW LOOP
110 HCOLOR = 3: REM WHITE
120 HPLOT X,Y : REM DRAW OBJECT
130 HCOLOR = 0: REM BLACK
140 HPLOT X,Y : REM ERASE IT
200 REM MAKE IT MOVE!
210 X = X + XV : Y = Y + YV
220 IF X > 278 THEN XV = XV*(-1)
230 IF X < 1 THEN XV = XV*(-1)
240 IF Y > 158 THEN YV = YV*(-1)
250 IF Y < 1 THEN YV = YV*(-1)
260 GOTO 100
```

In this program we see both components of motion, vertical and horizontal. Again, a bounce consists of taking the negative value of the component we are reversing. The flicker is caused by erasing the dot so soon after we draw it, and also by the scanning nature of the TV or monitor. It can be smoothed out by adding a line:

```
125 FOR I = 1 TO 5: NEXT I
```

This also will slow down the speed of the ball a bit, but it does help the overall screen appearance. You are advised to watch this fascinating program run for a while, meditating on the nature of the programming steps occurring throughout the travel, and particularly at each bounce. This concept is essential to any further animation efforts on your Apple.

Calling Hi-Res Graphics Routines

April 1982

In the previous chapter we discussed hi-res graphics and how to plot a bouncing hi-res ball. We constructed a simple Applesoft program to illustrate the principles involved:

```

10 HGR
20 X = 140 : Y = 80
30 XV = 1 : YV = 1
100 REM DRAW LOOP
110 HCOLOR = 3: REM WHITE
120 HPLOT X,Y : REM DRAW OBJECT
130 HCOLOR = 0: REM BLACK
140 HPLOT X,Y : REM ERASE IT
200 REM MAKE IT MOVE!
210 X = X + XV : Y = Y + YV
220 IF X > 278 THEN XV = XV*(-1)
230 IF X < 1 THEN XV = XV*(-1)
240 IF Y > 158 THEN YV = YV*(-1)
250 IF Y < 1 THEN YV = YV*(-1)
260 GOTO 100

```

Note that this loop has a basic pattern of: draw → erase → calculate → check → (do it again...).

For the Applesoft program shown, this works fairly well and is very understandable. There is one problem, however: very little time passes between the draw and erase stages, compared to the amount of time spent in the calculate and test sections. The result on the screen is a large amount of flicker, resulting from the dot spending more of its time black than white.

One solution to this is to make a small modification to the original Applesoft program, so that it appears as follows:

```

0 REM FP DOT DEMO PROGRAM
10 HGR
15 HCOLOR = 3 : HPLOT 0,0 TO 279,0 TO 279,159 TO 0,159 TO 0,0
20 X = 140 : Y = 80
30 XV = 1 : YV = 1
100 REM CALC NEW POSN
110 TX = X + XV : TY = Y + YV

200 REM CHECK POSN
210 IF TX > 277 THEN XV = XV*(-1) : GOTO 110
220 IF TX < 2 THEN XV = XV*(-1) : GOTO 110

```

```

230 IF TY > 157 THEN YV = YV*(-1) : GOTO 110
240 IF TY < 2 THEN YV = YV*(-1) : GOTO 110

300 REM ERASE OLD POSN
310 HCOLOR = 0: REM BLACK
320 HPLLOT X,Y

400 REM DRAW NEW POSN
410 X = TX : Y = TY
420 HCOLOR = 3: REM WHITE
430 HPLLOT X,Y
440 GOTO 100

```

This routine not only draws a nice border around the screen, but also follows this general pattern: calculate → check → erase → draw → (start over).

The advantage of this technique is that relatively little time is spent between the erase and redraw stages. Thus the dot is on the screen the majority of the time and very little flicker is apparent.

Another new detail is the use of a set of temporary variables, TX and TY. These store the new position while the old one is being erased. The new one is then drawn and TX, TY are made “official” by being passed to the “real” X, Y variables.

As a minor point, also note that we have reduced the boundary test points in lines 200–240 so that the dot reverses direction before actually contacting the boundary we have drawn. Otherwise, the boundary would be erased by the dot passing through it on each bounce.

Now let’s look at how to implement this program in assembly language.

Taking the Opposite of a Signed Number

In chapter 10 we discussed the sign bit and how to represent negative numbers.¹ Recall that negative numbers are defined using the *two’s complement* system: reverse each bit of the positive number, then add one.

All that we need now is a routine that will produce the opposite of a number given it—that is produce the two’s complement of a positive number and also the positive equivalent when given a negative value. To do this, we’ll use the EOR command.

EOR is useful in creating a routine to convert between signed numbers because of its ability to reverse all of the bits in a given byte. The conversion is done with two individual routines. In the examples below, the routines convert a constant value, #\$34, back and forth. In a working version of this program, the value would be passed via a register or a memory location, as will be shown later.

¹ [CT] The original *Softalk* article #19 (April 1982) contained a section on “signed binary numbers.” In *Assembly Lines: The Book* (and in this book), this material is presented in chapter 10.

```

                                Positive to Negative
ENTRY   LDA   #$34              ; %00110100 = +52
                                ; TO BE CONVERTED TO -52
        EOR   #$FF              ; %11111111 TO REVERSE BITS
                                ; RESULT = %11001011
        CLC
        ADC   #$01              ; RESULT = RESULT + 1
                                = %11001100 = $CC
DONE    STA   MEM               ; STORE RESULT
        RTS

                                Negative to Positive
ENTRY   LDA   #$CC              ; %11001100 = $CC = -52
                                ; TO BE CONVERTED BACK
        SEC
        SBC   #$01              ; ACCUM = ACCUM - 1
                                = %11001011 = $CB
        EOR   #$FF              ; REVERSE ALL BITS
                                ; RESULT = %00110100 = $34 = +52
DONE    STA   MEM               ; STORE RESULT
        RTS

```

Note that in this example the percent sign is used to indicate the binary form of the number. Some assemblers (such as *Merlin*) support this notation.

The Real Thing: Hi-Res in Assembly

We now have the tools necessary to construct the assembly-language version of the last Applesoft listing. Assemble and run this listing:

```

1 *****
2 * AL19-HIRES ONE DOT PROGRAM *
3 *****
4 *
5 *
6 *          OBJ  $6000
7          ORG  $6000
8 *
9 X          EQU  $E0          ; $E0,$E1
10 Y         EQU  $E2
11 XV        EQU  $06          ; $06,$07
12 YV        EQU  $08
13 TX        EQU  $09          ; $09,$0A
14 TY        EQU  $0B
15 *
16 PREAD     EQU  $FB1E
17 WAIT      EQU  $FCA8
18 HCOLOR    EQU  $F6F0
19 HGR       EQU  $F3E2
20 HPLLOT    EQU  $F457
21 HPOSN     EQU  $F411
22 HLIN      EQU  $F53A
23 *
6000: 20 E2 F3 24 ENTRY   JSR  HGR

```

```

6003: A2 03 25          LDX #03      ; WHITE
6005: 20 F0 F6 26      JSR HCOLOR
        27          *
6008: A9 00 28          BOX LDA #00      ; Y = 0
600A: A8 29          TAX
600B: AA 30          TAX
600C: 20 57 F4 31      JSR HPLLOT   ; PLOT 0,0
600F: A9 17 32          LDA #23      ; 279 MOD 256
6011: A2 01 33          LDX #01      ; 279/256
6013: 20 3A F5 34      JSR HLIN     ; FROM 0,0 TO 279,0
        35          *
6016: A9 17 36          LDA #23
6018: A2 01 37          LDX #01
601A: A0 9F 38          LDY #9F     ; Y = 159
601C: 20 3A F5 39      JSR HLIN     ; 279,0 TO 279,159
        40          *
601F: A9 00 41          LDA #00
6021: A2 00 42          LDX #00
6023: A0 9F 43          LDY #9F
6025: 20 3A F5 44      JSR HLIN     ; 279,159 TO 0,159
        45          *
6028: A9 00 46          LDA #00
602A: A2 00 47          LDX #00
602C: A0 00 48          LDY #00
602E: 20 3A F5 49      JSR HLIN     ; 0,159 TO 0,0
        50          *
6031: A9 00 51          SET LDA #00
6033: 85 07 52          STA XV+1
6035: A9 01 53          LDA #01
6037: 85 06 54          STA XV     ; XV = 1
6039: 85 08 55          STA YV     ; YV = 1
        56          *
603B: A2 8C 57          POSN LDX #8C
603D: A0 00 58          LDY #00     ; X = 140
603F: A9 50 59          LDA #50     ; Y = 80
6041: 20 11 F4 60      JSR HPOSN   ; SET CURSOR AT X,Y
        61          *
6044: 18 62          CALC CLC
6045: A5 E0 63          LDA X
6047: 65 06 64          ADC XV
6049: 85 09 65          STA TX
604B: A5 E1 66          LDA X+1
604D: 65 07 67          ADC XV+1
604F: 85 0A 68          STA TX+1   ; TX = X + XV
        69          *
6051: 18 70          CLC
6052: A5 E2 71          LDA Y
6054: 65 08 72          ADC YV
6056: 85 0B 73          STA TY     ; TY = Y + YV
        74          *
6058: A5 0A 75          CHK LDA TX+1
605A: D0 09 76          BNE CHK2
605C: A5 09 77          LDA TX
605E: C9 02 78          CMP #02
6060: B0 03 79          BCS CHK2
6062: 20 AE 60 80      JSR RVRSX   ; TX < 2

```

```

81 *
6065: A5 0A 82 CHK2 LDA TX+1
6067: C9 01 83 CMP #001
6069: 90 09 84 BCC CHK3
606B: A5 09 85 LDA TX
606D: C9 16 86 CMP #16
606F: 90 03 87 BCC CHK3
6071: 20 AE 60 88 JSR RVRSX ; TX >= $116 (278)
89 *
6074: A5 0B 90 CHK3 LDA TY
6076: C9 02 91 CMP #002
6078: B0 03 92 BCS CHK4
607A: 20 D6 60 93 JSR RVRSY ; TY < 2
94 *
607D: A5 0B 95 CHK4 LDA TY
607F: C9 9E 96 CMP #9E
6081: 90 03 97 BCC ERASE
6083: 20 D6 60 98 JSR RVRSY ; TY >= $9E (158)
99 *
6086: A2 00 100 ERASE LDX #000 ; BLACK = 0
6088: 20 F0 F6 101 JSR HCOLOR
608B: A6 E0 102 LDX X
608D: A4 E1 103 LDY X+1 ; GET X,X+1
608F: A5 E2 104 LDA Y ; GET Y
6091: 20 57 F4 105 JSR HPLOT ; ERASE POINT
106 *
6094: A2 03 107 PLOT LDX #003 ; WHITE1 = 3
6096: 20 F0 F6 108 JSR HCOLOR
6099: A6 09 109 LDX TX
609B: A4 0A 110 LDY TX+1 ; GET TX,TX+1
609D: A5 0B 111 LDA TY ; GET TY
609F: 20 57 F4 112 JSR HPLOT ; PLOT POINT
113 *
60A2: A2 00 114 DELAY LDX #000 ; PDL0
60A4: 20 1E FB 115 JSR PREAD
60A7: 98 116 TYA
60A8: 20 A8 FC 117 JSR WAIT
118 *
119 *
60AB: 4C 44 60 120 GOBACK JMP CALC
121 *
122 *
60AE: A5 07 123 RVRSX LDA XV+1
60B0: 30 12 124 BMI NEGPOSX
125 *
60B2: A5 06 126 POSNEGX LDA XV
60B4: 49 FF 127 EOR #FF
60B6: 18 128 CLC
60B7: 69 01 129 ADC #001
60B9: 85 06 130 STA XV
60BB: A5 07 131 LDA XV+1
60BD: 49 FF 132 EOR #FF
60BF: 69 00 133 ADC #000
60C1: 85 07 134 STA XV+1
60C3: 60 135 RTS ; XV -> -XV
136 *

```



```

60C4: A5 06    137 NEGPOX LDA XV
60C6: 38      138         SEC
60C7: E9 01    139         SBC #$01
60C9: 49 FF    140         EOR #$FF
60CB: 85 06    141         STA XV
60CD: A5 07    142         LDA XV+1
60CF: E9 00    143         SBC #$00
60D1: 49 FF    144         EOR #$FF
60D3: 85 07    145         STA XV+1
60D5: 60      146 DONEX  RTS           ; -XV -> XV
        147 *
        148 *
        149 *
60D6: A5 08    150 RVRSY  LDA YV
60D8: 30 0A    151         BMI NEGPOSY
        152 *
60DA: A5 08    153 POSNEGY LDA YV
60DC: 49 FF    154         EOR #$FF
60DE: 18      155         CLC
60DF: 69 01    156         ADC #$01
60E1: 85 08    157         STA YV
60E3: 60      158         RTS           ; YV -> -YV
        159 *
60E4: A5 08    160 NEGPOSY LDA YV
60E6: 38      161         SEC
60E7: E9 01    162         SBC #$01
60E9: 49 FF    163         EOR #$FF
60EB: 85 08    164         STA YV
60ED: 60      165 DONEY  RTS           ; -YV -> YV
        166 *
60EE: 3A      167         CHK

```

When you run this routine, notice how much faster it executes and how the speed of the dot can be varied using paddle 0.

This routine essentially parallels the Applesoft routine shown earlier. Lines 24–50 clear the hi-res screen and draw the border. Lines 51–55 set the velocity components to 1; lines 57–61 position the hi-res cursor in the center of the screen. This also conveniently loads \$E0–E2 with the desired X and Y coordinates of the dot. Remember that \$E0, \$E1, and \$E2 are the zero-page locations used by the Applesoft hi-res routines for the X and Y coordinates of its cursor.

Lines 62–73 calculate the new position of the dot by adding the respective velocity components to the X and Y coordinates. Lines 75–98 check to see whether this new position is still within the specified screen boundaries. If it has reached the edge, the appropriate velocity components are reversed for the next go-round's calculation.

Line 100 starts the erasing of the current dot position, immediately followed by a drawing of the new position. Note that the equivalent of the X=TX:Y=TY statement is apparently missing. In actuality, it is automatically accomplished by the JSR H PLOT on line 112. Remember that the contents of the Accumulator, X- and Y-Registers are automatically assigned to \$E0–E2 by H PLOT. Line 114 does a

short delay by getting a value from paddle 0 to be used by the WAIT (\$FCA8) routine. After the delay, a JMP CALC restarts the entire process.

Lines 126–165 are applications of the sign-reversal routines shown earlier. Notice that RVRSY is a one-byte reversal, while RVRSX illustrates the reversal of a two-byte value. Similarly, CALC shows that the same addition routine is used for both signed binary (our current condition) and unsigned binary (as in previous chapters).

Table-Driven Graphics

For graphics of any complexity—anything involving more than one dot—a little improvement on this routine is needed. One of the most common ways of doing this is to use a table of all the current points on the screen and their corresponding velocities. Motion is then managed by sequentially scanning through the table and using the entire calculation, check and erase/plot section as a sub-routine.

To convert the routine presented earlier, make the following changes to the source code (the hex data from the assembly is included to assist in error checking):²

1. Add these lines to the end of the listing (new line numbers shown):

```

237 *
238 *
239 *
240 *
6154: A2 00      241 SETUP   LDX  #$00
6156: BD 62 61   242 LOOP    LDA  DATA,X
6159: 9D 00 10   243        STA  TABLE,X
615C: E8         244        INX
615D: E0 28     245        CPX  #40      ; 8 BYTES * NUM DOTS
615F: 90 F5     246        BCC  LOOP
6161: 60         247 DONE   RTS
248 *
6162: 8C 00 50   249 DATA   HEX  8C005000 ; X,Y(1) = 8C,50
6166: 01 00 01   250        HEX  01000100 ; XV,YV(1) = 1,1
251 *
616A: 8E 00 52   252        HEX  8E005200 ; X,Y(2) = 8E,52
616E: 01 00 01   253        HEX  01000100 ; XV,YV(2) = 1,1
254 *
6172: 90 00 54   255        HEX  90005400 ; X,Y(3) = 90,54
6176: 01 00 01   256        HEX  01000100 ; XV,YV(3) = 1,1
257 *
617A: 92 00 56   258        HEX  92005600 ; X,Y(4) = 92,56
617E: 01 00 01   259        HEX  01000100 ; XV,YV(4) = 1,1
260 *
6182: 94 00 58   261        HEX  94005800 ; X,Y(5) = 94,58
6186: 01 00 01   262        HEX  01000100 ; XV,YV(5) = 1,1

```

² [CT] The checksum for the new program is \$06.

```

                263 *
618A: 06        264      CHK

```

2. Rewrite line 120 (will end up as 190) as:

```

6113: 60        190 GOBACK  RTS

```

3. Rewrite the beginning of the source as:

```

1 *****
2 *      AL19-HIRES LOTS DOTS      *
3 *****
4 *
5 *
6 *      OBJ  $6000
7 *      ORG  $6000
8 *
9 TABLE  EQU  $1000
10 CTR    EQU  $0C
11 NUM    EQU  $05      ; FIVE DOTS
12 *
13 X      EQU  $E0      ; $E0,$E1
14 Y      EQU  $E2
15 XV     EQU  $06      ; $06,$07
16 YV     EQU  $08
17 TX     EQU  $09      ; $09,$0A
18 TY     EQU  $0B
19 *
20 PREAD  EQU  $FB1E
21 WAIT   EQU  $FCA8
22 HCOLOR EQU  $F6F0
23 HGR    EQU  $F3E2
24 HPLOT  EQU  $F457
25 HPOSN  EQU  $F411
26 HLIN   EQU  $F53A
27 *
6000: 20 E2 F3 28 ENTRY  JSR  HGR
6003: A2 03    29          LDX  #$03      ; WHITE
6005: 20 F0 F6 30          JSR  HCOLOR
31 *
6008: 20 54 61 32 TABLESET JSR  SETUP
33 *
600B: A9 00    34 BOX    LDA  #$00      ; Y = 0
600D: A8      35          TAY
600E: AA      36          TAX
600F: 20 57 F4 37          JSR  HPLOT      ; PLOT 0,0

```

4. Insert the code for the table lookup starting at new line 68:

```

6047: A9 00    68 LOOKUP  LDA  #$00
6049: 85 0C    69          STA  CTR
604B: A5 0C    70 GET    LDA  CTR
604D: 0A      71          ASL
604E: 0A      72          ASL
604F: 0A      73          ASL      ; X = CTR*8

```

		74	*		
6050:	AA	75		TAX	
6051:	BD 00 10	76		LDA	TABLE,X
6054:	85 E0	77		STA	X
6056:	E8	78		INX	
6057:	BD 00 10	79		LDA	TABLE,X
605A:	85 E1	80		STA	X+1
605C:	E8	81		INX	
605D:	BD 00 10	82		LDA	TABLE,X
6060:	85 E2	83		STA	Y
6062:	E8	84		INX	
6063:	E8	85		INX	; Y + 1 NOT USED
		86	*		
6064:	BD 00 10	87		LDA	TABLE,X
6067:	85 06	88		STA	XV
6069:	E8	89		INX	
606A:	BD 00 10	90		LDA	TABLE,X
606D:	85 07	91		STA	XV+1
606F:	E8	92		INX	
6070:	BD 00 10	93		LDA	TABLE,X
6073:	85 08	94		STA	YV
		95	*		
6075:	20 AC 60	96	SERVICE	JSR	CALC
		97	*		
6078:	A5 0C	98	PUT	LDA	CTR
607A:	0A	99		ASL	
607B:	0A	100		ASL	
607C:	0A	101		ASL	
607D:	AA	102		TAX	
		103	*		
607E:	A5 E0	104		LDA	X
6080:	9D 00 10	105		STA	TABLE,X
6083:	E8	106		INX	
6084:	A5 E1	107		LDA	X+1
6086:	9D 00 10	108		STA	TABLE,X
6089:	E8	109		INX	
608A:	A5 E2	110		LDA	Y
608C:	9D 00 10	111		STA	TABLE,X
608F:	E8	112		INX	
6090:	E8	113		INX	; SKIP BYTE
		114	*		
6091:	A5 06	115		LDA	XV
6093:	9D 00 10	116		STA	TABLE,X
6096:	E8	117		INX	
6097:	A5 07	118		LDA	XV+1
6099:	9D 00 10	119		STA	TABLE,X
609C:	E8	120		INX	
609D:	A5 08	121		LDA	YV
609F:	9D 00 10	122		STA	TABLE,X
		123	*		
60A2:	E6 0C	124		INC	CTR
60A4:	A5 0C	125		LDA	CTR
60A6:	C9 05	126		CMP	#NUM ; NUMBER OF DOTS
60A8:	90 A1	127		BCC	GET
60AA:	B0 9B	128		BCS	LOOKUP
		129	*		

```

                130 *
                131 *
60AC: 18        132 CALC      CLC
60AD: A5 E0    133          LDA  X
60AF: 65 06    134          ADC  XV

```

Run this routine from the Monitor with a 6000G or from Applesoft with a CALL 24576. If calling from the Monitor, make sure you have entered the Monitor from Applesoft when you do the CALL -151 to ensure that the Applesoft ROM or RAM card bank is selected. Note that although the entire routine is in assembly language, it does require the presence of the Applesoft hi-res routines in the \$D0000-\$F7FF range. By using paddle 0 you can vary the speed of execution considerably. One drawback of using the WAIT routine is that 0 will be just as slow as 255 when adjusting the paddle. Otherwise, it should behave quite nicely. To speed things up further, NOP out the JSR to WAIT on line 187. An even greater speed increase is achieved by similarly disabling the JSR PREAD on line 185, although with PREAD gone there is no longer any control over the speed. However, this will give you an idea of the maximum speed possible for the five dots using standard Applesoft hi-res routines.

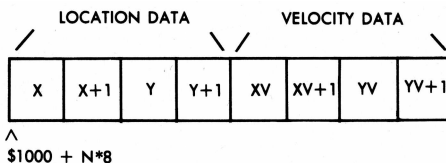
The main points to note in the new listing are the JSR to SETUP on line 32, the LOOKUP section in lines 68-128, and the table generator at the end in lines 241-263.

SETUP creates a data table starting at location \$1000 that contains a number of eight-byte blocks, each of which contains the information necessary for a given dot. The block is made up of two four-byte subunits. The first four bytes give the location data for the X and Y coordinates. Notice that the fourth byte is not used. Space in the table could have been saved by omitting this byte, but the eight byte length per entry allows us to use a few simple ASLs, as will be explained momentarily.

The second four bytes hold the velocity data, again in an X, Y format, with byte four being unused.

LOOKUP basically does three things. First it retrieves the data for a dot and puts it in the current X, Y, XV, YV bytes. Second, it feeds these to the CALC and PLOT routines. Third, when CALC/PLOT returns, the new location and velocity values are stored back in the table.

Examining the code starting at GET, you can see that CTR is used to keep track of which dot we're currently processing. This is multiplied by 8 to get the base address of the data for that dot. Remember that ASL can be used to multiply



easily by a power of two depending on the number of ASLs you use. Each ASL is equivalent to multiplying by 2.

Once the base address offset is determined, this is put in the X-Register and the data retrieved via a series of LDA/STA operations. After returning from CALC/PLOT, the process is reversed to store the new data.

Conclusion

Hi-res is an involved topic, and it's challenging to try to present the right mix of clarity and in-depth explanation. My goal is to provide enough of the basics to give you the springboard to pursue your own interests.

In general, the principles provided in this chapter and the one before it are the foundation of most animated graphics programs. Tables are especially worth your consideration as they provide a straightforward way of managing a larger number of screen points.

By now it also should be evident that even in assembly language, the Applesoft routines themselves are still the most restraining portion in terms of speed and execution. In all fairness to Applesoft, though, realize that their speed is sacrificed for simplicity and convenience of operation.

Next chapter's topic will be the layout of the hi-res screen itself, and how certain dedicated routines can be created to get a little more out of the ol' Apple.

Structure of the Hi-Res Display Screen

May 1982

In the preceding discussions of hi-res graphics we've relied on the existing Applesoft BASIC routines to do the necessary plotting of points from assembly language. From your previous experience with Applesoft and even from the most recent hi-res moving-dot programs presented, you may have noticed certain peculiarities about hi-res graphics. The problems lie in certain intrinsic shortcomings in the explanation of hi-res graphics offered so far.

To explore this area further let's examine, one by one, a number of problems that can occur—and thus discover the underlying structure of the hi-res display screen.

Loading a Hi-Res Screen: the “Fill” Effect

The fundamental question to be answered in this discussion is, “How are individual points plotted on the screen?” It should be relatively easy to accept the notion that to display a screen whose appearance can be arbitrarily changed, the RAM portion of the computer must be used. The area used is the range of memory from \$2000 to \$3FFF (8192 to 16383 decimal). This is called the page one hi-res display. The Apple II is also capable of displaying an alternate memory range called, cleverly enough, the page two hi-res display. This display is derived from the data contained in the memory range \$4000 to \$5FFF.

This chapter will focus primarily on page one, although for the most part page two can be considered to be just a simple offset from page one.

It also should be intuitively obvious that the display must in some way be linked to the actual contents of each byte in the ranges mentioned. This can easily be investigated by doing the following:

From Applesoft BASIC, select and clear the page one hi-res display by typing in HGR<RETURN>. If the cursor is not still visible, press <RETURN> until it reappears at the bottom of the screen.

Now enter the Monitor with a CALL -151. The first thing to do is to fill memory with a sample value. Do this by entering the following:

```
2000:FF
2001<2000.3FFFM
```


When you press <RETURN>, the screen should rapidly fill to white. Enter <CTRL>C to return to BASIC. Let's save the screen now by placing a convenient disk in the drive and entering:

```
BSAVE TESTPIC,A$2000,L$2000
```

Besides providing the information on how to save a hi-res image, the purpose of this instruction was to allow you to watch the screen fill at a little slower pace. You may have noticed when you filled the screen just now that it did not fill in an exactly continuous pattern, line-by-line from top to bottom. It did happen rather quickly, though.

Clear the screen by typing HGR<RETURN> again, and now load the data from disk back into memory by entering:

```
BLOAD TESTPIC
```

This time the screen should fill more slowly, and the somewhat strange pattern this generates will be more apparent. So now our problem is: "How is a vertical screen position (line) selected in terms of its memory address?" (Or: "Why does the screen load in such a funny way?")

Your first impulse might be to say "Well, if I were designing the computer, I'd just multiply the number of the line I wanted by the number of bytes per line to get the base address (the address of the first byte of the line) for the line. For example, if each line took forty bytes (which, by the way, it does), line 0 would have a base address of \$2000. Line 1 would be $\$2000 + 1 \times \28 ($\$28 = 40$ decimal) = \$2028. Line 2 would be $\$2000 + 2 \times \$28 = \$2060$, and so on.

An additional benefit would show up in the form of some unused bytes on the hi-res page. For 192 lines, the last address used would be $\$2000 + (192 \times \$28) - 1 = \$3DFF$. Since we've allotted the area from \$2000 to \$3FFF for page one, this would leave \$200 (512 decimal) bytes left over!

Unfortunately, that's not the way the Apple was set up. It turns out that multiplication routines are kind of a drag in terms of speed and memory usage, unless you're using exact multiples of two. A much more compact (and faster) algorithm is:

```

1 *****
2 *   AL20-HIRES BASE ADDRESS   *
3 *   CALCULATOR ROUTINE      *
4 *****
5 *           OBJ   $300
6 *           ORG   $300
7 GBAS      EQU   $26
8 HPAG      EQU   $E6           ; HGR=$20, HGR2=$40
9 *
10 * CALC BASE ADDRESS FOR Y-COORD IN ACCUM.
11 * GBAS = ADDR OF 1ST BYTE OF LINE SPECIFIED.
12 * ASSUME ACCUM HAS BITS abcdefgh, C=carry
0300: 48 13 ENTRY   PHA           ; abcdefgh
```

```

0301: 29 C0      14      AND  #$C0      ; ab000000
0303: 85 26     15      STA  GBAS     ;
0305: 4A       16      LSR          ; 0ab00000
0306: 4A       17      LSR          ; 00ab0000
0307: 05 26     18      ORA  GBAS     ; abab0000
0309: 85 26     19      STA  GBAS     ;
030B: 68       20      PLA          ; abcdefgh
030C: 85 27     21      STA  GBAS+1   ;
030E: 0A       22      ASL          ; bcdefgh0 C=a
030F: 0A       23      ASL          ; cdefgh00 C=b
0310: 0A       24      ASL          ; defgh000 C=c
0311: 26 27     25      ROL  GBAS+1   ; bcdefghc C=a
0313: 0A       26      ASL          ; efgh0000 C=d
0314: 26 27     27      ROL  GBAS+1   ; cdefghcd C=b
0316: 0A       28      ASL          ; fgh00000 C=e
0317: 66 26     29      ROR  GBAS     ; eabab000
                30      *
0319: A5 27     31      LDA  GBAS+1   ; cdefghcd
031B: 29 1F     32      AND  #$1F     ; 000fghcd
031D: 05 E6     33      ORA  HPAG     ; 001fghcd (PAGE 1)
031F: 85 27     34      STA  GBAS+1   ; 001fghcd
                35      *
0321: 60       36      DONE  RTS

```

Although it's perhaps not obvious how this works, the routine does take any value in the Accumulator, from 0 to 191, and return the appropriate base address of the corresponding line in locations \$26, \$27 (GBAS). This code is "stolen" from a similar routine in the Applesoft hi-res routine HPOSN (\$F411) mentioned in the previous chapter.¹

The overall pattern to the screen-filling operation is as follows. The first forty bytes of memory correspond to line 0 of the screen display. The next forty bytes form line 63, and the next forty bytes line 127. At the end of the line 127 is a block of eight unused bytes. ($3 \times 40 + 8 = 128$ bytes). This pattern is repeated sixty-three more times to create all 192 screen lines.² ($3 \times 64 = 192$ lines; 64×128 bytes = 8,192 bytes per hi-res page.)

When hi-res page 1 is loaded from disk, the range of memory is filled sequentially from \$2000 to \$3FFF. What you see on the screen are twenty-four screen blocks, each consisting of eight lines gradually being filled. The twenty-four blocks also can be viewed as eight triplets, with each triplet made up of three lines, one line each at the top, middle, and bottom portions of the screen. The general screen fill pattern then is: 0, 63, 127; 8, 71, 135; 16, 79, 143; ... 62, 126, 191.

¹ [CT] Corrections to this code were taken from the June 1982 *Assembly Lines* article. Additional comments were added, following those from Bob Sander-Cederlof's Applesoft disassembly at <http://www.txbobsc.com/scsc/scdocumentor/>.

² [CT] The next 128 bytes (\$2080-\$20FF) correspond to screen lines 8, 71, and 135. The 128 bytes after that map to lines 16, 79, 143, and so forth. The first eight lines start at: \$2000, \$2400, \$2800, \$2C00, \$3000, \$3400, \$3800, \$3C00. See chapter 31 for details.

It is not essential at this point that you be entirely fluent in terms of which line corresponds to which memory range; only that you realize that the screen does not fill in quite the pattern that might otherwise be expected. Fortunately, the routine just given can calculate the base address of any horizontal line we wish to access.

Another Problem: Shifting Colors

Enter the following:

```
HGR
HCOLOR = 1
HPlot 0,0
CALL 62454
HCOLOR = 5
HPlot 0,0 TO 100,100
```

The first two steps are fairly innocent; they merely select and clear the hi-res page, then set the color to green. Trying to HPlot 0,0 gives the first problem: it doesn't seem to work. This is consistent with the warning given earlier, that even-numbered colors plot only even coordinates, and odd-numbered colors plot only odd coordinates. Green, being an odd-value color, is not plotted at $X = 0$.

The CALL 62454 is a call to a routine that clears the screen to the last color plotted (whether or not the result was visible). After you set the color to orange (HCOLOR = 5), an attempt to draw a diagonal line produces a series of rectangles. What accounts for both of these effects?

You'll recall that 40 bytes per line are used to hold the data to display the 280 dot positions on each line. There are eight bits in a byte, giving us a total of 320 bits to work with. As it happens, only seven of each eight are used in mapping the displayed screen dots ($7 \times 40 = 280$ dots).

Consider the illustration below:

Address:	\$2000							\$2001							\$2002									
Bit:	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
Color:	V	G	V	G	V	G	V	0	G	V	G	V	G	V	G	0	V	G	V	G	V	G	V	0
	B	0	B	0	B	0	B	1	0	B	0	B	0	B	0	1	B	0	B	0	B	0	B	1
X coordinate:	0	1	2	3	4	5	6	-	7	8	9	10	11	12	13	-	14	15	16	17	18	19	20	-

Figure 20-1: Bit Positions and Screen Colors

What Figure 20-1 shows is the color and position assignment of each bit within the first three bytes of memory for page one of the hi-res screen display. Although only the first three bytes of line 0 are shown, the pattern holds for the entire display.

Note the following major points:

1. Not every color can be displayed at every X coordinate. Specifically, even colors (violet = 2, blue = 4) are available only on even X coordinates. Odd colors (green = 1, orange = 5) are available only at odd X coordinates.
2. Within any byte, bit 7 is used to determine which row—top or bottom—is selected. This means that for any particular group of seven dot positions, represented by a single byte, only the colors in *either* the top or bottom rows can be shown at one time. For example, it is *not* possible to have green and orange dots displayed simultaneously within the same seven-dot group.
3. The order of the colors within every other byte is reversed with respect to its neighbors. This is to ensure that the individual colors properly alternate with successive X positions, such as between bytes 0 and 1, 1 and 2, and so on.

The color chart is shown below:

Set 1	Set 2
0 = Black1	4 = Black2
1 = Green	5 = Orange
2 = Violet	6 = Blue
3 = White1	7 = White2

Now perhaps it will make a little more sense. Set 1 colors are all those selected when the high-order bit is off (bit 7 = 0). Set 2 are all those selected when the high-order bit is on (bit 7 = 1). Any attempt to plot a point from one set will convert any existing dots from the other set, provided all dots are defined within a common byte. Obviously,

plotting a dot at X coordinate 7 (byte \$2001) will not have any effect on dot positions 0 to 6, since they are stored in a separate byte (\$2000).

White is drawn by turning on two adjacent dots, either a violet-green pair for white1, or a blue-orange pair for white2. Conversely, black is formally done by turning off two dots at once, the pair of which would correspond to the ones used for a white plot as just described.

Within a particular byte, bit 7 will always be left in a state determined by the nature of the last color plot, regardless of how many dots were previously in some other particular condition. This is why the earlier diagonal line plot acted so strangely. By clearing the screen to green, every screen byte was set so as to have the green bits on and the violet bits off (bit 7 = 0). See Figure 20-2.

Address:	\$2000								\$2001								\$2002							
Bit:	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
Value:	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	0
Color:	G	G	G	-	G	G	G	-	G	G	G	-												
X coordinate:	0	1	2	3	4	5	6	-	7	8	9	10	11	12	13	-	14	15	16	17	18	19	20	-

Figure 20-2: Bit Values for Green Pixels

Location \$2000, for example, would hold the value \$2A. Since the pattern is shifted for \$2001, an all-green dot group would correspond to the value \$55. To add to the confusion, remember that Figure 20-1 shows the bits in the reverse order from the notation normally used in this book. Ordinarily we'd show location \$2000 holding a \$2A in binary notation as: 00101010. Since the screen dots are displayed by least-significant position first, though, this is reversed when showing a screen display to make it easier to interpret:

\$2A = 00101010 → (reverse to match Figure 20-1) → 01010100

and for the other bytes:

\$5A = 01010101 → (reversed) → 10101010

When HPLLOT tried to draw an orange dot at 0, 0 we would ordinarily expect no effect. However, the high bit was reversed, and this converted the display of all current green dots to orange.

At all odd coordinates the direct plot is successful, but all remaining dots in the particular byte still converted to their high-bit-on equivalents.

Figure 20-3 shows the contents of \$2000 to \$2002 after the orange HPLLOT.

Address:	\$2000								\$2001								\$2002							
Bit:	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
Value:	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1	1	0	1	0	1	0	1	0	1
Color:	0	0	0	-	0	0	0	-	0	0	0	-												
X coordinate:	0	1	2	3	4	5	6	-	7	8	9	10	11	12	13	-	14	15	16	17	18	19	20	-

Figure 20-3: Bit Values for Orange Pixels

Another smaller but equally annoying example is shown by this simple procedure:

```
HGR
HCOLOR = 1: HPLOT 1,0
HCOLOR = 5: HPLLOT 5,0

HGR
HCOLOR = 1: HPLLOT 1,0
HCOLOR = 5: HPLLOT 6,0

HGR
HCOLOR = 1: HPLLOT 1,0
HCOLOR = 5: HPLLOT 7,0
```

Step through each statement carefully, noting what happens after the attempt to plot the orange dot. In the first case, the first green dot is converted even though the dots are visually separated. This is because they are both determined within the same byte. In the second case, even though the second dot is not plotted, the conversion still occurs. In the third case, the second plot uses a second and distinct byte, so the first dot is unaffected regardless of the color of the second plot.

Other Problems: When Is White Not White?

Answer: when you're plotting only one dot at a time.

In the last few programs involving the movement of hi-res dots, you may have noticed that at slow speeds the color of the dot alternated between violet and green depending on its position. Similarly, even though we specified white as the color to be used in the box frame drawn at the beginning of each program, the left vertical line was violet while the right one was green.

This is because white does not actually turn on two dots at once. What it really does is let either dot (violet/green or orange/blue) be acceptable for a given HPLLOT. White appears only when two adjacent dots are drawn, usually as a result of a line being drawn with some degree of horizontal tilt to it.

In the moving-dot programs, the dot appears white when moving at higher speeds because the alternation between colors occurs quickly enough that your eye tends to do the blending on a time basis, rather than the usual positional one.

Super Hi-Res Graphics

The last topic for this chapter is not a problem, but rather an unheralded benefit of this crazy system of screen displays. You may have noticed in the previous example that when the second dot was plotted, the green dot moved slightly to the right when it changed to orange. Up until now, you've been led to believe that the violet/blue or green/orange options for each bit represented a

unique screen position—a single dot. For the 280-point model of the screen, they do. For example, either violet or blue can be plotted with an H PLOT 0,0 statement.

In reality, however, a more accurate representation can be constructed as in Figure 20-4.

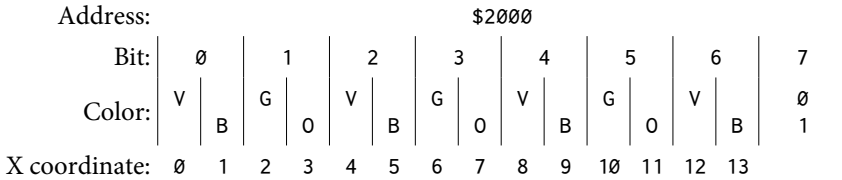


Figure 20-4: Bit Positions and Colors for 560-dot Mode

In this model, you can see that the high-bit-on colors are shifted a half position to the right of the high-bit-off colors. What this means is that you can plot points in a 560-dot mode, giving a much better resolution than the usual 280-point mode. This involves enough calculation that it's best done in assembly language. In the next chapter we'll investigate the techniques for plotting in all of these various modes using some new routines.

Hi-Res Plotting in Assembly

June 1982

In the previous chapter we looked at how the Apple hi-res screen is set up and at how each dot on the screen is linked to a bit position of a byte in memory.

In this chapter we'll present a more detailed explanation of plotting a point and, more specifically, provide routines for some new ways of plotting to the hi-res screen.

Normal Point Plotting

In Figure 20-1 (Chapter 20) we saw how the hi-res screen colors are mapped out in memory.

You'll remember that we could access either the violet/green or blue/orange dot pairs depending on whether the high-order bit (bit 7) of the byte in question was set. To plot a color dot on the screen we need to carry out the following steps:

1. Use the Y coordinate to determine on which horizontal screen line to plot. Because the lines are not mapped continuously, a special routine is used to calculate the base address. In this case the term refers to the address associated with the first byte on the line given by Y.

In normal Applesoft, this base address is called GBAS ("Graphics Base address") and is stored in the byte pair \$26, \$27. Location HPAG (\$E6 = Hi-res PAGe) is used to indicate whether the plot is to be on page 1 or page 2 of the hi-res screen.

As it happens, we can use the HPOSN (\$F411) routine in Applesoft to do this calculation for us, but the listing in chapter 20 (HIRES BASE ADDRESS CALCULATOR) is provided for your entertainment, and for possible use if you should decide to write an Applesoft-independent routine.

2. Once the base address of the horizontal line has been determined, the position of the byte relative to the left edge needs to be established. Because seven dots are stored in each byte, the byte we need to access can be determined by dividing the X coordinate by 7. This result is stored in location HNDX (\$E5 = Horizontal iNDeX). It is used by putting the contents of \$E5 into the Y-Register for an LDA (\$26), Y operation—but more on that later.

	For X = Even	For X = Odd
\$F6F6:	\$00 = 0000 0000 Black1 (0)	\$00 = 0000 0000
\$F6F7:	\$2A = 0010 1010 Green (1)	\$55 = 0101 0101
\$F6F8:	\$55 = 0101 0101 Violet (2)	\$2A = 0010 1010
\$F6F9:	\$7F = 0111 1111 White1 (3)	\$7F = 0111 1111
\$F6FA:	\$80 = 1000 0000 Black2 (4)	\$80 = 1000 0000
\$F6FB:	\$AA = 1010 1010 Orange (5)	\$D5 = 1101 0101
\$F6FC:	\$D5 = 1101 0101 Blue (6)	\$AA = 1010 1010
\$F6FD:	\$FF = 1111 1111 White2 (7)	\$FF = 1111 1111

Figure 21-1: Applesoft Color Masks

3. The color mask needs to be set up. The color mask is a bit pattern that shows which bits in a byte are acceptable possibilities for a plot. The color mask is stored in location \$E4 (COLBYTE). Rather than literally calculating, Applesoft stores all of the possible color masks starting at location \$F6F6 (see Figure 21-1).

Ones and zeros are used to indicate which dots are on and which are off for the color indicated. Black1 is the simplest: it is achieved by turning every dot off. White1 is its converse, achieved by turning every dot on. Note that bit 7 does not correspond to a displayed dot and is left a 0 (high bit off).

If you compare the color masks for green and violet to the chart in Figure 20-1, you'll note that the ones match the available dots for the given color in a byte. Remember, the order of the bits is reversed when mapping to the screen, so that bits 0 to 6 are mapped left to right on the screen.

The second set of masks in Figure 21-1 are the colors with the high-bit set (bit 7 = 1). The same pattern as before is used, except that the high bit is set for all four colors.

Looking at Figure 20-1 again, you'll note that the masks shown on the left will work for all even-addressed bytes, that is, bytes such as \$2000, \$2002, and so on. For the odd-addressed bytes (\$2001, \$2003, and so on), the colors are shifted one bit position. When HPOSN is called, along with determining GBAS, it checks the HNDX calculated and, if that is an odd address, shifts the color byte. The result, whether shifted or not, is always put in location \$1C (HCOLOR1). The results of such a possible shift are shown on the right side of Figure 21-1.

(An interesting result of this process is that you cannot clear the entire screen to an actual color [green, violet, blue, or orange] by filling memory with a single value. Try it. Clear the hi-res screen with an HGR, then enter the Monitor with CALL -151. Then type in:

```
*2000:2A
*2001<2000.3FFFM
```

The screen should clear to alternating vertical bars of green and violet.)

4. Now the actual bit position of interest needs to be selected. This actually has already been done by HPOSN. The result of the X coordinate divided by seven was put in HNDX, and the remainder of that division just happens to correspond to the actual bit position within the byte we want. The only remaining problem is that the result is a number from 0 to 6, and what we need is a byte with only that particular bit turned on. This is again derived from a table within Applesoft (in this case starting at \$F5B2—see Figure 21-1). The result from this table is then put in location \$30 (HMASK).

\$F5B2:	\$81 =	1000 0001
\$F5B3:	\$82 =	1000 0010
\$F5B4:	\$84 =	1000 0100
\$F5B5:	\$88 =	1000 1000
\$F5B6:	\$90 =	1001 0000
\$F5B7:	\$A0 =	1010 0000
\$F5B8:	\$C0 =	1100 0000

Figure 21-2: Bit Mask

Now at last we're ready to do the actual plot.

The plotting sequence (normally found at \$F45A) looks like this:

```

F45A-  A5 1C      LDA  $1C          ; HCOLOR1
F45C-  51 26      EOR  ($26),Y      ; (GBAS),Y
F45E-  25 30      AND  $30          ; HMASK
F460-  51 26      EOR  ($26),Y      ; (GBAS),Y
F462-  91 26      STA  ($26),Y      ; (GBAS),Y
F464-  60         RTS

```

This last operation is probably best clarified with an actual example.

Given:

```

HGR
HCOLOR = 1
H PLOT 15,0

```

Procedure:

1. JSR \$F3E2 (HGR). Clears the hi-res screen. Sets HPAG (\$E6) to #\$20.
2. LDX #\$01
JSR \$F6F0 (HCOLOR)
This puts the mask value %00101010 in HCOLOR1 (\$E4).
3. LDX X (low-order byte of the X coordinate)
LDY X+1 (high-order byte of X)
LDA Y (Y coordinate)
JSR HPOSN

Note that the percent sign (%) in the mask value is used to indicate the binary form of a number. This form is used in the remark portions of many of the source listings in this book as an added aid to the explanations. Although some assemblers allow binary numbers in the operand, we have limited their use here to the remark field to reduce compatibility problems.

The procedure given above will:

- a) Calculate the base address using the page index at \$E6 (usually \$20). In this case the result will be \$2000. The result is stored in GBAS, GBAS+1 (\$26, \$27)
 - b) Divide 15 (the X coordinate) by 7. The result (2) is put in HNDX (\$E5). The remainder of the division (1) is used to access the bit mask table. The result of this table lookup (%10000010 found at \$F5B2, X where X=1) is put in HMASK (\$30).
 - c) Check HNDX to see if the byte offset is odd. If so, shift the color byte mask. Since in this case \$E5 holds a 2, no shift is required. Thus the color mask %0010 1010 is put in HCOLOR1 (\$1C) in preparation for the plot.
4. JSR \$F45A (HPLLOT). This completes the process with:

```
LDY HNDX      ($E5) = '2'
LDA HCOLOR1   ($1C) = %0010 1010
EOR (GBAS),Y  ($2002) = %0000 0000
                  %0010 1010 (EOR'd)
AND HMASK     ($30) = %1000 0010
                  %0000 0010 (AND'd)
EOR (GBAS),Y  ($2002) = %0000 0000
                  %0000 0010 (EOR'd)
STA (GBAS),Y  ($2002) = %0000 0010
```

screen looks like: 0100 000- Green dot lights!

The net effect of step 4 is to say: "Look at the bit mask pattern and compare it to the color mask. If there is a one in the color mask at the given dot position, turn that dot on (set the bit to 1) If there's a 0 at that position, turn the dot off (clear bit to 0)."

Alternate Plotting Modes

So far, all we have really done is to explain further something we were already using. This new explanation makes possible some alternative ways of plotting to the hi-res screen. In fact, by using the existing Applesoft routines, the new routines are rather short and, best of all, easy to explain. If you are unsettled right now about the finer details of the masking operations, don't worry. The real point of all that is to give you some feel for the general processes involved.

For starters, let's review some basic problems encountered so far with the normal Applesoft HPLLOT. The first arises when you try to plot using just one color. By setting HCOLOR equal to 1, 2, 5, or 6, we limit the possible dots which can be plotted to every other dot on the normal screen. This can be disconcerting when you have a statement like:

```
HCOLOR = 1: HPLLOT 100,100
```

and nothing happens. The reasons for this were discussed in earlier chapters, but now it should be even more obvious that the color mask specifies only odd-dot positions for HCOLOR = 1, making it impossible to plot at X = 100.

The second problem occurs when you're plotting with HCOLOR = 3 or HCOLOR = 7. Even though we have specified white, an attempt to plot a single point always comes out as a colored dot. It is only when drawing more than one point (such as in a line) that white appears. Let's examine possible solutions to these problems.

140-Point Resolution Mode

For the first problem of invisible points, one solution is to accept that there are only 140 points available for a given color and to alter our frame of reference to recognize that reality. An easy way of doing this is to always work with an X coordinate value in the range of 0 to 139, and then to double the value when actually doing the H PLOT. The main drawback to this approach is the speed loss due to the multiplications, and the fact that odd color values must also be shifted by one (since odd colors can only plot at odd X positions). The situation now would look like this:

```

HCOLOR = 2          HCOLOR = 1
X = 15: Y = 30     or  X = 20: Y = 30
H PLOT X*2, Y      H PLOT X*2 + 1, Y

```

Another approach is to create an assembly-language routine to do this for us automatically. Here's the source listing for such a routine.

```

1 *****
2 *           AL21-HIRES PLOT .140      *
3 *                                           *
4 *****
5 *
6 *
7 *           OBJ   $300
8 *           ORG   $300
9 *
10  CHKCOM   EQU   $DEBE
11  FRMNUM   EQU   $DD67
12  GETADR   EQU   $E752
13  LINNUM   EQU   $50
14  COMBYTE  EQU   $E74C
15 *
16  X        EQU   $E0
17  Y        EQU   $E2
18 *
19  HCOLOR   EQU   $F6F0
20  HGR      EQU   $F3E2
21  H PLOT   EQU   $F457
22  H PLOT2  EQU   $F45A
23  COLBYTE  EQU   $E4

```

```

                24 *
0300: 20 BE DE 25 ENTRY JSR CHKCOM
0303: 20 67 DD 26 JSR FRMNUM
0306: 20 52 E7 27 JSR GETADR
                28 *
0309: 06 50 29 CALC ASL LINNUM
030B: 26 51 30 ROL LINNUM+1 ; X*2
                31 *
030D: A9 02 32 LDA #$02 ; %0000 0010
030F: 24 E4 33 BIT COLBYTE
0311: F0 06 34 BEQ C1 ; NO MATCH COLOR EVEN
0313: E6 50 35 INC LINNUM
0315: D0 02 36 BNE C1
0317: E6 51 37 INC LINNUM+1
                38 *
0319: A5 50 39 C1 LDA LINNUM
031B: 85 E0 40 STA X
031D: A5 51 41 LDA LINNUM+1
031F: 85 E1 42 STA X+1
                43 *
0321: 20 4C E7 44 GETY JSR COMBYTE
0324: 8A 45 TXA ; PUT Y-COORD IN ACC
0325: A6 E0 46 PLOT LDX X
0327: A4 E1 47 LDY X+1
0329: 20 57 F4 48 JSR HPLOT
                49 *
032C: 60 50 DONE RTS
032D: C1 51 CHK

```

This program is designed to be called from Applesoft, serving as a subroutine for an undefined overall program. The advantage of the routine is that HCOLOR may be set to any value, although white will still plot only one color. Values for the X coordinate may range from 0 to 139.

Assuming that the routine is loaded starting at location \$300 (768 decimal), the syntax for calling it would be:

```
CALL 768, X, Y
```

where X and Y are the coordinates for the desired plot.

Examining the listing, you will see that the first step is to use the calls to Applesoft on lines 25 through 27 to retrieve the X coordinate from Applesoft. The resulting two-byte representation for the value will end up in LINNUM (\$50, \$51).

Once we have the value for X, the remaining process is very straightforward. The X coordinate is doubled by the pair of left shifts on lines 29 and 30. Next, the color byte is checked to see if the HCOLOR previously selected was an odd or even color value. A brief look at the color mask chart in Figure 21-1 shows that bit 1 (rather than bit 0) is the key to whether a color is odd or even. If the color is odd, LINNUM is incremented by one to select the next odd X-coordinate position.

The Y coordinate is then retrieved using COMBYTE. Since Y cannot be larger than 191, the one-byte retrieval routine can be used.

At that point, the usual call to H PLOT is done with the new X coordinate.

A little rumination on this routine should convince you that it is functionally identical to this BASIC algorithm:

```

0  HGR: HOME: VTAB22
10  INPUT "HCOLOR";C : HCOLOR = C
20  INPUT "COORDINATES: "; X,Y
30  X = X * 2
40  IF C / 2 <> INT (C / 2) THEN X = X + 1
50  H PLOT X,Y

```

The assembly-language routine given can always be used directly from other assembly-language programs by deleting lines 25 through 27 and changing 44 and 45 to read LDY Y. The routine would then be called by putting the desired X coordinate in LINNUM (\$50, \$51), and the Y coordinate in Y (\$E2).

560-Point Resolution Mode

The disadvantage of the 140-point method just shown is that the resolution of the graphics is obviously limited. This is particularly apparent in attempts to draw near-vertical lines; it's easy to observe the degree of stair-stepping that occurs. Low-resolution plotting modes produce very broken near-vertical lines.

If color is not a concern (such as when using a black-and-white monitor), then why not just plot using white? Since we won't know that the colors are actually varying depending on the X coordinate specified, a black-and-white display will look fine.

Well, if that's the case, then you might as well go for all you can get and use the 560-point mode. The theory to this mode is that the high-order bit of each screen byte can be used to choose between dots shifted one-half of a position with respect to the usual 280-point mode. The argument against this method is that the plotting of dots within the same byte can distort the first byte plotted.

For example, if the first dot plotted is on the farthest left position possible (high bit off), then a successive plot of any HCOLOR with the high bit set (HCOLOR = 4 through 7) will change the color of the dot and shift it to the right. As it happens, this is not much of an argument since the same holds true for the normal 280-point mode, and even for the 140-point mode. The inescapable fact is that plotting two colors with conflicting high-bit conditions within the same byte will always affect the first dot plotted. If the distortion is unavoidable then you might as well enjoy the benefits of the higher resolution, especially if you're going to have to cope with the distortion problem anyway.

Without further introduction, here then is a routine implementing the 560-point plotting mode.

Like the PLOT.140 routine, this is assumed to be loaded at \$300 and would be called in a manner identical to that for the previous routine:

```
CALL 768, X, Y
```

The main difference here is that X can now have a range of 0 to 559, and that HCOLOR is always set to white. As with normal Applesoft, what this really means is that we'll take any color we can get for a given plot, and that true white will result only when dots are plotted adjacent to each other. Here's the listing for this routine:

```

1 *****
2 *      AL21-HIRES PLOT.560      *
3 *                                *
4 *****
5 *
6 *
7 *      OBJ $300
8 *      ORG $300
9 *
10 CHKCOM EQU $DEBE
11 FRMNUM EQU $DD67
12 GETADR EQU $E752
13 LINNUM EQU $50
14 COMBYTE EQU $E74C
15 *
16 X      EQU $E0
17 Y      EQU $E2
18 *
19 HPLOT  EQU $F457
20 COLBYTE EQU $E4
21 *
0300: 20 BE DE 22 ENTRY   JSR  CHKCOM
0303: 20 67 DD 23         JSR  FRMNUM
0306: 20 52 E7 24         JSR  GETADR
25 *
0309: 46 51 26 CALC    LSR  LINNUM+1
030B: 66 50 27         ROR  LINNUM      ; X/2
030D: A9 7F 28 C0     LDA  #$7F        ; %0111 1111
030F: 85 E4 29         STA  COLBYTE
0311: 90 04 30         BCC  C1        ; X=EVEN
0313: A9 FF 31         LDA  #$FF        ; %1111 1111
0315: 85 E4 32         STA  COLBYTE
33 *
0317: A5 50 34 C1     LDA  LINNUM
0319: 85 E0 35         STA  X
031B: A5 51 36         LDA  LINNUM+1
031D: 85 E1 37         STA  X+1
38 *
031F: 20 4C E7 39 GETY   JSR  COMBYTE
0322: 8A      40         TXA                      ; PUT Y-COORD IN ACC
0323: A6 E0 41 PLOT    LDX  X
0325: A4 E1 42         LDY  X+1
0327: 20 57 F4 43         JSR  HPLOT

```

```

                                44  *
032A: 60                        45  DONE   RTS
032B: 9B                        46                CHK

```

The operation of this routine is also fairly simple. As with the PLOT . 140 program, the value for X is retrieved from the calling program. In this case, though, CALC divides the passed value by two. Note that a left-shift operation is used, not the right shifts (for a multiply) that were used in the 140-mode.

You'll recall that LSR LINNUM+1 (LSR = Logical Shift Right) will shift all bits in LINNUM+1 (the high-order byte) to the right one position, forcing a 0 at the rightmost position and putting the old bit 0 in the carry. This is immediately followed by the ROR ("rotate right") instruction which again shifts all the bits in LINNUM (the low-order byte), puts the carry into bit 7, and drops the last bit 0 into the carry, thus replacing the old value. For example:

```

X-COORD = 289 = $121 = %0000 0001  0010 0001
                                LINNUM+1  LINNUM

LSR LINNUM+1:  %0000 0001  →  %0000 0000 (Carry=1)
ROR LINNUM:    %0010 0001  →  %1001 0000 (Carry=1)

```

The rather coincidental beauty of this is that the carry flag will end up being set or cleared depending on whether the original value for X was odd or even. This is needed because in the 560-point mode, we'll use the odd or even nature of X to determine whether to set the high bit.

X (560)	X (280)	Color mask to use
0	0	White1 (bit 7 = 0)
10	5	White1 (bit 7 = 0)
201	100	White2 (bit 7 = 1)
501	250	White2 (bit 7 = 1)

Basically what we do is to divide the X coordinate by two to get a value acceptable to normal Applesoft, and then force the color to be either white1 or white2 depending on how we want the high bit set in the final plot.

Lines 28 through 32 set the color mask to the appropriate value by checking the carry flag to see if the original value of X was odd or even. Then LINNUM is transferred to our actual X-coordinate bytes. The routine is then completed with the usual call to H PLOT, as was done in the PLOT . 140 routine.

This process could be simulated from Applesoft with the following routine:

```

0  HGR: HOME: VTAB 22
10  INPUT "COORDINATES? "; X, Y
20  HCOLOR = 3 : REM WHITE1
30  IF X / 2 <> INT(X / 2) THEN HCOLOR = 7 : REM WHITE2 FOR X=ODD
40  X = X / 2
50  H PLOT X, Y

```


It's likely, however, that you'll find the assembly-language routine considerably faster, and certainly much easier to implement.

A Demonstration Program

To give you something to show off these routines, here's a program in Applesoft that calls both routines and shows the differences in their appearance.

```

10 D$ = CHR$(4): REM AL21.PLOTLINE.A
100 REM NORMAL TEST
110 HGR : HCOLOR = 3
120 FOR I = 0 TO 100
130 HPLOT I, I
140 NEXT I
200 REM PLOT.140 TEST
205 PRINT D$;"BLOAD AL21.PLOT140,A$300"
210 FOR I = 0 TO 100
220 CALL 768,I,I
230 NEXT I
300 REM PLOT.560 TEST
305 PRINT D$;"BLOAD AL21.PLOT560,A$300"
310 FOR I = 0 TO 100
320 CALL 768,I,I
330 NEXT I

```

Notice that this program loads each routine from a disk file as it's needed. Basically this illustrates the steepest vertical angle at which a line can be drawn without any noticeable stair-stepping, or breaking, in the line. It also conveniently shows a perhaps unexpected change in the actual visual result of the plot, even though all three lines were done with similar FOR-NEXT loops.

Normally, the 280-point mode is conveniently proportional. That is to say, a move of five points horizontally on the screen is about the same actual distance on the screen as a move of five points vertically. This ensures that a square will in fact look "square" when drawn on the screen. Thus the first plot is at the "proper" 45 degrees when drawn using HPLOT 1, 1.

When the number of screen points is halved, as in the case of the PLOT.140 routine, the result will be to "stretch" the screen horizontally by a factor of two. Similarly, packing in twice as many points (namely 560 versus 280) across has the effect of compressing the screen. These effects must be considered when doing geometric designs on the screen.

We'll leave it as an exercise for you to draw three parallel lines using each of the three modes.

By now, you've probably also noticed some minor flaws in the clarity of the 560-point line. In the next chapter we'll explore the matter further, discovering why the faint spots occur and how to fix them.

Even Better Hi-Res Plotting

July 1982

The previous chapter concluded with a demonstration program that showed the relative appearances of a line drawn with the normal H PLOT command as well as with special 140- and 560-point mode plotting routines.

The entire plotting process was based on a model of point display in which each point on the screen corresponds to the status of a particular bit within a memory byte. For general plotting, Figure 20-1 (in Chapter 20) illustrates the corresponding color points.

The 140-point mode was created to ensure that for any H PLOT-type action, a consistent color dot would always be plotted. This consistency is not ordinarily available in the Apple's usual 280-point mode.

For instances in which color is not a concern, an alternate scheme was devised that would be indifferent to the color of the dot illuminated (as the viewer would be when using a black-and-white monitor). An added feature of this scheme allows a resolution of 560 points per line. This was done by using the high-order bit of each byte to shift a given dot one-half of a position.

When the final demonstration program was run, the last line was drawn in the 560-point mode. You may have noticed, though, that certain points on the line were rather faint. This brings us to the discussion of one of the last (?) bugs in the hi-res graphics routines.

Change the previous chapter's test program to appear as follows:

```

10 D$ = CHR$(4)
40 HOME: INPUT "BLACK1 OR BLACK2? (1 OR 2)"; I
100 REM NORMAL TEST
110 HGR: HCOLOR = I*4 - 4: H PLOT 0,0: CALL 62454: HCOLOR = 11 - I*4
120 FOR I = 0 TO 100
130 H PLOT I,I
140 NEXT I
200 REM PLOT.140 TEST
205 PRINT D$;"BLOAD AL21.PLOT140,A$300"
210 FOR I = 0 TO 100
220 CALL 768,I,I
230 NEXT I
300 REM PLOT.560 TEST
305 PRINT D$;"BLOAD AL21.PLOT560,A$300"
310 FOR I = 0 TO 100
320 CALL 768,I,I
330 NEXT I

```

When you run this program, enter either 1 or 2 to specify which “flavor” of black you want for the background. Under normal circumstances, an HGR statement clears the background to black1, (high bit off on each byte) and plots are done using white1. This program changes that by using the alternate white for the background selected; that is to say, if you select black1 for the background, white2 will be used to plot. If you select black2, white1 will be used.

Examining the listing, then, you’ll notice that line 110 sets HCOLOR to black1 or black2, does the required plot, and then clears the background to that color. A CALL 62454 will always clear the current hi-res screen to the last color plotted (see page 134 of your *Applesoft II BASIC Programming Reference Manual* for a description of this).

Following the screen clear, white is set in the alternate mode described earlier.

Lines 100–140 draw a diagonal line point-by-point as was done in the previous chapter, but now the line should appear to have a few faint spots in it. If you choose black2 as the background, the line will have places where the dots appear slightly larger than you’d have expected.

Similar effects can be observed in the 140- and 560-mode lines.

Interactions between Adjacent Bytes

The entire premise of the 560-point mode was that the high-order bit of each byte affected the final display position of each other bit within it. We have seen how changing the status of bit 7 (the high-order bit) may shift a given dot one-half of a position, depending on whether the bit is set.

Now for the new wrinkle. It turns out that for dots associated with bit 6 of a byte, the high-order bit of the *next* byte in memory *also affects the display of the first byte*.

As an example, first clear the hi-res screen with an HGR and then enter the Monitor via the usual CALL -151.

Now enter the following values into memory. You should see an effect similar to the description at the right of each statement.

```
*2138: 40 (Dot is plotted; width = 1 unit)
*2139: 80 (The dot extends; new width = 1.5 units)
*2139: 00 (The dot is back to normal; width = 1 unit)
*2138: C0 (The dot grows fainter; width = 0.5 units)
*2139: 80 (The dot is back to normal; width = 1 unit)
```

The references to a *width* are an approach to explaining what happens. If you have a black-and-white monitor, the relative visual strengths of the dots can be related to an apparent width of the dots when illuminated on the monitor screen. On a color television or monitor the widths aren’t discernible, but differences in color and brightness can be seen.

Before any further explanations, let's re-examine the 560-point model.

You'll recall that although the violet and blue dots officially occupy the same screen position horizontally, in actuality a half-position shift may happen, depending on whether the high bit is set. When \$2138 was set to \$40, we were, as such, plotting position 12 on the display. When \$2138 was set to \$C0, position 13 was illuminated. The flaw can be explained by imagining that the high-order bit of \$2139 (the next byte after \$2138) can also produce a slight shift on a dot produced by bit 6 of \$2138. The general rule is that for any dot produced by bit 6, the succeeding byte of memory must have a high-order bit (bit 7) set to the same value as bit 7 of the byte being plotted.

If this rule is not observed, one of two things will happen:

1. If bit 7 of the displayed byte is clear and the next byte is set, the dot will be extended or enlarged—slightly.
2. If bit 7 of the displayed byte is set and the next byte is clear, the dot will be reduced slightly, resulting in a fainter image.

An interesting result is the conclusion that even the “normal” method of plotting (that is, white) will give ragged displays when adjacent bytes have contrary high-bit settings!

Some “New and Improved” Routines

Well, then...that has been a lot to digest. In fact, at this point you might just want to take a break to let everything sink in, maybe fix yourself a nice cup of tea and meditate on it for a while.

Glad to see you again! One of the difficulties in presenting the material in the last few chapters has resulted from the discovery that hi-res graphics is not all that logical. Much of hi-res graphics seems to be very empirical in nature. That is, it's more a matter of accepting that things are a certain way as derived from experimentation, than of trying to account for the innermost workings of a seemingly random event. (In this case, the innermost workings are related to the purely electronic world of wires, video protocols, and so forth, which is mostly incidental to the programmer!)

The worst is probably over, though. At this point you should have at least a general feel for how the dots are mapped on the screen. Let's now create some final routines that encompass the various quirks of the hi-res system as it presently exists.

PLOT.140+

The first one to fix is the 140-point mode routine. For all routines the approach will be very direct:

1. Determine whether the dot being plotted involves bit 6 of the byte of memory in question. If not, don't worry.
2. If bit 6 is used, check the status of the high-order bit (bit 7) of the byte.
3. Fix the high-order bit of the next byte in memory, if needed, to match that of the first byte. Here's the new routine to do just that:

```

1 *****
2 *      AL22-HIRES PLOT.140+ *
3 *                                     *
4 *****
5 *
6 *
7 *      OBJ $300
8 *      ORG $300
9 *
10 *      CHKCOM EQU $DEBE
11 *      FRMNUM EQU $DD67
12 *      GETADR EQU $E752
13 *      LINNUM EQU $50
14 *      COMBYTE EQU $E74C
15 *
16 *      X      EQU $E0
17 *
18 *      HPLOT EQU $F457
19 *      COLBYTE EQU $E4
20 *      HMASK EQU $30
21 *      HNDX EQU $E5
22 *      GBAS EQU $26
23 *
0300: 20 BE DE 24 ENTRY JSR CHKCOM
0303: 20 67 DD 25 JSR FRMNUM
0306: 20 52 E7 26 JSR GETADR
27 *
0309: 06 50 28 CALC ASL LINNUM
030B: 26 51 29 ROL LINNUM+1 ; X*2
30 *
030D: A9 08 31 LDA #$08 ; %00001000
030F: 24 E4 32 BIT COLBYTE
0311: F0 06 33 BEQ C1 ; NO MATCH COLOR EVEN
0313: E6 50 34 INC LINNUM
0315: D0 02 35 BNE C1
0317: E6 51 36 INC LINNUM+1
37 *
0319: A5 50 38 C1 LDA LINNUM
031B: 85 E0 39 STA X
031D: A5 51 40 LDA LINNUM+1
031F: 85 E1 41 STA X+1
42 *

```

```

0321: 20 4C E7 43 GETY JSR COMBYTE
0324: 8A 44 TXA ; PUT Y-COORD IN ACC
0325: A6 E0 45 PLOT LDX X
0327: A4 E1 46 LDY X+1
0329: 20 57 F4 47 JSR HPLOT
48 *
032C: A5 30 49 CHK LDA HMASK
032E: C9 C0 50 CMP #C0 ; %11000000
0330: D0 11 51 BNE DONE
52 *
0332: A4 E5 53 FIX LDY HNDX
0334: C8 54 INY
0335: B1 26 55 LDA (GBAS),Y
0337: 24 E4 56 BIT COLBYTE
0339: 30 04 57 BMI HISET
033B: 29 7F 58 HICLR AND #$7F ; %01111111
033D: 10 02 59 BPL STORE ; ALWAYS
033F: 09 80 60 HISET ORA #$80 ; %10000000
0341: 91 26 61 STORE STA (GBAS),Y
0343: 60 62 DONE RTS
0344: 06 63 CHK

```

The listing through line 47 should appear similar to the previous chapter's routine. Lines 48 through 62 add a check to see whether the next byte in memory needs to be adjusted according to the three-step procedure just described.

The first step is to examine location \$30 (HMASK). You'll remember from the previous chapter that this is a mask used to indicate which bit position is to be set to plot the point. If bit 6 were set, this location will hold the value C0 (binary %11000000). Lines 49 through 51 check for this.

If a match is found, we know bit 6 was set by the plot. We must now access the next byte in memory and either set or clear bit 7 of that to match our original byte. Since HNDX (\$E5) holds the offset of the current byte (usually used by combining with GBAS (\$26) in the form LDA (GBAS,Y)), we can load the Y-Register with HNDX and then increment using the INY on line 54 to shift our attention to the next byte. The data for that byte is then loaded into the Accumulator on line 55. Now for the sleight of hand. We want to check the status of the first byte, but if we load the Accumulator we'll lose the data currently held there. To solve the problem, consider this: The color mask byte COLBYTE (\$E4) holds the mask used only moments before to do the plot. We can check the high-order bit of this value to determine the status of bit 6 in the byte accessed by the plot. Since it's bit 7 we're interested in, we can also use the BIT command to do the check.

Line 56 does a BIT COLBYTE. This will move bit 7 of COLBYTE into the Status Register, after which a BMI (Branch on MInus) or a BPL (Branch on PPlus) can be used to check how the bit was set.

In this case, the BMI is used to detect bit 7 being set. If this branch is taken, the program will skip to line 60. If not, the HICLR ("high-bit clear") section will be entered. In this section, the AND operator is used to force the clearing of the

high bit in the Accumulator. Since this will also clear the sign bit of the Status Register, the BPL following this operation is always taken.

If HISET ("high-bit set") is entered, the ORA # $\$80$ will force the setting of bit 7 of the Accumulator. (If you need more information on the logical operators, you may wish to consult chapter 12.) Line 61 (STORE) puts the contents of the Accumulator back into memory, immediately followed by the RTS which ends the routine.

PLOT.560+

This routine is also a variation on a program presented in the previous chapter and again uses a check system identical to that used in PLOT.140.

```

1 *****
2 *      AL22-HIRES PLOT .560+      *
3 *                                  *
4 *****
5 *
6 *
7 *      OBJ  $300
8 *      ORG  $300
9 *
10  CHKCOM EQU  $DEBE
11  FRMNUM EQU  $DD67
12  GETADR EQU  $E752
13  LINNUM EQU  $50
14  COMBYTE EQU  $E74C
15 *
16  X      EQU  $E0
17 *
18  HPLLOT EQU  $F457
19  COLBYTE EQU  $E4
20  HNDX   EQU  $E5
21  HBIT   EQU  $30
22  GBAS   EQU  $26
23 *
0300: 20 BE DE 24  ENTRY   JSR  CHKCOM
0303: 20 67 DD 25           JSR  FRMNUM
0306: 20 52 E7 26           JSR  GETADR
27 *
0309: 46 51 28  CALC     LSR  LINNUM+1
030B: 66 50 29           ROR  LINNUM      ; X/2
030D: A9 7F 30  C0     LDA  #$7F        ; %0111 1111
030F: 85 E4 31           STA  COLBYTE
0311: 90 04 32           BCC  C1        ; X=EVEN FROM ROR
0313: A9 FF 33           LDA  #$FF        ; %1111 1111
0315: 85 E4 34           STA  COLBYTE
35 *
0317: A5 50 36  C1     LDA  LINNUM
0319: 85 E0 37           STA  X
031B: A5 51 38           LDA  LINNUM+1
031D: 85 E1 39           STA  X+1
40 *

```

```

031F: 20 4C E7 41 GETY JSR COMBYTE
0322: 8A 42 TXA ; PUT Y-COORD IN ACC
0323: A6 E0 43 PLOT LDX X
0325: A4 E1 44 LDY X+1
0327: 20 57 F4 45 JSR HPLOT
46 *
032A: A5 30 47 CHK LDA HBIT
032C: C9 C0 48 CMP #$C0 ; %11000000
032E: D0 11 49 BNE DONE
0330: A4 E5 50 FIX LDY HNDX
0332: C8 51 INY
0333: B1 26 52 LDA (GBAS),Y
0335: 24 E4 53 BIT COLBYTE
0337: 30 04 54 BMI HISET
0339: 29 7F 55 HICLR AND #$7F ; CLEAR BIT 7
033B: 10 02 56 BPL STORE
033D: 09 80 57 HISET ORA #$80 ; SET BIT 7
033F: 91 26 58 STORE STA (GBAS),Y
59 *
0341: 60 60 DONE RTS
0342: 56 61 CHK

```

Ordinarily this would be a fine place to end this chapter, but there's one more routine worth presenting. So far what you've got is a choice between plotting a single color (PLOT.140) or taking whatever color you get in exchange for the capacity for greater horizontal resolution.

Well, with just a little more effort we can create a routine that will offer the same degree of horizontal accuracy *and* guarantee that any dot plotted will be white.

PLOT.560-White

Normally when you specify white when using Apple graphics, you're really saying, "I don't care what color," because any attempt to plot a single point will illuminate only a colored dot, not a true white dot. This is because white is really formed by plotting two adjacent dots. This is consistent with the examination of the COLBYTE bit pattern for acceptable bits to set combined with the given HMASK bit pattern for a specified horizontal position within the byte. This process of plotting was described in greater detail in the previous chapter but, as a quick refresher, remember that this combination would successfully do the equivalent of:

```

Statement:          HCOLOR = 3 : HPLLOT 3,0
Mask Patterns:     COLBYTE: %0111 1111 (HCOLOR bit mask)
                   HMASK:   %1000 1000 (bit 3 set; ignore high bit)
                   RESULT:  %0000 1000 (position 3 is set green)

```

You might imagine that if the HMASK could have been set up to have two adjacent bits set, the result might have been a true white dot:

Statement: HCOLOR = 3 : H PLOT 3,0
 Mask Patterns: COLBYTE: %0111 1111 (HCOLOR bit mask)
 HMASK: %1001 1000 (bits 3 and 4 set)
 RESULT: %0001 1000 (positions 3 and 4—white)

As it happens, this can be done, and here's the new routine to do it!

```

1 *****
2 *      AL22-HIRES PLOT.560W      *
3 *                                  *
4 *****
5 *
6 *
7 *      OBJ   $300
8 *      ORG   $300
9 *
10  CHKCOM EQU $DEBE
11  FRMNUM EQU $DD67
12  GETADR EQU $E752
13  LINNUM EQU $50
14  COMBYTE EQU $E74C
15 *
16  X      EQU $E0
17 *
18  HPOSN EQU $F411
19  H PLOT EQU $F457
20  COLBYTE EQU $E4
21  HCOLOR1 EQU $1C
22  HMASK EQU $30
23 *
0300: 20 BE DE 24  ENTRY   JSR  CHKCOM
0303: 20 67 DD 25           JSR  FRMNUM
0306: 20 52 E7 26           JSR  GETADR
27 *
0309: 46 51 28  CALC     LSR  LINNUM+1
030B: 66 50 29           ROR  LINNUM      ; X/2
030D: A9 7F 30           LDA  #$7F      ; %01111111 WHITE1
030F: 85 E4 31           STA  COLBYTE
0311: 90 04 32           BCC  C1      ; X=EVEN
0313: A9 FF 33           LDA  #$FF      ; %11111111 WHITE2
0315: 85 E4 34           STA  COLBYTE
35 *
0317: A5 50 36  C1      LDA  LINNUM
0319: 85 E0 37           STA  X
031B: A5 51 38           LDA  LINNUM+1
031D: 85 E1 39           STA  X+1
40 *
031F: 20 4C E7 41  GETY   JSR  COMBYTE
0322: 8A 42           TXA           ; PUT Y-COORD IN ACC
0323: A6 E0 43  PLOT   LDX  X
0325: A4 E1 44           LDY  X+1
0327: 20 11 F4 45           JSR  HPOSN
032A: A5 30 46           LDA  HMASK
032C: 0A 47           ASL
032D: 05 30 48           ORA  HMASK

```

```

032F: 85 30   49           STA  HMASK
0331: 20 5A F4 50           JSR  HPL0T+3
      51 *
0334: 24 30   52  CHK      BIT  HMASK
0336: 50 22   53           BVC  DONE      ; BIT 6 CLEAR
      54 * BIT 6 CLEAR: POSITION = 0-9
      55 * BIT 6 SET: POSITION = 10-13
      56 *
0338: 24 1C   57  CHK2     BIT  HCOLOR1
033A: 10 06   58           BPL  HICLR      ; BIT 7 TEST
033C: A9 FF   59  HISET    LDA  #$FF      ; WHITE2
033E: 85 1C   60           STA  HCOLOR1
0340: D0 04   61           BNE  CHK3      ; ALWAYS
0342: A9 7F   62  HICLR    LDA  #$7F      ; WHITE1
0344: 85 1C   63           STA  HCOLOR1
      64 *
0346: A9 20   65  CHK3     LDA  #$20      ; %00100000
      66 *
      67 * HMASK: %11100000 IF POSITION = 10,11
      68 * HMASK: %11000000 IF POSITION = 12,13
      69 *
0348: 24 30   70           BIT  HMASK
034A: D0 06   71           BNE  NOPL0T     ; BIT 5 SET
034C: A9 81   72  PLT      LDA  #$81      ; %1000001
034E: 85 30   73           STA  HMASK
0350: D0 04   74           BNE  FIX      ; ALWAYS
0352: A9 80   75  NOPL0T  LDA  #$80      ; %1000000
0354: 85 30   76           STA  HMASK
      77 *
0356: C8      78  FIX      INY
0357: 20 5A F4 79           JSR  HPL0T+3
      80 *
035A: 60      81  DONE    RTS
035B: 88      82           CHK

```

This routine starts out much like the other PLOT.560 routine. Lines 24 through 44 are identical and perform the same function of calculating which X value to hand to the normal Applesoft routine. The first difference appears at line 50, where a JSR HPOSN is performed instead of a JSR HPL0T. This is done to allow Applesoft to go through its usual preparation for a plot. This sets up the color mask and position mask bytes and also the base address calculation.

At this point, we step into the usual process to tamper with the HMASK (\$30) value. As in the earlier example, this ordinarily would have just a single bit position “marked” for the upcoming PLOT. However, by using the ASL, ORA HMASK combination on lines 47 and 48, we can shift the original pattern and then superimpose the new pattern on the old.

Example: For X = 3

```

Original HMASK:  %1000 1000
ASL:              %0001 0000
ORA HMASK:       %1001 1000

```

The address usually given for the H PLOT routine, \$F457, includes a JSR to H POSN. Because we've already done this, a JSR H PLOT+3 accomplishes the first stage of our operation; namely, the plotting of a pure white dot.

Now the remaining problem is to take care of end-of-the-byte flaws. This can occur for four possible plots. For each byte, there are fourteen possible positions which can be plotted, numbered 0 through 13 (see Figure 20-4 in chapter 20). For positions 10 and 11, bits 5 and 6 will be set. Because bit 6 can be affected by bit 7 of the next byte in memory, a check for bit agreement must be made.

Stranger still, if positions 12 and 13 are plotted, only bit 6 is available, which would normally put us back to having plotted only a colored dot. To fix this, we have to go again to the next byte in memory and do another plot to illuminate just the very first dot of that byte.

In general then, the process will be:

1. Fix H MASK to turn on two adjacent bits where possible.
2. H PLOT with altered H MASK.
3. Check for bit 6 usage. If none, exit routine.
4. Set bit 7 of the next byte to agree with bit 7 of the current byte. Check whether bit 5 is being used. If not, go directly to H PLOT+3.
5. If bit 5 is set, set H MASK to plot only the first dot of the next byte.
6. Make a second pass to H PLOT to plot the X + 1 screen coordinate, single dot only. If H MASK set to # \$81, only the high-order bit will be set, with no actual plot done.

If you now examine line 52 of the listing, you'll see the BIT command is again used, this time to check bit 8 of H MASK. The BIT command forces bit 6 of the Status Register (the overflow flag) to the same value as bit 6 of H MASK. Thus BVS (Branch oVerflow Set) and BVC (Branch oVerflow Clear) can be used to check for bit 6 set or clear, respectively. In our case, BVC will branch to the exit point, DONE, if bit 6 is clear.

If bit 6 is set, lines 57-63 set the high bit of the other color mask byte, H COLOR (\$1C), to agree with the previous plot. This color mask byte is used later by H PLOT. Because we'll be skipping the usual entry point (\$F457), we have to set this byte specifically.

Once the color byte is set, another check is done to see if bit 5 is set. This is done by again using the BIT command. Since only bits 6 and 7 can be checked via the Status Register, we must load the Accumulator with a numeric image of the bit we wish to test for. In this case, the value used is # \$20 (%00100000). After the BIT command, a BNE (Branch Not Equal) will be taken if bit 5 is set. Yes, it sounds backward, but then BIT is a rather strange command.

Given the appropriate result of the BIT test, HMASK is loaded with either #81 or #80 depending on whether we wanted an actual plot to take place.

At line 78 (FIX), we take advantage of the fact that the Y-Register is still set to the correct value to access the current memory byte. By doing the INY, we advance the pointer to the next byte so that the JSR H PLOT+3 will make the appropriate corrections to the next byte in memory.

A Final Demo Program

To finish things off, let's try one last Applesoft program to make use of the new routines. This is an extension of the first listing presented at the beginning of this chapter, and it will give you an opportunity to compare the relative screen appearances of different routines.

```

10 D$ = CHR$(4)
40 HOME: INPUT "BLACK1 OR BLACK2? (1 OR 2)"; I : I = I - 1
100 REM NORMAL TEST
110 HGR: HCOLOR = I*4: H PLOT 0,0: CALL 62454: HCOLOR = 7 - 4*I
120 S = 1: REM SCALE FACTOR
130 K = 20: REM OFFSET VALUE
140 GOSUB 900
200 REM PLOT.140 TEST
205 PRINT D$;"BLOAD AL22.PLOT140,A$300"
210 S = 0.5: K = 40
220 F = 1: REM FUNCTION FLAG
230 GOSUB 900
300 REM PLOT.560 TEST
305 PRINT D$;"BLOAD AL21.PLOT560,A$300"
310 S = 2: K = 60
320 GOSUB 900
400 REM PLOT.560+ TEST
405 PRINT D$;"BLOAD AL22.PLOT560,A$300"
410 K = 80
420 GOSUB 900
500 REM PLOT.560W TEST
505 PRINT D$;"BLOAD AL22.PLOT560W,A$300"
510 K = 100
520 GOSUB 900
600 END
900 REM PLOTTER
930 FOR I = 0 TO 100
940 X = (I + K)*S : Y = I
950 X = X / 2
960 IF F = 0 THEN H PLOT X,Y
965 IF F THEN CALL 768, X, Y
970 NEXT I
980 RETURN

```

You'll also notice that this has the scaling factors built into it to make each line slant at the same angle. The offset factor K is used to move each plot to the right a little for appearance's sake.

By adding line 955 like so:

```
955  X = X / 2
```

you can slant the lines even further to show off the maximum slant possible for the 560-point modes. You might also want to try this program with the 140- and 560-point routines from the previous chapter to see how they perform in place of the new ones.

Conclusion

These routines are best used in mathematical charts rather than in pure graphics such as pictures. The main argument against the 560-point mode is that you can't be assured that plotting one point will not affect nearby points. As we've demonstrated here, there apparently is no approach that can guarantee this will not happen. It would seem, then, a matter of your own preference as to which to use. Our hope is that these routines will widen your options for your own programming goals and that they've taught you a little along the way.

The usual approach in this book has been to simplify any idea when first presenting it. In the area of graphics, though, simplicity has not been easy. For the most part, hi-res graphics gives the impression of being only marginally logical. In any event, though, now you're probably starting to get a feel for how the contents of memory affect what is displayed on the screen. In the final analysis, the real challenge of hi-res graphics is manipulating the contents of memory to produce the visual effects you want.

Hi-Res Graphics SCRN Function

August 1982

In lo-res graphics, the SCRN(X,Y) function returns the value of the color of the screen at the X, Y coordinate specified. Unfortunately, no equivalent function exists for use with hi-res graphics in Applesoft BASIC.

In the last few chapters we've seen how to plot points in a variety of ways. Now, here is a routine for doing a hi-res equivalent of the SCRN(X,Y) function. One conceivable use for this might be in a game program in which it's important to know when one object is touching another. Using the SCRN routine given here, you can test to see whether any points have already been plotted at the coordinates a presumably moving object is about to use.

```

1 *****
2 * AL23-HI-RES SCRN FNCTN *
3 * 6/22/82 *
4 *****
5 *
6 *
7 * OBJ $300
8 * ORG $300
9 *
10 CHKCOM EQU $DEBE
11 FRMNUM EQU $DD67
12 GETADR EQU $E752
13 LINNUM EQU $50
14 COMBYTE EQU $E74C
15 PTRGET EQU $DFE3
16 CHKNUM EQU $DD6A
17 GIVAYF EQU $E2F2
18 MOVMF EQU $EB2B
19 *
20 X EQU $E0
21 Y EQU $E2
22 *
23 HPOSN EQU $F411
24 HNDX EQU $E5
25 HBIT EQU $30
26 GBAS EQU $26
27 *
0300: 20 BE DE 28 ENTRY JSR CHKCOM
0303: 20 67 DD 29 JSR FRMNUM
0306: 20 52 E7 30 JSR GETADR
31 *
0309: A5 50 32 SET LDA LINNUM

```

```

030B: 85 E0    33          STA  X
030D: A5 51    34          LDA  LINNUM+1
030F: 85 E1    35          STA  X+1
          36          *
0311: 20 4C E7  37  GETY     JSR  COMBYTE
0314: 86 E2    38          STX  Y
          39          *
0316: A5 50    40  CHKX     LDA  LINNUM
0318: 4A        41          LSR          ; PUT BIT 0 IN CARRY
0319: A9 01    42          LDA  #$01    ; SET BIT 0
031B: 85 50    43          STA  LINNUM  ; %0000 0001
031D: B0 02    44          BCS  CHKHI   ; X='ODD'
031F: 06 50    45          ASL  LINNUM  ; SHIFT LEFT ONE POSN
          46          ; %0000 0010
          47          *
0321: A6 E0    48  CHKHI   LDX  X
0323: A4 E1    49          LDY  X+1
0325: A5 E2    50          LDA  Y
0327: 20 11 F4  51          JSR  HPOSN
          52          *
032A: A4 E5    53          LDY  HNDX
032C: B1 26    54          LDA  (GBAS),Y
032E: 48        55          PHA          ; SAVE DATA
032F: 10 08    56          BPL  HICLR   ; BIT 7 CLR
0331: A5 50    57  HISET     LDA  LINNUM
0333: 09 04    58          ORA  #$04    ; SET BIT 2
0335: 85 50    59          STA  LINNUM  ; BIT 'ON'
0337: D0 06    60          BNE  CHKBIT  ; ALWAYS
0339: A5 50    61  HICLR     LDA  LINNUM
033B: 29 8B    62          AND  #$8B    ; CLR BIT 2
033D: 85 50    63          STA  LINNUM
          64          *
033F: 68        65  CHKBIT   PLA          ; RETRIEVE SCREEN BYTE
0340: 25 30    66          AND  HBIT    ; SELECT BITS OF INTEREST
0342: 29 7F    67          AND  #$7F    ; CLR BIT 7
0344: D0 06    68          BNE  SEND    ; BIT IS "ON"
          69          *
0346: A5 50    70  OFF      LDA  LINNUM
0348: 29 8C    71          AND  #$8C    ; CLR BITS 0,1
034A: 85 50    72          STA  LINNUM
          73          *
          74          *
034C: 20 BE DE  75  SEND     JSR  CHKCOM
034F: A4 50    76          LDY  LINNUM
0351: A9 00    77          LDA  #$00
0353: 20 F2 E2  78          JSR  GIVAYF
0356: 20 E3 DF  79          JSR  PTRGET
0359: 20 6A DD  80          JSR  CHKNUM
035C: AA        81          TAX
035D: 20 2B EB  82          JSR  MOVMF
          83          *
0360: 60        84  DONE     RTS
0361: 0C        85          CHK

```

An Overview

You'll remember that in the previous chapter we used the Applesoft HPLLOT routine to plot a point. The X and Y coordinates for the point were passed to the routine via normal Applesoft variables.

The final plot was accomplished by setting a particular bit within a byte of memory. The bit to be set is determined by creating a "mask" for the bit position within the byte.

Figure 20-1 (our old friend from chapter 20) was used as a guide to which bits are set for any given color and X coordinate.

For our hi-res SCRNM function we need to identify whether the bit corresponding to a given X, Y coordinate has been set, to take into account the high-order bit (bit 7) where necessary, and then return a value between 0 and 7 corresponding to the color of the dot. Before going any further, take a look at Figure 23-1, which shows the bit patterns for the color values that might be returned.

Color	Value	Binary
Black1	0	0000 0000
Green	1	0000 0001
Violet	2	0000 0010
White1	3	0000 0011
Black2	4	0000 0100
Orange	5	0000 0101
Blue	6	0000 0110
White2	7	0000 0111

Figure 23-1: Color Bit Patterns

What the SCRNM routine does is establish a temporary register in which the bit pattern for the color value to be returned to the user will be constructed. Notice that for any of the possible color values we need concern ourselves only with the last three bit positions. This greatly simplifies our task.

Note also that when a dot is "off" (either black1 or black2), the routine returns a number with bits 0 and 1 cleared. Bit 2 will still have to be specifically conditioned, however, since Black2 sets the high-order bit of a byte even though no dot is illuminated.

Because neither white is directly plotted, the routine will never return a value of 3 or 7. Remember that when white is specified, Applesoft normally plots only one color. Thus our SCRNM routine has no way of determining whether a given dot is a pure color or part of a larger dot pattern creating a white line or area.

To determine a dot's color from among the four remaining colors, we look at the X position of the dot. Since you can plot only even color values at even coordinates, and odd color values at odd coordinates, the two final bit positions of the color register value will be 01 or 10 depending on whether X is odd or even. The status of the third bit depends on whether the dot's high-order bit is set.

When all of these checks are collected into a routine, we have the following procedure.

1. Lines 28–38 retrieve the values of the X and Y coordinates from the Apple-soft call command. These are transferred to the hi-res registers (\$E0–\$E2).
2. The value for the X coordinate is returned in LINNUM (\$50, \$51) and, as such, can be checked for whether it is odd or even. To do this we need only check the low-order byte to see whether the last bit (bit 0) is set. The easiest way to do this is to use the LSR (Logical Shift Right) command on line 41 to shift the last bit into the carry flag, which will be tested almost immediately.

Let's talk a bit (pardon the pun) of programming style here. We could test for all six possible color conditions individually, but it turns out that it is easier to set up the final color value more subtly. We'll start by assuming that some color will be present. Line 42 puts a possible value (\$01) into LINNUM as a starting point. (Since we're done with LINNUM from lines 28–38 we can now use it as our working register for the color value. Also note that we no longer need to worry about LINNUM+1 since the color value will never exceed 255.)

Now we can do the carry test, BCS (Branch on Carry Set), to see whether the coordinate was odd or even. If the carry bit is set, X was odd and LINNUM already contains the bit pattern for all of the colors that could be plotted at an odd coordinate. If the carry is clear, line 45 will be executed and will shift the pattern to the left one position to correspond to the "even" colors.

3. Lines 48–51 do the JSR HPOSN which will calculate the address of the byte in memory that corresponds to the coordinates given. See the plot routines from previous chapters if you need refreshing on this. Lines 53–55 load the byte into the Accumulator and push it onto the stack to be retrieved later.

The test on line 56 checks for whether the high-order bit was set. A BPL (Branch PLus) is done if the bit was clear. If the bit was set, we need to set bit 2 of LINNUM (our color register). Note that bit 2 is clear for HCOLORS 0–3 and set for HCOLORS 4–7. Bit 2 is set using the ORA (logical OR with Accumulator). If the high-order bit was clear, the logical AND command is used to clear bit 2.

4. Final check. Now we need to see whether the dot was actually turned on. The memory byte is retrieved from the stack using the PLA (PuLL Accumulator) and masked with HBIT (\$30). HBIT is a mask created by the HPOSN routine to show which bit corresponds to the given X coordinate. By masking HBIT with the memory byte we can isolate the bit we're interested in. As a further step, the AND #\$7F clears the high-order bit (which we've already tested for anyway). As an example, suppose that the memory location had held the value \$9B and the value for X was 4:

Note the final result will only be nonzero if the dot is on.

5. If the dot is on, everything is already set up, and we can proceed to the final exit phase. If the dot is off, the AND #8C on line 71 will clear only bits 0 and 1. This allows us to determine the status of the high-order bit, even if a dot is not actually plotted at the position given.

6. SEND (lines 75–84) is identical to the REAL VARIABLE SEND routine given in chapter 17 and is used to send our resulting value back to Applesoft. The only thing different in this case is that the routine loads a 0 into the Accumulator instead of the high-order byte of LINNUM (LINNUM+1) since, as mentioned previously, the value for color will never exceed 255.

Sample Program

To test this routine, BLOAD it at \$300 and call it using the syntax:

```
CALL 768, X, Y, C
```

where X and Y are the screen coordinates to examine, and C is the variable into which the routine will return the resulting color value from LINNUM.

As an example of using the SCRN routine from BASIC, this program will return all the possible values for C and illustrate the dependence of those values on HCOLOR and the X position:

```
0 HOME: VTAB 22: X = 0: Y = 0
5 PRINT CHR$(4);"BLOAD AL23.HGRSCRN"
10 FOR I = 0 TO 7
20 HGR: HCOLOR = I
30 HPLLOT X, Y
40 CALL 768, X, Y, C
50 PRINT "X = ";X;" COLOR = ";I;" RESULT = ";C
60 NEXT I
70 X = X + 1: IF X = 1 THEN 10
80 TEXT: END
```

The program goes through two passes, the first plotting all eight colors at X = 0, and the second with all eight colors at X = 1. After doing the plot, the program calls the SCRN routine to verify that it reads the color we think we plotted. It will do so except in the following cases:

1. White will always read as either 1, 2, 5, or 6¹. This is because when white is specified, a single HPLLOT illuminates only one color dot.
2. An attempt to plot an “odd” HCOLOR (1 or 5) on an even X coordinate or an “even” HCOLOR (2 or 6) on an odd X coordinate returns 0 or 4 as the result because of the plotting restrictions described in several of the previous chapters.

¹ [CT] originally read 2, 3, 5, 6

Conclusion

The SCRN routine can be applied in a variety of ways. In general, you can use this routine whenever you want to examine the screen to determine what color has been drawn. Possible applications might include graphics printing routines and games in which it is necessary to determine the existence of lines that represent walls or obstacles.

If you wish to use the routine directly from assembly language without calling it from Applesoft, simply delete the entry routines and load LINNUM with the X coordinate and \$E2 with the Y coordinate.

The Collision Counter, DRAW, XDRAW

September 1982

In the previous chapter we looked at a routine to simulate the `SCRN(X,Y)` function of BASIC. The notion of inquiring about points on the screen is closely related to this chapter's topic, the collision counter.

The collision counter is a one-byte memory location on page zero of the Apple's memory. Its value is a function of the Applesoft hi-res graphics routines specifically related to shape tables. The purpose of the collision counter is to keep track of any "collisions" between a shape being drawn on the screen and any previously drawn screen images. The collision counter is located at `$EA` (decimal 234) and is affected only by the commands `DRAW` and `XDRAW`.

Some Experiments

To illustrate the behavior of the collision counter, we'll first need a shape table to experiment with. The one given here is probably the simplest one possible—a single dot.

To enter the shape into memory, go into the Apple's Monitor by typing in `CALL -151<RETURN>`, and then enter:

```
300: 01 00 04 00 04 00
E8: 00 03
```

This will place the table in memory at location `$300` and set the pointer at `$E8`, `$E9` to point to the table.

The first two bytes of the table (`$01 00`) indicate the number of shapes in the table, which in our case is just one. The next two bytes (`$04 00`) give the offset from the beginning of the table (`$300`) to the start of the actual shape data (`$304`). The next two bytes (`$04 00`) are the actual bytes of data for the shape itself. In this example the shape table is a single "move" of one position up the screen.

You may wish to review the information on shape tables in your *Applesoft II BASIC Programming Reference Manual* (1978), pages 92–96.

The first experiment is to verify that we have in fact installed a usable shape table. This is most easily tested by putting your Apple into Applesoft BASIC and typing in:

```
HGR: HCOLOR = 3: ROT = 0: SCALE = 1
```

The screen should clear. You can now type in:

```
DRAW 1 AT 100,100
```

A single dot should appear on the screen. You can change the scale to three by typing in:

```
SCALE = 3
```

Test this by typing in:

```
DRAW 1 AT 100,100
```

A vertical line of three pixels should appear. If all has gone well so far, you can now try a third experiment. The purpose of the experiment will be to see how the collision counter reacts with various combinations of drawing colors, background colors, shape-drawing commands, and the previous condition of the collision counter.

Clear the screen with HGR again and try this sequence of commands, noting for each one what the conditions of the screen and collision counter are before and after the command is executed. (Note that references to “color” in this chapter will be in terms of “black” and “white” as would be seen on a black-and-white monitor. If you have a color display, the dots will appear as single-color dots—as explained in previous chapters.)

```
HCOLOR = 3: SCALE = 1: POKE 234,0: DRAW 1 AT 100,100: PRINT PEEK(234)
(0 should be printed along with a white dot on the screen)
```

```
DRAW 1 AT 100,100: PRINT PEEK(234)
(1, with a white dot)
```

```
DRAW 1 AT 100,100: PRINT PEEK(234)
(1, with a white dot)
```

```
HCOLOR = 0: DRAW 1 AT 100,100: PRINT PEEK(234)
(0, the dot is erased)
```

```
DRAW 1 AT 100,100: PRINT PEEK(234)
(1, with no dot)
```

If you try all the various combinations, you should be able to replicate a chart something like the one on the next page.

HCOLOR	Command	Background	C = 0	C = 1	Result
White	DRAW	Black	0	0	White
White	DRAW	White	1	1	White
Black	DRAW	Black	1	1	Black
Black	DRAW	White	0	0	Black
White	XDRAW	Black	1	1	Black
White	XDRAW	White	0	0	Black
Black	XDRAW	Black	1	1	White
Black	XDRAW	White	0	0	Black

The first column shows the value of HCOLOR for the DRAW or XDRAW command. The second column shows which command we used. The third column shows which background color was present when the shape table was drawn.

The headings C = 0 and C = 1 refer to the status of the collision counter before the DRAW or XDRAW. The entries in each column show the value after the command is executed. The final column shows whether the resulting dot is white (“on”) or black (“off”).

The conclusions to be “drawn” from this chart are:

1. If a DRAW is done, the resulting dot will be consistent with the HCOLOR used. The collision counter will increment one unit for each dot on the screen that is already at the same “color” as the dot being drawn. That is, if white is your HCOLOR, the collision counter will count the number of white dots the shape hits. If your HCOLOR is black, the collision counter will return the number of black dots the shape draws over. This allows you to use a light background and dark shapes and still have everything work!
2. If XDRAW is used, the current HCOLOR has no effect. XDRAW always reverses the background dots. For a black background, XDRAW will increment the collision counter only for those dots turned “on.” If the background is white, the collision counter will be set to 0 only if all of the dots are turned “off.”
3. The previous state of the collision counter has no effect on the final value after the DRAW or XDRAW. This means that no preconditioning or initializing is necessary in a given routine.

DRAW versus XDRAW

Before proceeding further with the collision counter, it is important to take a moment to clarify the distinction between the two shape-table commands DRAW and XDRAW.

DRAW is very direct in that it basically does an H PLOT in whatever the current HCOLOR is, using the specified shape. As mentioned earlier, the collision counter simply adds up the total number of collisions with existing dots in the same “on”

or “off” state as the HCOLOR being used. Moving shapes with DRAW is done first by drawing the figure, and then either reversing the color by setting HCOLOR to black and then doing another DRAW, or using XDRAW to accomplish approximately the same thing.

XDRAW, on the other hand, uses the EOR (Exclusive OR) function to actually reverse the bits on the screen where the shape is to be drawn. What this means is that a fixed color as such is not used. Rather, each bit on the screen in the desired shape pattern is reversed from its current status. By following this with another XDRAW, the screen is restored and existing background figures are not erased.

Principles of Animation and Collision

Any hi-res game or simulation is basically just a simulation of reality in which a screen image successfully mimics the behavior of an object in the real world. The primary things to be simulated generally are motion and collisions. Both of these have been discussed in earlier chapters, particularly with regard to the idea of simulating convincing motions.

In our previous programs, the positions of an object was used to determine whether it was time to bounce the object off of a wall or some other object. In this sense, we can say that collisions were predicted rather than detected. The collision counter gives us a way of detecting collisions with objects on the screen whose current position may not be known. This takes on practical significance when you may not want to keep track of all the things flying about the screen, as is quite possible in many game scenarios.

Putting all of this together, we come up with the following general approaches:

1. DRAW a figure. Check the collision counter for nonzero values to detect a collision. DRAW with black, or XDRAW, to erase for the next movement. Background figures will be erased when using this technique.
2. XDRAW a figure. The value in the collision counter should equal the number of dots in the figure (that is, a constant value) if there is no collision with existing images. XDRAW again to erase. The value in the collision counter should return to 0 if no previous collision was made. This will leave background images intact, but figures drawn will have a “harlequin” appearance as they pass over background images. See the following demonstration program for an example of this.

The Scanner

The following two demonstration programs are called *The Scanner* because they are reminiscent of the classic radar screen sweep pattern.

The first program uses the XDRAW, XDRAW system of redrawing the image and thus, is nondestructive to other images on the screen.

```

1 *****
2 * AL24-SCANNER-XDRAW, XDRAW *
3 *****
4 *
5 * OBJ $7000
6 * ORG $7000
7 *
8 FLAG EQU $06
9 RT EQU $07
10 SCL EQU $08
11 *
12 *
13 PREAD EQU $FB1E
14 WAIT EQU $FCA8
15 HCOLOR EQU $F6F0
16 HGR EQU $F3E2
17 HPLOT EQU $F457
18 HPOSN EQU $F411
19 SPKR EQU $C030
20 *
21 ROT EQU $F9
22 SCALE EQU $E7
23 PTR EQU $E8
24 SHNUM EQU $F730
25 DRAW EQU $F605
26 XDRAW EQU $F661
27 CTR EQU $EA
28 *
7000: 4C 09 70 29 ENTRY JMP E2
30 *
7003: 01 00 04 31 TBL HEX 010004
7006: 00 04 00 32 HEX 000400
33 *
7009: A2 03 34 E2 LDX #$03 ; WHITE
700B: 20 F0 F6 35 JSR HCOLOR
700E: A2 00 36 LDX #$00
7010: 86 07 37 STX RT
7012: A2 03 38 LDX #$03
7014: 86 E8 39 STX PTR
7016: A2 70 40 LDX #$70
7018: 86 E9 41 STX PTR+1
42 *
701A: A9 01 43 SET LDA #$01
701C: 85 06 44 STA FLAG
45 *
701E: A2 8C 46 POSN LDX #$8C
7020: A0 00 47 LDY #$00 ; X = 140
7022: A9 50 48 LDA #$50 ; Y = 80
7024: 20 11 F4 49 JSR HPOSN ; SET CURSOR X,Y
50 *
7027: E6 07 51 CALC INC RT
7029: A2 00 52 LDX #$00 ; PDL 0
702B: 20 1E FB 53 JSR PREAD
702E: 98 54 TYA
702F: D0 01 55 BNE STORE
7031: C8 56 INY ; SCALE = 1

```



```

7032: 84 08    57  STORE   STY  SCL
      58      *
7034: A5 06    59  CHKFLG  LDA  FLAG
7036: F0 04    60      BEQ  ERASE
7038: C6 06    61      DEC  FLAG
703A: F0 14    62      BEQ  PLOT      ; ONLY ONCE
      63      *
703C: A2 01    64  ERASE   LDX  #$01
703E: 20 30 F7 65      JSR  SHNUM
7041: A5 F9    66      LDA  ROT
7043: 20 61 F6 67      JSR  XDRAW
      68      *
7046: A6 EA    69  SOUND   LDX  CTR
7048: F0 06    70      BEQ  PLOT
704A: AD 30 C0 71  CLK     LDA  SPKR
704D: CA      72      DEX
704E: D0 FA    73      BNE  CLK
      74      *
7050: A2 8C    75  PLOT   LDX  $$8C
7052: A0 00    76      LDY  $$00
7054: A9 50    77      LDA  $$50
7056: 20 11 F4 78      JSR  HPOSN
7059: A2 01    79      LDX  $$01
705B: 20 30 F7 80      JSR  SHNUM
705E: A5 08    81      LDA  SCL
7060: 85 E7    82      STA  SCALE
7062: A5 07    83      LDA  RT
7064: 85 F9    84      STA  ROT
7066: 20 61 F6 85      JSR  XDRAW
      86      *
7069: A2 01    87  DELAY  LDX  #$01      ; PDL 1
706B: 20 1E FB 88      JSR  PREAD
706E: 98      89      TYA
706F: 20 A8 FC 90      JSR  WAIT
      91      *
7072: 4C 1E 70 92  GOBACK  JMP  POSN
      93      *
7075: D3      94      CHK

```

After assembling the code at \$7000, enter the following from Applesoft:

```
HGR: HCOLOR = 3: HPLOT 100,0 TO 100,160
```

Preset paddle 0 to the minimum (0 = far left) and paddle 1 to the maximum (255 = far right).

Now activate the routine by entering:

```
CALL 28672
```

Experiment with different paddle values, slowly increasing the radius with paddle 0 until the scanner intersects the vertical line. At that point you should hear a number of clicks from the speaker as the lines cross each other.

Let's see how the program works. Line 29 starts the actual code by jumping over the data for the shape table. This is the same one-dot shape table you

entered earlier in this chapter. Lines 34 and 35 initialize the HCOLOR to 3 (white), although for this program that actually is not necessary. Lines 36 through 41 set our value for rotation to 0 (to be used later), and set the pointer \$E8, \$E9 to point at our table at \$7003.

Now here's the tricky part. In general we want to store two positions for the line we'll draw. The first is the old position (where it was last drawn) and the second is the new position where the new line will be drawn. You'll recall that we developed this technique in earlier chapters as a way of moving dots while minimizing the screen flicker.

For the simple dots, it didn't really matter if on the first pass through the program we erased a dot that wasn't really there. In this case, though, it does matter because using XDRAW will cause an image to appear if one wasn't already there to erase.

This is solved by using a one-pass flag that will tell the program to skip over the ERASE routine on the first time through. Lines 43 and 44 initialize this flag to 1.

Lines 46 through 49 use HPOSN to prepare for the later use of the shape tables. Line 51 increments the value for rotation on each pass through the loop. This causes the line to revolve. Wrap-around happens automatically when RT reaches 255, so no checking for ILLEGAL QUANTITY errors is required.

Lines 52 through 57 get the scale value from paddle 0, which corresponds to the eventual length of the plotted line. Note that a special check is done to avoid scale being set to #00, since Applesoft treats this the same way it treats 255. This makes the paddles a little more friendly to the user.

On the first pass through, FLAG will equal 1, so the test on line 60 will fail. It will then be decremented to 0 to clear the flag, and the forced branch to PLOT will be executed.

The routine for drawing the shape is very similar to routines in programs presented in earlier chapters. The main difference in this routine is our use of the routine XDRAW (\$F661), which is used the same way the DRAW routine was used before.

Once the PLOT section is completed, a wait is done at lines 87 through 90 by using the WAIT (\$FCA8) routine as a function of paddle 1.

Notice that on successive passes through the loop, FLAG will equal 0, and so ERASE will always erase the old position before PLOT creates the new one. RT (\$07) and SCL (\$08) are used to hold the new values for rotation and scale, respectively.

Because we are using the XDRAW, XDRAW method for the actual collision detection, we will use method 2, which says that the collision counter should return to 0 after the figure is erased. We use this fact to check on lines 69 and 70 for a zero-value collision counter. If the counter is not 0, the speaker is clicked that number of times before the program does the next plot.

In practice the speaker is a little undependable because the frequency of the clicks is so high. You may wish to experiment with different delays in the CLK loop, as is done in the sound routines. You may prefer the current method for this demo because of the intuitive nature of the clicks, but musical sounds can also provide some interesting insights into the process.

The usual HGR equivalent from this routine has purposely been left out to allow you to alter the screen with H PLOT and other Applesoft commands before running the scanner. Another interesting variation is to type in:

```
HGR: HCOLOR = 3: H PLOT 0,0: CALL 62454
```

The screen should clear to all white. Now activate the scanner by typing in:

```
CALL 28672
```

Now the clicking will depend more directly on the length of the line, although some interesting variation can be observed depending on the angle of the line as well. While you're reading along you might ponder why that would be, considering that the screen would seem to be clearly uniform in the number of dots the line is intersecting.

Once you've entertained yourself sufficiently with the first program, try this second variation, one that uses the DRAW, XDRAW method. Here the point of interest is that the scanning line erases anything it touches and so leaves a visible trail of where it has been when activated against a solid white background.

```

1 *****
2 * AL24-SCANNER-DRAW,XDRAW *
3 *****
4 *
5 * OBJ $7000
6 * ORG $7000
7 *
8 FLAG EQU $06
9 RT EQU $07
10 SCL EQU $08
11 PREAD EQU $FB1E
12 WAIT EQU $FCA8
13 HCOLOR EQU $F6F0
14 HGR EQU $F3E2
15 H PLOT EQU $F457
16 H POSN EQU $F411
17 SPKR EQU $C030
18 *
19 ROT EQU $F9
20 SCALE EQU $E7
21 PTR EQU $E8
22 SHNUM EQU $F730
23 DRAW EQU $F605
24 XDRAW EQU $F661
25 CTR EQU $EA
26 *

```

```

7000: 4C 09 70 27 ENTRY   JMP  E2
      28 *
7003: 01 00 04 29 TBL     HEX  010004
7006: 00 04 00 30         HEX  000400
      31 *
7009: A2 03 32 E2     LDX  #$03      ; WHITE
700B: 20 F0 F6 33         JSR  HCOLOR
700E: A2 00 34         LDX  #$00
7010: 86 07 35         STX  RT
7012: A2 03 36         LDX  #$03
7014: 86 E8 37         STX  PTR
7016: A2 70 38         LDX  #$70
7018: 86 E9 39         STX  PTR+1
      40 *
701A: A9 01 41 SET     LDA  #$01
701C: 85 06 42         STA  FLAG
      43 *
701E: A2 8C 44 POSN    LDX  #$8C
7020: A0 00 45         LDY  #$00      ; X = 140
7022: A9 50 46         LDA  #$50      ; Y = 80
7024: 20 11 F4 47         JSR  HPOSN    ; SET CURSOR X,Y
      48 *
7027: E6 07 49 CALC    INC  RT
7029: A2 00 50         LDX  #$00      ; PDL 0
702B: 20 1E FB 51         JSR  PREAD
702E: 98 52         TYA
702F: D0 01 53         BNE  STORE
7031: C8 54         INY      ; SCALE = 1
7032: 84 08 55 STORE   STY  SCL
      56 *
7034: A5 06 57 CHKFLG  LDA  FLAG
7036: F0 04 58         BEQ  ERASE
7038: C6 06 59         DEC  FLAG
703A: F0 0A 60         BEQ  PLOT      ; ONLY ONCE
      61 *
703C: A2 01 62 ERASE   LDX  #$01
703E: 20 30 F7 63         JSR  SHNUM
7041: A5 F9 64         LDA  ROT
7043: 20 61 F6 65         JSR  XDRAW
      66 *
7046: A2 8C 67 PLOT    LDX  #$8C
7048: A0 00 68         LDY  #$00
704A: A9 50 69         LDA  #$50
704C: 20 11 F4 70         JSR  HPOSN
704F: A2 01 71         LDX  #$01
7051: 20 30 F7 72         JSR  SHNUM
7054: A5 08 73         LDA  SCL
7056: 85 E7 74         STA  SCALE
7058: A5 07 75         LDA  RT
705A: 85 F9 76         STA  ROT
705C: 20 05 F6 77         JSR  DRAW
      78 *
705F: A6 EA 79 SOUND   LDX  CTR
7061: F0 06 80         BEQ  DELAY
7063: AD 30 C0 81 CLK    LDA  SPKR
7066: CA 82         DEX

```

```

7067: D0 FA      83          BNE  CLK
              84          *
7069: A2 01      85  DELAY  LDX  #01      ; PDL 1
706B: 20 1E FB   86          JSR  PREAD
706E: 98         87          TYA
706F: 20 A8 FC   88          JSR  WAIT
              89          *
7072: 4C 1E 70  90  GOBACK  JMP   POSN
              91          *
7075: A9         92          CHK

```

In this routine, the first variation is in the use of DRAW (versus XDRAW) on line 77. In addition, because we are now using the DRAW, XDRAW method, the collision counter detection now goes after the initial creation of the image as is done by PLOT. In terms of programming then, the changes are minor. It is interesting to note, though, how differently the screen behaves.

It is most instructive to start by typing in:

```
HGR: HCOLOR = 3: HPLOT 0,0: CALL 62454
```

The CALL 62454 is the routine that clears the hi-res screen to the last HCOLOR plotted, so we'll take advantage of it to fill the screen with dots for our DRAW, XDRAW scanner to detect. Make sure the paddles are set to 0 for paddle 0 and 255 for paddle 1. Then activate the routine by typing in:

```
CALL 28672
```

As you eventually sweep out all possible angles and radii, you'll notice that not all screen locations can be reached from a fixed point. This is because of a limited number of rotation positions (as opposed to a continuous 360-degree motion) and also because of the line nature of the screen display.

By looking carefully you can see that there are more point interceptions, and thus collisions and clicks, at the near-vertical, -horizontal, and 45-degree positions than at the angles in between. This tends to give a modulated sound to the clicks as the "beam" scans when running the first program against a white background.

The Possibilities

Once you understand the idea behind the collision counter, it can be very useful in both arcade game-type software and other simulations. You'll probably be able to imagine all sorts of novel ways of applying this technique in your own programs.

In the next chapter, we'll give non-graphics enthusiasts a break and look a little more into some areas of assembly-language programming that we haven't yet covered.

Explosions and Special Effects

October 1982

In the previous chapter we looked at the collision counter and at how it could be used in hi-res graphics programs in which collisions might have to be detected. This chapter we'll see some further uses of the collision counter, along with simple examples of how an explosion might be simulated. In a way, this chapter could be considered a brief introduction to some special effects.

Explosions, Rays, and Other Things That Go Bump in the Night

The basic principles behind writing simple tone routines in assembly language were presented in chapter eight. As you'll recall, sound of any kind is generated by accessing memory location \$C030. Each time this location is accessed by either a read or write operation (such as an LDA or STA command) the speaker clicks once. A tone or other noise is produced by doing a large number of very fast accesses. Consider, for example, this sample listing:

```

1 *****
2 *
3 * AL25-SIMPLE NOISE ROUTINE *
4 *
5 *****
6 *
7 * OBJ $300
8 * ORG $300
9 *
10 DRTN EQU $06
11 NUM EQU $07
12 SPKR EQU $C030
13 *
14 COMBYTE EQU $E74C
15 RND EQU $EFAE
16 FAC EQU $9D
17 *
0300: 20 4C E7 18 ENTRY JSR COMBYTE
0303: 86 06 19 STX DRTN ; SET LEN OF 'NOTES'
0305: 20 4C E7 20 JSR COMBYTE
0308: 86 07 21 STX NUM ; SET # OF 'NOTES'
22 *
030A: 20 AE EF 23 LOOP JSR RND ; CREATE A RND VALUE
030D: A6 06 24 LDX DRTN ; SET A COUNTER
030F: AD 30 C0 25 TICK LDA SPKR ; TOGGLE SPEAKER
0312: A4 9F 26 LDY FAC+2 ; PITCH=RANDOM VALUE

```

```

0314: 88      27  DELAY  DEY
0315: D0 FD   28      BNE  DELAY ; WAIT AWHILE
0317: CA      29  CYCLE  DEX
0318: D0 F5   30      BNE  TICK  ; KEEP PLAYING
          31  *
031A: C6 07   32  NUMBR  DEC  NUM
031C: D0 EC   33      BNE  LOOP  ; PLAY ANOTHER NOTE
031E: 60      34  EXIT  RTS
031F: 71      35      CHK

```

This routine is intended to be called from Applesoft BASIC by a program such as this one:

```

10 INPUT "DURATION,NUMBER: ";D,N
20 CALL 768,D,N
30 GOTO 10

```

When the routine is called, lines 18 through 21 use the routine COMBYTE (\$E74C) in Applesoft to read the values being passed by the calling program and store these values in DRTN (\$06) and NUM (\$07).

DRTN is then used to determine the length of a tone to be generated, and NUM determines how many tones will be played. You could think of this program as a random melody generator.

At line 23, a JSR is done to Applesoft's random-number function. This fills the floating-point Accumulator (usually called FAC: \$9D-\$A2) with a random number in floating-point form. For our purposes we need only a single byte, which we'll get from \$9F. Very shortly we'll retrieve this byte from FAC for use in our routine. You might think that any of the six bytes in the FAC would be sufficiently random, but it turns out that the first two bytes, FAC and FAC+1 (\$9D, \$9E), don't vary sufficiently to generate good random numbers.

Line 24 retrieves the value for DRTN to prepare for entering the main tone service loop. TICK clicks the speaker once and then loads the Y-Register with our random value. Because this value is then used in the DELAY loop, the interval between clicks varies each time a new random number is used. This is equivalent to a different frequency being produced each time, and thus gives us randomly-pitched notes.

CYCLE is a secondary loop that executes the TICK/DELAY loop a certain number of times, determined in this case by the value given to DRTN by the calling program. The number of CYCLEs determines the overall apparent length of a particular tone unit.

NUMBR is a larger loop that determines how many notes the sound routine will generate, according to the value given for NUM.

Run the Applesoft program with this routine assembled at \$300 and try different combinations for DRTN and NUM. If DRTN is a large value (greater than 20), a random melody of NUM notes is generated. As DRTN gets smaller, you have to increase NUM to get sounds that last equivalent lengths of time. The value pair 10,

50 for DRTN and NUM creates sort of a ray-gun sound, and the pair 3, 20 produces a reasonable explosion effect. In the latter case, the amount of time each note is played becomes so short that the notes tend to blend together into what's essentially just a random noise pattern.

A random tonal pattern is, in fact, the key to any definition of noise, and noise is what an explosion is all about. What we need is a way of generating a lot of high-speed random data for a good noise routine. The RND function helps us to create the random data, but it takes so long to execute the routine for each note that there is a limit to the number of notes we can generate in a short period of time.

One technique we used earlier when speed was a problem was table look-ups. Let's apply this technique to sound generation and see what we can produce.

```

1 *****
2 *
3 * AL25-SIMPLE NOISE ROUTINE 2 *
4 *
5 *****
6 *
7 *      OBJ   $300
8 *      ORG   $300
9 *
10 CTR      EQU  $06
11 DRTN     EQU  $07
12 PTCH     EQU  $08
13 SPKR     EQU  $C030
14 *
15 COMBYTE  EQU  $E74C
16 RND      EQU  $EFAE
17 FAC      EQU  $9D
18 *
0300: A9 00 19 INIT      LDA  #$00
0302: 85 06 20          STA  CTR
0304: 20 AE EF 21 LOOP    JSR  RND
0307: A5 9F 22          LDA  FAC+2
0309: A4 06 23          LDY  CTR
030B: 99 00 10 24         STA  $1000,Y
030E: E6 06 25          INC  CTR
0310: D0 F2 26          BNE  LOOP
0312: 60 27 DONE      RTS
28 *
0313: 20 4C E7 29 ENTRY   JSR  COMBYTE
0316: 86 08 30          STX  PTCH
0318: 20 4C E7 31          JSR  COMBYTE
031B: 86 07 32          STX  DRTN
33 *
031D: A0 00 34 READ     LDY  #$00
031F: B9 00 10 35 BYTE    LDA  $1000,Y
0322: A2 08 36          LDX  #$08
0324: 4A 37 SHIFT    LSR
0325: 90 03 38          BCC  NEXTBIT

```



```

0327: 8D 30 C0 39 TICK STA SPKR
          40 *
032A: CA 41 NEXTBIT DEX
032B: D0 F7 42 BNE SHIFT
032D: A6 08 43 LDX PTCH
032F: CA 44 DELAY DEX
0330: D0 FD 45 BNE DELAY
0332: C8 46 NEXTBYTE INY
0333: D0 EA 47 BNE BYTE
0335: C6 07 48 DEC DRTN
0337: D0 E4 49 BNE READ
0339: 60 50 EXIT RTS
033A: 33 51 CHK

```

This routine has two *entry points*. This means that the routine has to be called twice. The first time, a call to \$300 (768 decimal) generates the table of data to be used. This need be done only once. The noise pattern is generated by calling \$313 (787 decimal) whenever a sound is desired. This routine is also designed to be called from an Applesoft BASIC program such as:¹

```

10 CALL 768: REM CREATE TABLE
20 INPUT "PITCH,DURATION";P,D
30 CALL 787,P,D : REM CALL NOISE ROUTINE
40 GOTO 20

```

In this case the two parameters passed to the noise routine are pitch (PTCH) and duration of the noise period (DRTN). At first thought, pitch may seem to be a contradictory notion when applied to noise, particularly in light of our previous definition of noise as a random mix of frequencies. The pitch, however, does not need to be an entirely homogeneous mixture of frequencies.

It's possible to favor either high or low frequencies in the mix and thus to influence the suggestive nature of the noise. High-frequency mixes sound like rays or fast-moving rockets. Low-frequency mixes remind the listener of the low roar of a slow-moving rocket or a garden-variety explosion.

Examining the new routine, then, let's see how this noise generator works. The first call to INIT creates the table of random values. Lines 19 and 20 initialize to \$00 a counter we'll be using shortly. A call to the random function is then made to generate a random byte. Next, the Y-Register is loaded with the value held in CTR. This value is used as an index to the location in the range from \$1000 through \$10FF where we will store the random byte. CTR is then incremented to the next position and LOOP is executed until CTR wraps back around to \$00 after cycling 256 times.

You may wonder why this code was not used instead:

```

          LDY #$$$0
LOOP     JSR RND
          LDA FAC+2

```

¹ [CT] Line 20 incorrectly had INPUT "D,P"; P,D

```

STA $1000,Y
INY
BNE LOOP

```

Although it's much shorter and more direct, the routine fails because RND scrambles the Y-Register, thus losing any running value for our position in the table being created. This fact necessitates the use of a back-up counter (CTR) to remember the current value that Y should be set to.

The INIT routine, then, will fill 256 bytes of memory starting at \$1000 with a random pattern of bytes. More important, this also results in a random pattern of bits, which will be used very soon by the noise routine.

When \$313 (787 decimal) is called, COMBYTE is used to read the values for PTCH and DRTN from the calling Applesoft program.

READ then starts the process of scanning the data table for the random data to be used in generating the noise pattern. The trick in this program comes in using the actual bit status of the data rather than entire bytes.

After each byte is loaded into the Accumulator on line 35, a bit-shifting routine is executed eight times to determine the on or off status of each bit. Line 36 initializes the X-Register to act as our counter in this eight-step loop.

Line 37 uses the LSR command (Logical Shift Right) to move all of the bits in the Accumulator one position to the right. The end-position bit, bit 0, falls into the carry.

Line 38 then tests the carry flag and, if the flag is clear (bit not set), skips the speaker-toggling step found at line 39.

NEXTBIT decrements our counter in the X-Register, and if X hasn't reached 0, loops back to SHIFT. If X has reached 0, X is reset with the PTCH value and a delay loop is entered.

When the delay loop is finished, the Y-Register is incremented in preparation for reading the next byte in the data table.

As it happens, reading each bit of 256 bytes does not take that long. Our sound would be over rather soon if we didn't do just one extra step. Although we could generate and read larger tables, another approach is to reread the table a set number of times. This is where the DRTN value is used, and the table read is repeated the number of times specified by DRTN.

The main area of experimentation in this routine is with different values for PTCH. Smaller values produce higher-sounding noise patterns; larger values generate more of a roar.

A Little More Sophistication

This last routine probably sounds more like an explosion to you than the first one did. This is due to the higher noise content of the sound as compared to the more musical first routine. Something is still missing, though. A classical

explosion doesn't sound the same from start to finish. It usually starts at a higher or lower pitch and works its way up or down, depending on the nature of the explosion. What we need is a way to modulate the frequency mix as a function of time.

By linking the delay value to our position in the table, we can accomplish this goal. Here's the new listing:

```

1 *****
2 * *
3 *AL25-SIMPLE RAMP NOISE ROUTINE*
4 * *
5 *****
6 *
7 * OBJ $300
8 * ORG $300
9 *
10 CTR EQU $06
11 DRTN EQU $07
12 PTCH EQU $08
13 SPKR EQU $C030
14 *
15 COMBYTE EQU $E74C
16 RND EQU $EFAE
17 FAC EQU $9D
18 *
0300: A9 00 19 INIT LDA #$00
0302: 85 06 20 STA CTR
0304: 20 AE EF 21 LOOP JSR RND
0307: A5 9F 22 LDA FAC+2
0309: A4 06 23 LDY CTR
030B: 99 00 10 24 STA $1000,Y
030E: E6 06 25 INC CTR
0310: D0 F2 26 BNE LOOP
0312: 60 27 DONE RTS
28 *
0313: 20 4C E7 29 ENTRY JSR COMBYTE
0316: 86 07 30 STX DRTN
0318: A0 00 31 READ LDY #$00
031A: B9 00 10 32 BYTE LDA $1000,Y
031D: A2 08 33 LDX #$08
031F: 4A 34 SHIFT LSR
0320: 90 03 35 BCC NEXTBIT
0322: 8D 30 C0 36 TICK STA SPKR
37 *
0325: CA 38 NEXTBIT DEX
0326: D0 F7 39 BNE SHIFT
0328: A6 07 40 LDX DRTN
032A: CA 41 DELAY DEX
032B: D0 FD 42 BNE DELAY
032D: C8 43 NEXTBYTE INY
032E: D0 EA 44 BNE BYTE
0330: C6 07 45 DEC DRTN
0332: D0 E4 46 BNE READ
0334: 60 47 EXIT RTS

```

0335: 39 48 CHK

This program is designed to be called from an Applesoft program that looks like this:

```
10 CALL 768: REM GENERATE TABLE
20 INPUT "START?";S
30 CALL 787, S
40 GOTO 20
```

The main difference between this routine and the previous one is that just prior to the delay loop, the X-Register is loaded with the current DRTN counter value, as opposed to a user-defined pitch value. Thus, no PTCH is specified in the calling program and you may select only a starting point on the ramp, as it is sometimes called.

Entering a value of 255 results in the longest sound possible. It is rather interesting to have your Apple sound like a 727 ready to take off through your ceiling.

Putting it All Together

Now that we've got some sound effects to add to our knowledge of hi-res graphics, let's put everything together into a simple demonstration of how an explosion might be simulated in a game program.

Assemble the following listing and run it either with BRUN or CALL 4096 (from BASIC), or 1000G (from the Monitor).

```
1 *****
2 *
3 *AL25-SIMPLE EXPLOSION ROUTINE *
4 *
5 *****
6 *
7                    ORG    $1000
8 *
9 NUM            EQU    $06
10 SPKR           EQU    $C030
11 *
12 RND            EQU    $EFAE
13 FAC            EQU    $9D
14 KYBD           EQU    $C000
15 STROBE        EQU    $C010
16 *
17 HGR            EQU    $F3E2
18 HCOLOR        EQU    $F6F0
19 SHNUM         EQU    $F730
20 XDRAW         EQU    $F661
21 HPOSN         EQU    $F411
22 SHTBL         EQU    $E8
23 SCALE         EQU    $E7
24 *
```

```

1000: 4C 63 10 25 ENTRY JMP START
      26 *
1003: 03 00 55 27 TABLE HEX 0300550033000800
1006: 00 33 00 08 00
100B: 2C 24 2D 28 HEX 2C242D242DE4DB93
100E: 24 2D E4 DB 93
1013: 3E 36 37 29 HEX 3E36372E362D3635
1016: 2E 36 2D 36 35
101B: 36 2D C6 30 HEX 362DC6DBDB23272C
101E: DB DB 23 27 2C
1023: 25 2C 3C 31 HEX 252C3C3F363F373E
1026: 3F 36 3F 37 3E
102B: 36 40 C0 32 HEX 3640C040C028352E
102E: 40 C0 28 35 2E
1033: 35 2D 00 33 HEX 352D00243F3CBC12
1036: 24 3F 3C BC 12
103B: 0E 96 09 34 HEX 0E9609C04C493C2C
103E: C0 4C 49 3C 2C
1043: 2C 2D 24 35 HEX 2C2D2494921A352D
1046: 94 92 1A 35 2D
104B: 36 EE DB 36 HEX 36EEDB233C27941B
104E: 23 3C 27 94 1B
1053: 3E 36 3F 37 HEX 3E363F06001B282D
1056: 06 00 1B 28 2D
105B: 2D F8 DB 38 HEX 2DF8DB636DE52300
105E: 63 6D E5 23 00
      39 *
1063: 20 E2 F3 40 START JSR HGR
1066: A2 03 41 LDX #03 ; WHITE
1068: 20 F0 F6 42 JSR HCOLOR
106B: A9 03 43 LDA #03
106D: 85 E8 44 STA SHTBL
106F: A9 10 45 LDA #10
1071: 85 E9 46 STA SHTBL+1 ; TABLE AT $1003
1073: A9 01 47 LDA #01
1075: 85 E7 48 STA SCALE ; SCALE = 1
1077: A9 0A 49 LDA #0A
1079: 85 06 50 STA NUM ; # OF CYCLES
      51 *
107B: A2 8C 52 SHIP LDX #8C
107D: A0 00 53 LDY #00 ; X = 140
107F: A9 50 54 LDA #50 ; Y = 80
1081: 20 11 F4 55 JSR HPOSN ; POSITION 'CURSOR'
1084: A2 01 56 LDX #01 ; #1 = SHIP
1086: 20 30 F7 57 JSR SHNUM
1089: A9 00 58 LDA #00 ; ROT = 0
108B: 20 61 F6 59 JSR XDRAW
      60 *
108E: AD 00 C0 61 KEY? LDA KYBD
1091: 10 FB 62 BPL KEY? ; NO KEYPRESS
1093: 8D 10 C0 63 STA STROBE ; CLEAR STROBE
      64 *
1096: A2 8C 65 ERASE1 LDX #8C
1098: A0 00 66 LDY #00
109A: A9 50 67 LDA #50
109C: 20 11 F4 68 JSR HPOSN

```

```

109F: A2 01 69          LDX  #01
10A1: 20 30 F7 70      JSR  SHNUM
10A4: A9 00 71          LDA  #00
10A6: 20 61 F6 72      JSR  XDRAW          ; ERASE SHIP
      73 *
10A9: A2 8C 74  LOOP   LDX  #8C
10AB: A0 00 75          LDY  #00
10AD: A9 50 76          LDA  #50
10AF: 20 11 F4 77      JSR  HPOSN
10B2: A2 02 78          LDX  #02          ; 1ST EXPL SHAPE
10B4: A5 06 79          LDA  NUM
10B6: 6A 80             ROR
10B7: B0 01 81          BCS  SET          ; IF 'ODD'
10B9: E8 82             INX          ; 2ND EXPL SHAPE
10BA: 20 30 F7 83      SET   JSR  SHNUM
10BD: A9 00 84          LDA  #00
10BF: 20 61 F6 85      JSR  XDRAW          ; DRAW EXPLOSION
      86 *
10C2: 20 AE EF 87      GETPTCH JSR  RND
10C5: A2 10 88          LDX  #10
10C7: AD 30 C0 89      TICK  LDA  SPKR          ; CLICK SPEAKER
10CA: A4 9F 90          LDY  FAC+2        ; PITCH = RND
10CC: 88 91            DELAY  DEY
10CD: D0 FD 92          BNE  DELAY
10CF: CA 93            CYCLE   DEX
10D0: D0 F5 94          BNE  TICK
      95 *
10D2: A2 8C 96  ERASE2  LDX  #8C
10D4: A0 00 97          LDY  #00
10D6: A9 50 98          LDA  #50
10D8: 20 11 F4 99      JSR  HPOSN
10DB: A2 02 100        LDX  #02
10DD: A5 06 101        LDA  NUM
10DF: 6A 102           ROR
10E0: B0 01 103        BCS  SET2         ; IF 'ODD'
10E2: E8 104           INX          ; 2ND EXPLOSION FIG.
10E3: 20 30 F7 105     SET2   JSR  SHNUM
10E6: A9 00 106        LDA  #00
10E8: 20 61 F6 107     JSR  XDRAW          ; ERASE FIGURE
      108 *
10EB: C6 06 109  DRTN   DEC  NUM
10ED: D0 BA 110        BNE  LOOP
10EF: 60 111  EXIT    RTS
10F0: 28 112          CHK

```

When the program is run, the hi-res screen should clear and a flying-saucer-like ship should appear in the middle of the screen. Pressing any key will blow up the spaceship. Let's see how this is done.

Lines 27 through 38 contain the data for a three-element shape table. This table is jumped over when the program is first run. START clears the hi-res screen in the usual manner and initializes the shape-table pointers and the HCOLOR and SCALE values. Lines 49 and 50 set NUM to 10, to be used later as the number of cycles the explosion routine will go through.

SHIP draws the spaceship in the center of the screen. KEY? waits for a key-press. When a key is pressed, the code moves on to ERASE, which erases the ship prior to starting the explosion sequence.

The explosion sequence itself consists of a three-part loop. These parts consist of: (1) drawing one of two explosion shapes, (2) creating a little noise with the speaker, and (3) erasing the explosion shape drawn in step 1.

This sequence is then repeated a number of times depending on how long you want the explosion to last. In detail, here's how this sequence is carried out.

Lines 74–77 position the hi-res cursor at the ship's old position. Lines 78–82 then select one of the two explosion shapes included in the table based on whether NUM (the current loop counter) is odd or even.

This is done by first loading the X-Register with what might be called a default value of \$02 for the first explosion shape (which is the second item in the table). NUM is then loaded into the Accumulator and a ROR (ROtate Right) command is done to shift all of the bits to the right one position. Bit 0 will then be forced into the carry, where we can test with the BCS (Branch Carry Set) command. (This is similar to the technique used earlier for the noise routine. In fact, the LSR command would have worked just as well here, but a little variety can sometimes be nice.)

If the carry was set, then NUM was odd and we'll go right to the next phase. If the carry was clear, then NUM was even and the INX (INcrement X) will be executed. Remember that the X-Register is always loaded with the shape number you want to DRAW or XDRAW prior to calling SHNUM. If the INX is done, X goes from \$02 to \$03, thus indicating shape number 3, which corresponds to the second explosion shape in the table.

Once an explosion shape has been drawn, the first noise routine presented earlier is used to generate a short burst of quick random notes. This passes for some background noise for an explosion. After a few quick sounds, ERASE2 again XDRAWS the shape selected in LOOP. This has the effect of erasing the previous image. Finally, lines 109 and 110 check NUM to see if the loop is finished. As written, line 49 sets the loop counter to ten passes, but you may want to try different values to suit your own tastes.

Because all imaging is done with XDRAW, the HCOLOR setting actually is irrelevant; this routine would work on any screen background. You may want to try clearing the screen to different backgrounds as described in the previous chapter and see how the routine given here behaves.

The Shooter Program

What we need now is some sort of collective example of how all of this can be put together as it might be done in an actual game. Although it's not necessarily your definitive hi-res arcade game, the following is offered for your general interest and amusement.²

```

1 *****
2 *      AL25-SHOOTER PROGRAM      *
3 *****
4 *
5 *
6          ORG  $1000
7 *
8 FLAG    EQU  $E3
9 X       EQU  $E0
10 Y      EQU  $E2
11 X0     EQU  $06
12 Y0     EQU  $08
13 NUM    EQU  $0C
14 *
15 PREAD  EQU  $FB1E
16 WAIT   EQU  $FCA8
17 PB0    EQU  $C061
18 HCOLOR EQU  $F6F0
19 HGR    EQU  $F3E2
20 HPLOT  EQU  $F457
21 HPOSN  EQU  $F411
22 HLIN   EQU  $F53A
23 ROT    EQU  $F9
24 SCALE  EQU  $E7
25 SHNUM  EQU  $F730
26 DRAW   EQU  $F605
27 XDRAW  EQU  $F661
28 HFIND  EQU  $F5CB
29 CTR    EQU  $EA
30 PTR    EQU  $E8
31 SPKR   EQU  $C030
32 RND    EQU  $EFAE
33 FAC    EQU  $9D
34 *
1000: 4C 67 10 35 ENTRY    JMP  E2
36 *
1003: 04 00 59 37          HEX  0400590037000C00
1006: 00 37 00 0C 00
100B: 0A 00 04 38          HEX  0A0004002C242D24
100E: 00 2C 24 2D 24
1013: 2D E4 DB 39          HEX  2DE4DB933E36372E
1016: 93 3E 36 37 2E
101B: 36 2D 36 40          HEX  362D3635362DC6DB
101E: 35 36 2D C6 DB
1023: DB 23 27 41          HEX  DB23272C252C3C3F

```

² [CT] Lines 94–98 were changed to divide the paddle value by 4 to convert the rotation into the allowed range of 0–63.


```

1026: 2C 25 2C 3C 3F
102B: 36 3F 37 42      HEX  363F373E3640C040
102E: 3E 36 40 C0 40
1033: C0 28 35 43      HEX  C028352E352D0024
1036: 2E 35 2D 00 24
103B: 3F 3C BC 44      HEX  3F3CBC120E9609C0
103E: 12 0E 96 09 C0
1043: 4C 49 3C 45      HEX  4C493C2C2C2D2494
1046: 2C 2C 2D 24 94
104B: 92 1A 35 46      HEX  921A352D36EEDB23
104E: 2D 36 EE DB 23
1053: 3C 27 94 47      HEX  3C27941B3E363F06
1056: 1B 3E 36 3F 06
105B: 00 1B 28 48      HEX  001B282D2DF8DB63
105E: 2D 2D F8 DB 63
1063: 6D E5 23 49      HEX  6DE52300
1066: 00

      50 *
1067: 20 E2 F3 51 E2   JSR  HGR          ; CLR SCRN
106A: A2 03 52         LDX  #$03
106C: 20 F0 F6 53     JSR  HCOLOR
      54 *
106F: A2 00 55     WALL  LDX  #$00
1071: A0 00 56         LDY  #$00          ; X = 0
1073: A9 05 57         LDA  #$05          ; Y = 5
1075: 20 57 F4 58     JSR  HPLOT        ; PLOT 0,5
1078: A9 17 59         LDA  #23          ; 279 MOD 256
107A: A2 01 60         LDX  #01          ; 279/256
107C: A0 05 61         LDY  #$05          ; Y = 5
107E: 20 3A F5 62     JSR  HLIN         ; 0,5 TO 279,5
      63 *
1081: A9 17 64         LDA  #$17
1083: A2 01 65     LDX  LDX  #$01          ; X = 279
1085: A0 06 66         LDY  #$06          ; Y = 6
1087: 20 3A F5 67     JSR  HLIN         ; 279,5 TO 279,6
      68 *
108A: A9 00 69         LDA  #$00
108C: A2 00 70         LDX  #$00          ; X = 0
108E: A0 06 71         LDY  #$06          ; Y = 6
1090: 20 3A F5 72     JSR  HLIN         ; 279,6 TO 0,6
      73 *
1093: A9 03 74     SET  LDA  #$03
1095: 85 E8 75         STA  PTR
1097: A9 10 76         LDA  #$10
1099: 85 E9 77         STA  PTR+1        ; SET TBL = $1003
109B: A9 01 78         LDA  #$01
109D: 85 E7 79         STA  SCALE
109F: 85 E3 80         STA  FLAG
10A1: A9 0A 81         LDA  #$0A
10A3: 85 0C 82         STA  NUM          ; # OF EXPLOSIONS
10A5: A2 8C 83     SHIP  LDX  #$8C
10A7: A0 00 84         LDY  #$00
10A9: A9 50 85         LDA  #$50
10AB: 20 11 F4 86     JSR  HPOSN
10AE: A2 01 87         LDX  #$01          ; #1 = SHIP
10B0: 20 30 F7 88     JSR  SHNUM

```

```

10B3: A9 00 89          LDA  #000          ; ROT = 0
10B5: 20 05 F6 90      JSR  DRAW
      91 *
10B8: A2 00 92  CALC   LDX  #000
10BA: 20 1E FB 93      JSR  PREAD
10BD: 84 F9 94          STY  ROT          ; [CT] NEED TO SCALE TO 0-63
10BF: 46 F9 95          LSR  ROT          ; [CT] DIVIDE BY 2
10C1: 46 F9 96          LSR  ROT          ; [CT] DIVIDE BY 2
10C3: A9 48 97  PAUSE  LDA  #48
10C5: 20 A8 FC 98      JSR  WAIT
10C8: AD 61 C0 99      SHOOT? LDA  PB0
10CB: 30 03 100        BMI  YES
10CD: 4C B8 10 101     JMP  CALC          ; (NOPE)
      102 *
10D0: A2 8C 103  YES   LDX  #8C
10D2: A0 00 104        LDY  #000
10D4: A9 4E 105        LDA  #4E          ; Y = INSIDE SHIP
10D6: 20 11 F4 106     JSR  HPOSN
10D9: A2 04 107        LDX  #04          ; #4 = SINGLE DOT
10DB: 20 30 F7 108     JSR  SHNUM
10DE: A5 F9 109        LDA  ROT
10E0: 20 61 F6 110     JSR  XDRAW        ; DRAW RAY
      111 *
10E3: A2 05 112  SOUND  LDX  #05          ; # OF CYCLES
10E5: AD 30 C0 113  TICK  LDA  SPKR
10E8: A4 E7 114        LDY  SCALE
10EA: 88 115  DELAY  DEY
10EB: D0 FD 116        BNE  DELAY
10ED: CA 117  CYCLE  DEX
10EE: D0 F5 118        BNE  TICK
      119 *
10F0: A2 8C 120  ERASE1  LDX  #8C
10F2: A0 00 121        LDY  #000
10F4: A9 4E 122        LDA  #4E
10F6: 20 11 F4 123     JSR  HPOSN
10F9: A2 04 124        LDX  #04
10FB: 20 30 F7 125     JSR  SHNUM
10FE: A5 F9 126        LDA  ROT
1100: 20 61 F6 127     JSR  XDRAW        ; ERASE RAY
1103: A5 EA 128        LDA  CTR
1105: C9 02 129        CMP  #02
1107: B0 0F 130        BCS  HIT
      131 *
1109: E6 E7 132  NEXT   INC  SCALE
110B: E6 E7 133        INC  SCALE
110D: E6 E7 134        INC  SCALE
110F: A5 E7 135        LDA  SCALE
1111: C9 90 136        CMP  #90
1113: 90 BB 137        BCC  YES
1115: 4C 7B 11 138     JMP  MISS
      139 *
1118: 20 CB F5 140  HIT   JSR  HFIND        ; GET CURSOR POSN
111B: A5 E0 141        LDA  X
111D: 85 06 142        STA  X0
111F: A5 E1 143        LDA  X+1
1121: 85 07 144        STA  X0+1

```

```

1123: A5 E2      145          LDA  Y
1125: 85 08      146          STA  Y0          ; SAVE CURSOR POSN
1127: A9 01      147          LDA  $$01
1129: 85 E7      148          STA  SCALE      ; RESET SCALE
                        149  *
112B: A2 03      150  EXPLOS  LDX  $$03      ; WHITE
112D: 20 F0 F6  151          JSR  HCOLOR
1130: A6 06      152          LDX  X0
1132: A4 07      153          LDY  X0+1
1134: A5 08      154          LDA  Y0
1136: 20 11 F4  155          JSR  HPOSN
1139: A2 02      156          LDX  $$02      ; 1ST EXPLOSION
113B: A5 0C      157          LDA  NUM
113D: 6A         158          ROR
113E: B0 01      159          BCS  BOOM      ; IF 'ODD'
1140: E8         160          INX
1141: 20 30 F7  161  BOOM    JSR  SHNUM
1144: A9 00      162          LDA  $$00
1146: 20 05 F6  163          JSR  DRAW      ; DRAW 1ST EXPLOSION
                        164  *
1149: 20 AE EF  165  GETPTCH JSR  RND
114C: A2 10      166          LDX  $$10      ; # OF CYCLES
114E: AD 30 C0  167  TICK2   LDA  SPKR
1151: A4 9F      168          LDY  FAC+2     ; PITCH = RND
1153: 88         169  DELAY2  DEY
1154: D0 FD      170          BNE  DELAY2
1156: CA         171  CYCLE2  DEX
1157: D0 F5      172          BNE  TICK2
                        173  *
1159: A2 00      174  ERASE2  LDX  $$00      ; BLACK
115B: 20 F0 F6  175          JSR  HCOLOR
115E: A6 06      176          LDX  X0
1160: A4 07      177          LDY  X0+1
1162: A5 E2      178          LDA  Y
1164: 20 11 F4  179          JSR  HPOSN
1167: A2 02      180          LDX  $$02
1169: A5 0C      181          LDA  NUM
116B: 6A         182          ROR
116C: B0 01      183          BCS  BOOM2   ; IF 'ODD'
116E: E8         184          INX
116F: 20 30 F7  185  BOOM2  JSR  SHNUM
1172: A9 00      186          LDA  $$00      ; ROT = 0
1174: 20 05 F6  187          JSR  DRAW      ; ERASE FIGURE
1177: C6 0C      188  DRTN    DEC  NUM
1179: D0 B0      189          BNE  EXPLOS
                        190  *
117B: A9 01      191  MISS    LDA  $$01
117D: 85 E7      192          STA  SCALE      ; RESET SCALE
117F: A9 0A      193          LDA  $$0A
1181: 85 0C      194          STA  NUM      ; RESET NUM
                        195  *
1183: 4C B8 10  196  AGAIN  JMP  CALC
                        197  *
1186: 66         198          CHK

```

This is an independent program that can be called from Applesoft BASIC by typing in `CALL 4096` or from the Monitor by typing in `1000G`. You can also directly `BRUN` the assembled object file.

When the program is run, a spaceship-like form similar to the one drawn in the explosion routine is drawn in the center of the screen. At the top of the screen, a wall made up of two horizontal lines is also drawn. Turning paddle 0 and pressing the corresponding pushbutton will fire a ray from the ship. If the ray hits the wall, an explosion occurs and the wall is left suitably damaged. You must press `RESET` to terminate the program.

The program combines many of the techniques described in this chapter and the previous one. It can be summarized as follows:

1. Initialize a shape table containing four shapes: a spaceship, two explosions, and a one-dot shape for the ray effect.
2. `HYPLOT` a wall of two horizontal lines at top of screen.
3. `DRAW` shape 1 (the spaceship) at the center of the screen.
4. Read paddle 0. Store the value in the rotation register.
5. Pause to encourage paddle reliability.
6. Read pushbutton 0. If it is not pressed, go back to step 4.
7. Button pushed: Start the fire sequence.
8. Draw a dot shape starting inside the ship. The rotation value set in step 4 determines the angle of the shot.
9. Make some noise with the simple noise routine.
10. Erase the dot shape.
11. Check the collision counter to see if anything was hit.
12. If nothing was hit, add 3 to the `SCALE` value. If it is still less than `#$90`, go back to step 8.
13. If there was no impact, restore `SCALE` to 1 and the explosion counter to `#$0A`. Then go back to step 4.
14. If something was hit, find the end of the ray by calling the Applesoft `HFIND` routine. Save this position value.
15. `DRAW` one of the explosion shapes in white.
16. Make some noise.
17. `DRAW` the same explosion shape in black to erase not only the shape, but also the parts of the wall that were hit.

18. Go back to step 15 ten times for an exciting (?) explosion.

19. Restore SCALE and the explosion counter. Go back to step 4.

Because this program is made up of the various routines used earlier, this summary should be sufficient to explain the overall method of operation.

The use of the single-dot shape to create the ray is similar to the technique used in chapter 24's Scanner programs. The new things presented in Shooter are the incrementing by three (lines 132–134) to create a faster firing appearance and the use of HFIND if an impact is detected.

Remember that the HFIND (\$F5CB) routine in Applesoft is used after drawing any shape to find out where we've been left. We used HFIND in this program to determine where the impact occurred.

Also note that DRAW rather than XDRAW is used in this program to ensure that portions of the wall are destroyed by the impact. In contrast to the explosion program, this program cannot be run on any hi-res screen background without changing the colors used by the ray and the explosion routines.

Passing Floating-Point Data

November 1982

In chapters 16 and 17 we discussed how Applesoft variable data could be passed from BASIC to assembly language and back again. The rationale was that in many cases a program created by combining Applesoft and assembly language is an effective approach to a problem. Successive chapters on hi-res graphics included these techniques so as to have a convenient way of experimenting with the various routines.

It is highly recommended that you review the appropriate chapters if you're not entirely familiar with the nature of Applesoft variable storage. Pages 127 and 137 in the *Applesoft II BASIC Programming Reference Manual* also provide very valuable information well worth referring to in the course of reading the material presented here.

For the most part, however, all of the past discussions were limited to dealing with two-byte integer data. That is to say, the possibility of dealing with true floating-point data was not considered. In many cases, integer values from 0 to 65535 or -32767 to 32767 are more than adequate for our purposes, as was the case when passing tone routines or X and Y coordinates for plotting.¹ However, there are times when greater precision, or fractional values, are required.

Dealing with floating-point numbers from a pure assembly-language program is a fairly complex topic, and our intent here is not to explain completely the inner workings of floating-point operations. Rather, let's explore the options made available by taking advantage of the existing routines in the Applesoft BASIC interpreter. These generally can be considered to be always present in the background of an operating assembly-language program.

For those of you who hope to speed up floating-point operations in Applesoft, writing your own routines may not be that effective. This is because the routines in Applesoft are already written in machine language. We can, however, gain important speed improvements just by calling the routines directly. This is because we can eliminate the normal process of interpreting BASIC statements that otherwise would occur in Applesoft. This is what the currently available compilers do, and we can expect similar speed improvements to a BASIC pro-

¹ [CT] Recall from chapter 10 that the minimum technically should be -32768 . However, Applesoft and Integer BASIC restrict the minimum integer to -32767 .

gram by using routines directly from assembly language (two to five times faster than in straight Applesoft).

Internalization of Data: Integer versus Real Variables

The first step in our inquiry is to investigate how Applesoft stores numeric data and to look at the differences in how integer variables and real variables are stored.

Start by initializing your Apple's memory with an FP statement. Then enter:

```
A% = 10: A = 10
```

The result is that two variables and their values have been set up in memory. Now to find them!

Enter the Monitor with the usual CALL -151. Then enter:

```
67 68 AF B0
```

You should get:

```
0067- 01
0068- 08
00AF- 03
00B0- 08
```

You may recall from chapter 16 that these four memory locations (\$67, \$68 and \$AF, \$B0) are used to store the beginning and the end of the current Applesoft program. We can see from the display that the program resides from \$801 to \$803. A very short program, indeed, but that's understandable since we haven't entered any program lines.

Now let's examine the pointer at \$69, \$6A and \$6B, \$6C. Do this by typing

```
69.6C
```

and pressing return. You should get:

```
0069- 03 08 11 08
```

This tells us that all simple (that is, non-array) variables are stored from \$803 to \$810.² Examine this area by entering:


```
803.810
```

You should get:

```
0803- C1 80 00 0A 00
0808- 00 00 41 00 84 20 00 00
0810- 00
```

² [CT] One less than \$811.

You'll recall from our discussions in previous chapters that integer and real variables are stored in the following format:

Integer:	C1	80	00	0A	00	00	00
	"A"	" "	"0"	"10"	—	—	—
	Name	Name	High	Low	Unused	Unused	Unused
	char1	char2	Byte	Byte			
	(bit 7 set)	(bit 7 set)					
Real:	41	00	84	20	00	00	00
	"A"	" "	1000	0010	0000	0000	0000
			0100	0000	0000	0000	0000
	Name	Name	Exponent	Mantissa	Mantissa	Mantissa	Mantissa
	char1	char2		m.s.b.			l.s.b.
	(bit 7 clear)	(bit 7 clear)					

Starting at \$803, we find the variable A% stored from \$803 to \$809. The first two bytes are the name characters. Two bytes are always used. If the variable name is only one character then a null (\$00 for real or \$80 for integer) is stored in the second position. Note that integer, real, and string variable names are differentiated by the combination of high bit settings in the two name-character bytes. Since only bits 0 through 6 are used for the character (ASCII is only a seven-bit code), bit 7 (the high-order bit) is available for encoding the variable type.

Integer variables always have both high bits set. Real variables always have both high bits clear. String variables always have the first name character clear and the second character set. (The notation for string variable names to the opposite effect on page 137 of the *Applesoft II BASIC Programming Reference Manual* is in error in this regard.)

The next two bytes, \$00 and \$0A, are the high- and low-order bytes for the value 10. You have probably noticed that integer variables are stored in a very simple way, with the value being broken down into the low- and high-order bytes. About the only peculiar item is the fact that the two bytes are stored high-order byte first, which is backward from the way we normally see them paired in most assembly-language code.

The three remaining bytes are unused.

\$80A to \$810 is where the real variable A is stored. You can see that the first two bytes again are the name characters, this time with the high bits clear. The remaining bytes make up the value for the variable A.

It should be obvious that although the values of the integer and real variables are stored as equal, the manner in which they are stored is not. The real variable has been encoded into a five-byte sequence, the logic behind which is not readily apparent. Well, don't despair; it is not actually necessary for us to understand the exact details of the conversion routine.

In general, it will suffice to say that an exponential notation is used to store the number. This is how numbers of such large magnitudes ($\pm 10^{38}$) are accommodated by Applesoft. If you rouse some of your more ancient high-school memories, you'll recall that the basic idea to exponents is that any number can be expressed with two numbers, the exponent and the mantissa.

For example, the number 10 is equal to 10^1 , The number 100 is equal to 10^2 . It is reasonable to assume, then, that a number like 50 might just happen to be equal to $10^{1.5}$. As it happens, that's not quite right, but the basic idea is there. In fact, 50 is really equal to $10^{1.69897}$ (or thereabouts). The 1 part of the number is called the *exponent* (or occasionally the order of magnitude) of the number. The 69897 is called the *mantissa*. You may have fond memories of spending pleasant hours in math classes looking through books with lookup tables to find these values for given numbers.

In any event, it's precisely this type of technique that is used to encode the values of real variables.³ Fortunately for us, it will not be necessary to create our own routines to handle numbers in this format; a wealth of such routines already exist in Applesoft.

The remainder of this chapter will concentrate on some brief exercises in passing floating-point numbers back and forth between Applesoft and assembly language. Then in upcoming chapters we'll explore how to perform various mathematical operations once your assembly-language program has possession of the data.

The Floating-Point Accumulator (FAC)

Applesoft has its own internal set of registers that it uses during its various calculations. The most important of these by far is the floating-point Accumulator. This is usually labeled FAC in source listings that access this register.

The word *register* is used in a slightly different way here than it is when referring to 6502 registers such as the Accumulator or the X- or Y-Registers. Because a floating-point number is represented by a series of bytes, the FAC occupies the bytes from \$9D through \$A2.

You may be puzzled as to why the FAC uses six bytes when variable storage uses only five. This is because the FAC uses \$A2 as the sign byte to indicate the positive or negative status of the value. When finally encoded, the sign is included in the exponent and mantissa bytes and thus is no longer needed. Floating point numbers in the five-byte format are said to be "packed." The six-

³ [CT] From the *Apple II Technical Notes*: for the exponent, the top bit is the sign (with 0 for negative). The remainder of the byte minus one is the value of the exponent (for example, \$84 is a positive exponent of 3). The mantissa is a binary fraction, with an implied starting value of 1. The first bit is the sign bit (this time with 0 for positive). The remaining bits are fractional values starting with 0.5, 0.25, 0.125, etc. For example, \$20 gives a mantissa of $1 + 0.25$. So \$84 \$20 equals $1.25 \times 2^3 = 10$.

byte format is “unpacked.” The unpacked format is faster for calculations. The packed format is used to minimize storage space.

In general, whenever any type of calculation is done by Applesoft the FAC is the primary register used to hold the result. A second register, ARG (ARGument), is used for two-value calculations, such as 1.5×17 . The ARG register uses the bytes \$A5 through \$AA. For the time being, though, we need only concern ourselves with FAC.

Passing Data from Applesoft to the FAC

The first area to investigate is how to get a floating-point number passed from Applesoft to an assembly-language routine. The easiest way is by means of the USR function. The USR is a rather neglected part of Applesoft, probably because of the lack of documentation on its nature and applications. A program statement using USR might look something like this:

```
10 X = USR(Y)
```

When this statement is executed, three things happen:

1. The expression or variable within the parentheses is evaluated and the result put in the FAC.
2. A call to location \$0A (decimal 10) is done. This is equivalent to a CALL 10 in Applesoft. There is a three-byte jump instruction at location \$0A. It is assumed that the user has inserted the location of an existing assembly-language routine. For example, the code JMP \$300 might be found at \$0A. The program would then jump to \$300 to execute whatever routine the user might have put there.
3. When the user routine eventually does an RTS, the contents of the FAC are assigned to the variable to the left of the equal sign.

For example, type in and run this program:

```
10 POKE 10,0
20 Y = 10
30 X = USR(Y)
```

When run, the program should fall into the Monitor. Then type in:

```
9D.A2 (return)
```

You should get:

```
009D- 84 A0 00
00A0- 00 00 20
```

This is the data for the value 10, which the FAC stores in unpacked form.⁴ Here's what happened: Line 10 set location \$0A to a BRK. When the USR function was called, it put the sequence for 10 in the FAC and then called \$0A as expected. Since this was a break, we went into the Monitor and could then immediately examine the FAC.

Note that it is not possible to set the FAC from Applesoft and then to verify the status of the FAC by entering the Monitor with the usual CALL -151. Since the FAC will be used in calculating the value of -151, any prior data would be overwritten.

While you're in the Monitor, let's set up \$0A for our next experiment. Enter:

```
0A: 4C 00 03
```

This will set the vector to point at location \$300. Now create a trivial program (in this case, an immediate RTS) at \$300 by entering:

```
300: 60
```

Now return to Applesoft and enter and run this program:

```
10 Y = 10
20 X = USR(Y)
30 PRINT X
```

You should get the number 10 printed out. If you consider what we've discussed so far, it should be apparent why. The value 10 held by Y was passed to the FAC by the USR function. When our "routine" at \$300 was called, the FAC remained unchanged. Upon return from our routine, the FAC (still equal to 10) was assigned to the variable X.

Although the USR function is a convenient way of passing data, it is rather limited in terms of syntax. If you wanted to pass a number of parameters to a routine, another technique would be required. You may recall from previous chapters a routine called FRMNUM (\$DD67 = FoRMula NUMeric evaluator) that we used to evaluate variables being passed to assembly-language routines. After calling FRMNUM, GETADR (\$E752 = GET AdDRess) was used to convert the number to a two-byte integer LINNUM (\$51, \$52 = LINE NUMBER).

Well, since what we want is the FAC, we've already got the solution:

```
1 *****
2 *
3 *      AL26-BASIC TO FAC      *
4 *
5 *      SYNTAX: CALL 768,Y     *
6 *****
7 *
8 *      OBJ  $300
9 *      ORG  $300
```

⁴ [CT] Corrected from the original article, which presented the result in packed form.

```

10 *
11 CHKCOM EQU $DEBE
12 FRMNUM EQU $DD67
13 *
0300: 20 BE DE 14 ENTRY JSR CHKCOM
0303: 20 67 DD 15 JSR FRMNUM
0306: 00 16 BRK

```

This code should be assembled at \$300 and called with the following Applesoft program:

```

10 Y = 10
20 CALL 768,Y

```

When this program is run, you should fall into the Monitor. Then enter:

```
9D.A2
```

You should get:

```

009D- 84 A0 00
00A0- 00 00 20

```

This should verify that the FAC was properly loaded with the value 10.⁵

In reviewing the listing, you'll see that line 14 calls CHKCOM (\$DEBE = CHECK for COMma) to advance Applesoft's TXTPTR (\$B8, \$B9 = TeXT PoinTeR) past the comma following the 768. Line 15 then calls FRMNUM, which evaluates the variable or expression following the comma and puts the result in the FAC. Line 16 then does the BRK to leave us in the Monitor, from which we can check the FAC to verify that the correct value has been stored.

We have now, then, two techniques for passing data from Applesoft to the FAC. The first is to use the USR function (being sure, of course, to set up the vector at \$0A). The second is to use FRMNUM (\$DD67) to evaluate the expression or variable as part of a parameter list following a CALL statement.

Moving the FAC to a Memory Location

Since the FAC is so heavily used, it is sometimes helpful to move the data in it to another location for later use. In Applesoft, this is most often a temporary register or an actual variable. For now, let's see if we can move the data to an arbitrary location.

```

1 *****
2 *
3 * AL26-FAC TO MEMORY *
4 *
5 * SYNTAX: CALL 768,Y *
6 *****

```

⁵ [CT] Again, the FAC stores the value in unpacked form.

```

7 *
8 *          OBJ $300
9 *          ORG $300
10 *
11 CHKCOM EQU $DEBE
12 FRMNUM EQU $DD67
13 MOVMF EQU $EB2B
14 *
0300: 20 BE DE 15 ENTRY JSR CHKCOM
0303: 20 67 DD 16 JSR FRMNUM ; BASIC->FAC
0306: A0 03 17 LDY #03 ; HI BYTE
0308: A2 80 18 LDX #80 ; LO BYTE
030A: 20 2B EB 19 JSR MOVMF ; FAC->MEMORY
030D: 60 20 DONE RTS
21 *

```

The key to this technique is a routine in Applesoft called MOVMF (\$EB2B = MOVE to Memory from FAC), which takes the value in FAC and moves it to the location pointed to by the X- and Y-Registers (X, Y = low byte, high byte).

The listing given here uses our previous FRMNUM technique to get a predictable number into the FAC. The X- and Y-Registers are then loaded to point to \$380. When MOVMF is called, the contents of the FAC will be deposited there.

To see this, run the same Applesoft program, then enter the Monitor and enter:

```
380.384
```

You should get:

```
380- 84 20 00 00 00
```

This proves that we have successfully moved the data from FAC to an arbitrary place in memory.⁶

Moving Memory into the FAC

The converse of this operation is accomplished in much the same way. In this case, the Applesoft routine MOVFM (\$EAF9 = MOVE to FAC from Memory) is used. It requires that the Y-Register and Accumulator be loaded with the high- and low-order bytes of the address to be used as the data source for the FAC. (Note that there is a difference here: MOVMF uses X and Y; MOVFM uses X and A!)

```

1 *****
2 *                                     *
3 *          AL26-MEMORY TO FAC        *
4 *                                     *
5 *          SYNTAX: CALL 768          *
6 *****

```

⁶ [CT] MOVMF first converts from unpacked FAC form to packed form, then moves the data. Similarly, MOVFM converts from packed form back to the unpacked FAC form.

```

7      *
8      *          OBJ  $300
9      *          ORG  $300
10     *
11     MOVFM     EQU  $EAF9
12     *
0300: A0 03    13     ENTRY  LDY  #$03          ; HI BYTE
0302: A9 80    14     LDA   #$80          ; LO BYTE
0304: 20 F9 EA 15     JSR   MOVFM         ; MEMORY->FAC
0307: 00      16     BRK

```

Assuming that the previous routine has already been executed and that \$380 is loaded with the data appropriate to the value 10, type in CALL 768.

You should end up in the Monitor, at which point you can verify the contents of the FAC by entering:

```
9D .A2
```

You should get:

```
9D- 84 A0 00
A0- 00 00 20
```

Again, the BRK was used to end the routine so that we could immediately examine the contents of the FAC. This routine shows that we can move data from a section of memory back into the FAC.

Passing FAC Data Back to Applesoft

If the FAC does contain the result of an operation, how can we pass it back to a calling Applesoft program, preferably into the variable of our choice? Again, the answer is to use MOVFM. In this case, rather than moving the contents of the FAC into an arbitrary memory location, we'll find the location of the data bytes of a given real variable and then move the FAC into them. This has the effect of setting the variable equal to the contents of the FAC.

Consider this listing:

```

1      *****
2      *
3      *          AL26-FAC TO BASIC          *
4      *
5      *          SYNTAX: CALL 768,Y        *
6      *****
7      *
8      *          OBJ  $300
9      *          ORG  $300
10     *
11     CHKCOM     EQU  $DEBE
12     PTRGET     EQU  $DFE3
13     MOVFM      EQU  $EB2B
14     MOVFM      EQU  $EAF9

```

```

15 *
0300: A0 03 16 ENTRY LDY #03 ; HI BYTE
0302: A9 80 17 LDA #080 ; LO BYTE
0304: 20 F9 EA 18 JSR MOVFM ; MEMORY->FAC
19 *
0307: 20 BE DE 20 JSR CHKCOM
030A: 20 E3 DF 21 JSR PTRGET
030D: AA 22 TAX
030E: 20 2B EB 23 JSR MOVMF ; FAC->VARIABLE
0311: 60 24 DONE RTS

```

This routine again assumes that the floating-point data for the number 10 still exists at \$380. When this routine is run, lines 16 through 18 duplicate the previous listing to move the floating-point data from \$380 through \$384 into the FAC.

Line 20 uses CHKCOM to check the comma and move TXTPTR to the first character past the comma. Line 21 uses the PTRGET (\$DFE3 = PoinTeR GET routine) to locate the variable currently pointed to by TXTPTR. PTRGET is handy also in that it will create the variable in the variable table if it does not already exist. PTRGET returns with the Y-Register and Accumulator pointing to the data bytes of the specified variable. This will be precisely where we want the data in the FAC to be moved to. The only correction to be made is in regard to the fact that MOVFM requires that the Y- and X-Registers (rather than Y and the Accumulator as was left by PTRGET) hold the destination address. Line 22 solves this by using the TAX command, at which point MOVFM is called. We're now done, and the RTS will return to the calling program.

Test this routine with the following listing:

```

10 CALL 768,X
20 PRINT X

```

X gets set to 10 by having our routine transfer the floating-point data from \$380 through \$384 to the data bytes for the variable X.

Putting it All Together

For a real test of these combined techniques, let's see if we can successfully pass data from Applesoft to the FAC to a memory block and then back to the FAC and back to Applesoft. The following routine should demonstrate the entire operation as an overall example of the ideas presented thus far.

```

1 *****
2 * *
3 * AL26-BASIC.FAC.MEM.FAC.BAS *
4 * *
5 * SYNTAX: CALL 768,Y,X *
6 *****
7 *
8 * OBJ $300

```

```

9          ORG  $300
10         *
11        CHKCOM EQU  $DEBE
12        PTRGET EQU  $DFE3
13        FRMNUM EQU  $DD67
14        MOVFM  EQU  $EAF9
15        MOVMF  EQU  $EB2B
16        *
0300: 20 BE DE 17 ENTRY   JSR  CHKCOM
0303: 20 67 DD 18         JSR  FRMNUM      ; FP->FAC
19        *-----*
0306: A0 03 20         LDY  #$03      ; HI BYTE
0308: A2 80 21         LDX  #$80      ; LO BYTE
030A: 20 2B EB 22         JSR  MOVMF      ; FAC->MEMORY
23        *-----*
030D: A0 03 24         LDY  #$03
030F: A9 80 25         LDA  #$80
0311: 20 F9 EA 26         JSR  MOVFM      ; MEMORY->FAC
27        *-----*
0314: 20 BE DE 28         JSR  CHKCOM
0317: 20 E3 DF 29         JSR  PTRGET
031A: AA 30 30         TAX           ; MOVE LO BYTE->X
031B: 20 2B EB 31         JSR  MOVMF      ; FAC->FP
32        *-----*
031E: 60 33 DONE   RTS

```

Try this Applesoft program to call the routine:

```

10 Y = 10
20 CALL 768,Y,X
30 PRINT X

```

The value 10 should be printed for X. Dashed lines have been used to separate the four major sections of the routine. When you compare each section with the four routines presented, the net operation of the example should become clear.

The USR routine also could have been used and would eliminate two of the sections:

```

1 *****
2 *
3 * AL26-BASIC.FAC.MEM.FAC.BAS *
4 * VIA THE 'USR' *
5 *
6 * SYNTAX: X = USR(Y) *
7 *****
8 *
9 * OBJ $300
10 * ORG $300
11 *
12 CHKCOM EQU  $DEBE
13 PTRGET EQU  $DFE3
14 FRMNUM EQU  $DD67
15 MOVFM  EQU  $EAF9

```



```

                16  MOVMF   EQU   $EB2B
                17  *
0300: A0 03     18  ENTRY   LDY   #$03      ; HI BYTE
0302: A2 80     19          LDX   #$80      ; LO BYTE
0304: 20 2B EB  20          JSR   MOVMF     ; FAC->MEMORY
                21  *-----*
0307: A0 03     22          LDY   #$03
0309: A9 80     23          LDA   #$80
030B: 20 F9 EA  24          JSR   MOVFM     ; MEMORY->FAC
                25  *-----*
030E: 60       26  DONE   RTS

```

Notice that since the USR function calls the routine with the FAC already loaded with the value for Y, the first section of the previous routine is not needed. Also, since the USR function will automatically assign the contents of the FAC to the variable X, the last section of the previous routine is not needed.

The calling program for the routine would look like this:

```

10  POKE 11,0: POKE 12,3: REM SET UP USR VECTOR
20  Y = 10
30  X = USR(Y)
40  PRINT X: REM SHOULD PRINT '10'

```

Conclusion

By now you probably feel fairly comfortable with the idea of the floating-point Accumulator (FAC) and how data can be moved about between Applesoft and assembly language. In the next chapter we'll begin looking at some of the more sophisticated routines Applesoft uses to perform various arithmetic functions.



Floating-Point Math Routines

December 1982

In this chapter, we'll continue with our discussion of floating-point number operations. In the previous chapter we looked at how Applesoft uses the floating-point Accumulator (FAC) as the main register for most of its numeric operations. Routines were presented that demonstrated how data can be passed back and forth between a running Applesoft program and an assembly-language subroutine and also how numeric data can be moved in and out of block memory storage.

Using this foundation, we can now examine how to use Applesoft's routines for such basic math functions as addition, subtraction, multiplication, and division.

A word of advice is in order, however, before proceeding. Your first inclination may be to think that the routines given here will enable you to do simple math operations with greater speed in an Applesoft program. As it happens, this will not directly produce the speed increase you want. Remember, Applesoft is already using these very same routines; given that, no speed increase should be expected for such simple operations as $X = 5 \times 10$.

Our new syntax will be:

```
CALL 768,5,10,X
```

and as such involves just about as much overhead in the calling of the routine and the passing of data as would be involved in Applesoft.

You may ask then, "Why use an assembly-language call for these operations?"

There are a number of reasons, two of the more important of which follow.

First, when dealing with programs that require a high degree of accuracy, integer data may not be sufficient.

Suppose, for example, you have a program that simulates the motion of an object traveling in an elliptical (or other mathematically complex) path.

If the current position of the object is continually maintained by using integer coordinates in the range of the normal screen coordinates, errors will begin to creep in with successive recalculations of position. This might be evidenced by the figure failing to retrace itself. Although in theory the object should always return to its starting point when following an elliptical path, multiple rounding

errors may cause the object to “miss” its original starting point by a few screen units.

A more reliable approach would be to maintain the current position in a true floating-point format and round the number to the nearest integer prior to each plot.

Second, calculations dealing with a large number of variables or, more specifically, with arrays of real variables, will be executed faster by an assembly-language routine.

If, for example, you wanted to multiply an entire array by 5, it would in fact be faster to employ a routine that used this syntax:

```
CALL 768, A(0), 5
```

than to use:

```
FOR I = 1 TO 100: A(I) = A(I) * 5: NEXT I
```

These ideas and others are put to use in a number of commercial software products for the Apple. Many programs that require both speed and a high degree of accuracy use floating-point representations of numbers in assembly-language routines. Another common technique is to use the BCD (“Binary Coded Decimal”) format for the data.

There are also programming utilities that provide machine-language routines to be called directly from Applesoft. *Routine Machine*, *AmperMagic*, *Amperware*, *Apple Spice*, and *The Linker* (published by Southwestern Data Systems, Anthro-Digital Software, Scientific Software Products, Adventure International, and Micro Lab, respectively) are all designed to allow the programmer access to useful routines written entirely in machine language. One product in particular, *Ampersoft Program Library Vol. 1* (a *Routine Machine*-related package), deals almost entirely with array-related routines that use the advantages of the second principle mentioned to speed up array-related programs.

In general, all of these products are based on making use of the ampersand vector to call specialized routines, the way we’ve discussed in past chapters. And regardless of which package you prefer, it’s safe to say that the overall idea of a user-selectable library of prewritten machine-language routines easily called from BASIC is one of the most powerful and exciting ideas to come along in Applesoft programming in the last few years. In fact, virtually all of the routines presented over the last year are compatible with many of these ampersand utility packages.

As a matter of reference, it also should be noted that techniques are available for faster numeric operations without having to call Applesoft routines. These range from arithmetic processor boards, such as those manufactured by California Computer Systems and others, to software subsystems such as *Speed/ASM* (published by Sierra On-Line). A combination of both hardware and soft-

ware is also available from Applied Analytics in the form of *Micro-Speed*, a Forth-like language combined with an arithmetic processor board. ALF Products offers an 8088 processor card that includes software that speeds up Applesoft math functions. It also allows the calling of dedicated math functions from a recurring machine-language program, independent of Applesoft.

More Applesoft Internals

Well, then, just how does a person use the existing routines in Applesoft? As with most things we've covered, the important thing to know is the addresses of the entry points to the Applesoft routines for the basic math operations that interest us. We also need to revive the discussion of the ARG ("argument") register, which we mentioned briefly in the previous chapter.

The ARG register is identical in format to the FAC and is used to hold the second number in floating-point format when doing two-value functions such as addition, subtraction, multiplication, and division. The ARG register uses bytes \$A5 through \$AA.

To see how ARG is used, consider these important entry points to Applesoft math routines:

Function	ARG <func> FAC	MEM <func> FAC
Addition	FADDT (\$E7C1)	FADD (\$E7BE)
Subtraction	FSUBT (\$E7AA)	FSUB (\$E7A7)
Multiplication	FMULTT (\$E982)	FMULT (\$E97F)
Division	FDIVT (\$EA69)	FDIV (\$EA66)

For the first column of labels, the associated addresses show the entry point for the routines that will perform the given function between the ARG register and the FAC. For example, a call to FSUBT (\$E7AA) would subtract the contents of the FAC from the contents of the ARG. The result would be left in the FAC.

Prior to calling any of these four routines, the Accumulator must be loaded with the exponent value of the FAC (FACEXP = \$9D). This also serves to condition the zero flag. For example, to multiply FAC times ARG, the following code could be used;

```
LDA $9D
JSR $E982
```

The second column of labels refers to the routines used to perform the indicated function between the FAC and data stored in memory (such as in a real variable) or in a data block set up by the programmer.

To use these, the Y-Register and Accumulator must be set up with the address of the memory location holding the numeric data (Y, A = high byte, low

byte). When a routine is called, the data pointed to by Y, A will then be transferred into ARG and the direct function routine (first column) then called.

An Example That Doesn't Work

You may wonder why a sample listing that doesn't work is included here. The reason is that this listing does present, in a clear way, an overall example of what we've been discussing in this chapter and the previous one. It will also help you understand the changes we'll be making later on in order to create a routine that does work!

From Applesoft, the routine would be called from a program like this:

```
10 INPUT "X1, X2:"; X1, X2
20 CALL 768, X1, X2, RSLT
30 PRINT XI; " + "; X2; " = "; RSLT
```

Where X1 and X2 are the two arguments for the addition routine, that routine will be called. The result of the calculation will be sent back to the Applesoft program into the variable RSLT.

Here's the listing for the addition routine:

```

1 *****
2 *
3 * AL27-M.L. ADDITION SUBR 1 *
4 * (DOESN'T WORK) *
5 * *
6 * SYNTAX: CALL 768,X1,X2,RSLT *
7 * RSLT = X1 + X2 *
8 *****
9 *
10 * OBJ $300
11 * ORG $300
12 *
13 CHKCOM EQU $DEBE
14 PTRGET EQU $DFE3
15 FRMNUM EQU $DD67
16 FACEXP EQU $9D
17 MOVMF EQU $EB2B
18 MOVAF EQU $EB63
19 FADDT EQU $E7C1
20 *
0300: 20 BE DE 21 ENTRY JSR CHKCOM
0303: 20 67 DD 22 X1 JSR FRMNUM ; FP -> FAC
0306: 20 63 EB 23 JSR MOVAF ; FAC -> ARG
24 *
0309: 20 BE DE 25 X2 JSR CHKCOM
030C: 20 67 DD 26 JSR FRMNUM ; FP -> FAC
27 *
030F: A5 9D 28 ADD LDA FACEXP
0311: 20 C1 E7 29 JSR FADDT ; X1 + X2
30 *
0314: 20 BE DE 31 RSLT JSR CHKCOM
```

```

0317: 20 E3 DF 32          JSR  PTRGET
031A: AA      33          TAX          ; MOVE LO BYTE TO X
031B: 20 2B EB 34          JSR  MOVMF   ; FAC -> FP
                35      *
031E: 60      36          RTS

```

Line 21 begins the routine by first taking care of the comma following the 768 in the CALL statement. FRMNUM (\$DD67) is then used to evaluate the first expression. FRMNUM conveniently leaves the result in the FAC. Since we will want the first argument in the ARG register, MOVAF (\$EB63 = MOVE to ARG from FAC) is then used to move the data.

Line 25 again calls CHKCOM to “gobble” the next comma, after which FRMNUM is again used to evaluate the next value and place it in the FAC.

We would now expect the result to be in the FAC. Line 31 takes care of the third comma, after which PTRGET (\$DFE3) finds (or creates) the variable in which we want the result returned.

At this point, everything has been properly placed for the use of the FADDT routine to add the FAC and ARG registers together. Line 28 loads the Accumulator with FACEXP (\$9D) as the entry requirement for the next instruction, which is the actual execution of the FADDT routine.

The TAX on line 33 is used after PTRGET to move the low-order byte of the variable data address into the X-Register, after which MOVMF (\$EB2B = MOVE to Memory from FAC) is used to complete the data transfer.

Note: If you’re unfamiliar with the fundamental move routines, you may wish to go back to the previous chapter, which covered these supporting routines.

The nice part about this routine is how easily the setup for the addition routine was accomplished. With a little thought, though, you may realize this is to be expected. After all, the internal routines were created in the first place to process data easily within an Applesoft program.

Why it Doesn’t Work

The routine fails because of FRMNUM. Although it was mentioned that FRMNUM leaves its result in FAC, what you weren’t let in on was the fact that it also uses ARG during its calculations. This means that when we call FRMNUM a second time on line 26, we are unknowingly destroying the value we set up in ARG in lines 22 and 23.

The solution, then, is to save the FAC contents from the first value calculation in memory at a place other than ARG.

There are two alternatives. The first is to use some of Applesoft’s own temporary numeric registers, which are called, cleverly enough, TEMP1 (\$93–\$97), TEMP2 (\$98–\$9C), and TEMP3 (\$8A–\$8E). The only risk here is in the destruction

of data later on by other temporary calculations by FRMNUM (\$DD67) and FRMEVL (\$DD7B).

Another possibility would be to set aside our own temporary storage area. For this next example we'll do just that, using the last half of the input buffer, \$280-\$284.

Here's the revised listing, called using the same Applesoft program as before:

```

1 *****
2 * *
3 * AL27-M.L. ADDITION SUBR 2 *
4 * *
5 * SYNTAX: CALL 768,X1,X2,RSLT *
6 * RSLT = X1 + X2 *
7 *****
8 *
9 * OBJ $300
10 * ORG $300
11 *
12 CHKCOM EQU $DEBE
13 PTRGET EQU $DFE3
14 FRMNUM EQU $DD67
15 FACEXP EQU $9D
16 MOVMF EQU $EB2B
17 CONUPK EQU $E9E3
18 FADDT EQU $E7C1
19 *
0300: 20 BE DE 20 ENTRY JSR CHKCOM
0303: 20 67 DD 21 X1 JSR FRMNUM ; FP -> FAC
22 *
0306: A0 02 23 LDY #$02
0308: A2 80 24 LDX #$80 ; $280
030A: 20 2B EB 25 JSR MOVMF ; FAC -> MEMORY
26 *
030D: 20 BE DE 27 X2 JSR CHKCOM
0310: 20 67 DD 28 JSR FRMNUM ; FP -> FAC
29 *
0313: A0 02 30 ADD LDY #$02
0315: A9 80 31 LDA #$80 ; $280
0317: 20 E3 E9 32 JSR CONUPK ; MEMORY -> ARG
031A: A5 9D 33 LDA FACEXP
031C: 20 C1 E7 34 JSR FADDT ; X1 + X2
35 *
031F: 20 BE DE 36 RSLT JSR CHKCOM
0322: 20 E3 DF 37 JSR PTRGET
0325: AA 38 TAX ; MOVE LO BYTE TO X
0326: 20 2B EB 39 JSR MOVMF ; FAC -> FP
40 *
0329: 60 41 RTS

```

You'll notice in this listing that lines 23 and 24 set up the Y- and X-Registers for the subsequent call to MOVMF. This stores the data for the first value safely in memory.

The word “safely” is used with certain caveats. The input buffer is a useful area in which to store temporary data, but you should be aware of the kinds of conditions that will overwrite data placed there. DOS commands and input statements are the most likely threats. Also, commands executed from the immediate mode can overwrite the input buffer. This is in fact why we used \$280–\$284 for the temporary register. This allows you to try the routine from the immediate mode, since you are unlikely to use more than 127 characters as your command line when testing the routine.

Once the data is stored safely in memory, line 28 evaluates the next value, leaving the result in the FAC. At this point we use another routine, CONUPK (\$E9E3 = CONvert (?) and UnPacK), to move the data from \$280–\$284 back to ARG. Remember, this is necessary because FRMNUM on line 28 makes it impossible to store the value for X1 in ARG.

After CONUPK puts the data back in ARG, FADDT (\$E7C1) adds FAC to ARG.

A Little More Finesse

In the chart showing the various math routine entry points, you’ll remember that there was a set of routines that allow for dealing with data in memory directly. We can use these to create a slightly smaller version of the previous program which will eliminate our having to load ARG directly prior to calling FADDT. Here’s the improved listing:

```

1 *****
2 *                                     *
3 * AL27-M.L. ADDITION SUBR 3        *
4 *                                     *
5 * SYNTAX: CALL 768,X1,X2,RSLT *
6 *           RSLT = X1 + X2         *
7 *****
8 *
9 *           OBJ $300
10          ORG $300
11 *
12 CHKCOM EQU $DEBE
13 PTRGET EQU $DFE3
14 FRMNUM EQU $DD67
15 FACEXP EQU $9D
16 MOVMF EQU $EB2B
17 CONUPK EQU $E9E3
18 FADD EQU $E7BE
19 *
0300: 20 BE DE 20 ENTRY JSR CHKCOM
0303: 20 67 DD 21 X1 JSR FRMNUM ; FP -> FAC
22 *
0306: A0 02 23 LDY #$02
0308: A2 80 24 LDX #$80 ; $280
030A: 20 2B EB 25 JSR MOVMF ; FAC -> MEMORY
26 *
```



```

030D: 20 BE DE 27 X2      JSR  CHKCOM
0310: 20 67 DD 28      JSR  FRMNUM      ; FP -> FAC
          29 *
0313: A0 02 30 ADD      LDY  #$02
0315: A9 80 31          LDA  #$80      ; $280
0317: 20 BE E7 32      JSR  FADD      ; X1 + X2
          33 *
031A: 20 BE DE 34 RSLT   JSR  CHKCOM
031D: 20 E3 DF 35      JSR  PTRGET
0320: AA 36          TAX          ; MOVE LO BYTE TO X
0321: 20 2B EB 37      JSR  MOVMF     ; FAC -> FP
          38 *
0324: 60 39          RTS

```

The only difference between this routine and the previous one is that line 30 now sets up the Y-Register and Accumulator for a direct call to FADD (\$E7BE). This entry point automatically transfers the contents of \$280–\$284 to ARG and then “falls into” FADDT (\$E7C1).

Other Operations: Subtraction, Multiplication, and So On

Creating routines to do the other three functions is very simple. Rewriting lines 18 and 32 of the improved listing to use FSUB (\$E7A7), FMULT (\$E97F), and FDIV (\$EA66) will create the routines to perform the corresponding functions.

As it happens, there’s also a variety of other simple functions that can be performed on the FAC with a single JSR. A brief list is presented in Appendix D (Monitor Subroutines).

Information like what’s given in that list is quite valuable, if not indispensable, when you’re writing your own assembly-language routines that use Applesoft. There are a few notable sources for such information. The first is in an article by John Crossley of Apple Computer called “Applesoft Internal Entry Points,” which has been reprinted in a number of places including *Apple Orchard*, *Call-A.P.P.L.E.*, and *Call-A.P.P.L.E. In Depth #1*.

There is also a book called *What’s Where in the Apple?* by William F. Luebert that lists many of the entry points to not only Applesoft but also to the Monitor, DOS, Integer BASIC, and more.

Conclusion

We have seen how the FAC and ARG registers are used as the central points in almost all of Applesoft’s numeric calculations. In addition (no pun intended), we have seen how the individual math routines are called to perform the desired functions.

These new routines should be very useful in creating your own floating-point utilities. You may wish to try to create a routine to perform a simple function on an entire array as an exercise in using these new techniques.

The BCD, or Binary Coded Decimal

January 1983

This chapter's discussion centers on a little-mentioned operational mode of the 6502 microprocessor known as BCD, which stands for Binary Coded Decimal. In previous chapters we've looked at arithmetic operations that use binary and hexadecimal representations of the numbers involved. Such operations often require a certain degree of mental translation to produce a decimal equivalent. In terms of printing a number in ASCII form, even more difficulty is to be expected if you're using your own conversion routines rather than the built-in functions of DOS, Applesoft, and Integer BASIC.

The BCD mode greatly simplifies this process by storing numbers in one or more byte registers (either memory, X, Y, or the Accumulator) in a decimal-oriented manner. It does this by using two four-bit groups in each byte to represent a digit in base ten. In this way two digits per byte can be stored, thus giving a total value range of 0 to 99, versus 0 to 255 using binary.

This table provides an example of how the BCD counting scheme goes:

BCD	Hex	Binary	"Real Value"
0	\$00	0000 0000	0
1	\$01	0000 0001	1
2	\$02	0000 0010	2
3	\$03	0000 0011	3
.			
.			
9	\$09	0000 1001	9
10	\$10	0001 0000	16
11	\$11	0001 0001	17
.			
.			
14	\$14	0001 0100	20
15	\$15	0001 0101	21
16	\$16	0001 0110	22
17	\$17	0001 0111	23
18	\$18	0001 1000	24
19	\$19	0001 1001	25
20	\$20	0010 0000	32

One of the nice things about hexadecimal notation is that each digit of the hex number represents one-half (four bits) of the binary number. This is a great help when you must mentally convert from hex to binary and back again. BCD is

a variation on this theme in which the hex number really can be said to equal the decimal value (that is, the decimal and hex columns will always match).

About this time you may be thinking, “Well, that’s all very nice, but where does the 6502 come into the picture?”

So far, all we have here is a possible system for storing decimal numbers via our usual hex bytes. The good news is that the 6502 actually supports this mode in the addition and subtraction operations.

That’s right. The secret to making it work is to tell the 6502 that you wish to operate in this mode. This is done by means of the instruction `SED`, which stands for `SEt Decimal mode`. Once this instruction has been executed, all future add and subtract operations will be done in the BCD mode. When you’re done, be sure to clear everything back to normal with the `CLD`, for `CLear Decimal mode`, instruction.

Special note: Inadvertent setting of the decimal mode can cause the Apple to behave rather strangely and can be most puzzling when you’re trying to debug programs. `RESET` does not clear the decimal flag (bit 3 of the Status Register).¹ When in doubt do a `CALL -155`, or `FF65G` from the Monitor, to clear the decimal mode.

Let’s verify that this mode actually works with a sample program:²

```

1 *****
2 *
3 * AL28-BCD DEMO ROUTINE 1 *
4 *
5 *****
6 *
8000: F8 7 START SED ; SET BCD MODE
8001: 18 8 CLC
8002: A9 46 9 LDA #$46
8004: 69 38 10 ADC #$38
8006: D8 11 CLD
8007: 00 12 DONE BRK ; BRK TO DISPLAY

```

Using the `BRK` command is an easy way both to end the program and display the result of the addition in the Accumulator. When this routine is called with either an `8000G` or a `CALL 32768` from BASIC, you should get the Monitor break response with a display something like this:

```
8009- A=84 X=90 Y=00 P=34 S=DE
```

Ignoring the rest of the line, when we see the `A=84` we know that the Accumulator holds 84, the correct result of the addition operation. You can substitute other numbers to verify that it works correctly with all legal values.

¹ [CT] Actually, even though the 6502 CPU `RESET` does not clear the decimal flag, in the Apple ROM the `RESET` code does issue a `CLD`.

² [CT] The original sample numbers were 12 and 34, which actually have the same sum in BCD and normal mode. With 46 and 38, the sum is 84 in BCD but `$7E` in normal mode.

A similar experiment works with subtraction:³

```

1 *****
2 *
3 * AL28-BCD DEMO ROUTINE 2 *
4 *
5 *****
6 *
8000: F8 7 START SED ; SET BCD MODE
8001: 38 8 SEC
8002: A9 46 9 LDA #$46
8004: E9 38 10 SBC #$38
8006: D8 11 CLD
8007: 00 12 DONE BRK ; BRK TO DISPLAY

```

In this case, the result should be 8. Again, you may wish to substitute different values to verify its operation.

For both addition and subtraction, results of the operations “wrap ground” in a manner similar to the way hexadecimal calculations do. That is to say that $99 + 1$ will give a result of 00 (100 less the leading 1) and $0 - 1$ will give 9.

Limitations

Like everything else in life, BCD has its tradeoffs and failings. The first involves that vague reference made earlier to everything working with “legal values.” “What’s legal?” you may ask. You’ll note that certain hex values, such as \$0A, never appear. This is because in the BCD mode such a value is “illegal” because it uses a digit out of the range of 0 to 9. If you attempt to use such a value in the BCD mode, you’ll get inaccurate results.

To add to the fun, note also that the BEQ, BNE and INC, DEC families of instructions don’t work as expected either. The N-flag (sign/negative flag) and Z-flag (zero flag) are all linked to binary operations and not to BCD. Thus $01 + 99$ will yield 00, but N and Z remain unaffected, since the “true” binary result should have been \$9A. Also, no provision is made for negative numbers (signed arithmetic). How, then, do we test for special conditions?

The Carry Flag

The carry flag is the only direct indication of arithmetic results in BCD. In addition operations, the carry will be set if the result exceeds 99 (overflow). In subtraction, the carry will be cleared if the result is less than 0 (underflow).

In multiple-byte operations the carry is used in the same way as it is in “normal” hexadecimal arithmetic.

³ [CT] The original code incorrectly had CLC instead of SEC. In addition, the original sample numbers were 34 and 12, which actually give the same result in BCD and normal mode. With 46 and 38, subtraction gives 8 in BCD but \$0E in normal mode.

Common Operations

Since INC and DEC don't perform properly in the BCD mode, their functions must be implemented by using the ADC and SBC instructions:

```

1 *****
2 *                                     *
3 * AL28-BCD DEMO 'INC' ROUTINE *
4 *                                     *
5 *****
6 *
7 MEM      EQU  $06
8 BEEP     EQU  $FBDD
9 *
8000: F8   10  START   SED           ; SET BCD MODE
8001: 18   11          CLC
8002: A5 06 12          LDA  MEM
8004: 69 01 13          ADC  #$01
8006: B0 04 14          BCS  ERR      ; OVERFLOW
8008: 85 06 15          STA  MEM      ; MEM = MEM + 1
800A: D8   16          CLD
800B: 60   17  DONE   RTS
800C: 4C DD FB 18  ERR    JMP  BEEP

```

```

1 *****
2 *                                     *
3 * AL28-BCD DEMO 'DEC' ROUTINE *
4 *                                     *
5 *****
6 *
7 MEM      EQU  $06
8 BEEP     EQU  $FBDD
9 *
8000: F8   10  START   SED           ; SET BCD MODE
8001: 18   11          SEC
8002: A5 06 12          LDA  MEM
8004: E9 01 13          SBC  #$01
8006: 90 04 14          BCC  ERR      ; UNDERFLOW
8008: 85 06 15          STA  MEM      ; MEM = MEM - 1
800A: D8   16          CLD
800B: 60   17  DONE   RTS
800C: 4C DD FB 18  ERR    JMP  BEEP

```

Notice how the carry status is checked to detect overflow (result > 99) or underflow (result < 0) in the addition and subtraction routines, respectively. MEM is a memory location presumed to hold a legal BCD value.⁴

⁴ [CT] There is one problem with all of these routines: In the case of an error, the CLD is never reached. If you run these routines with a value in \$06 that causes an overflow (or underflow), your Apple will issue a strange “bah-beep”. Luckily, the BEEP subroutine clears the decimal mode before returning. However, in your own programs, you should be sure to issue a CLD for all code paths.

Multiple-byte operations are done in a manner similar to the way their hexadecimal equivalents are handled:

```

1 *****
2 *
3 * AL28-BCD ADDITION ROUTINE *
4 *
5 *****
6 *
7 *
8 MEM1 EQU $06 ; 6,7
9 MEM2 EQU $08 ; 8,9
10 RSLT EQU $0A ; A,B
11 BEEP EQU $FBDD
12 *
8000: F8 13 ENTRY SED
8001: 18 14 CLC
8002: A5 06 15 LDA MEM1
8004: 65 08 16 ADC MEM2
8006: 85 0A 17 STA RSLT
8008: A5 07 18 LDA MEM1+1
800A: 65 09 19 ADC MEM2+1
800C: 85 0B 20 STA RSLT+1 ; RSLT = MEM1 + MEM2
800E: B0 02 21 BCS ERR ; OVERFLOW
8010: D8 22 CLD
8011: 60 23 DONE RTS
8012: 4C DD FB 24 ERR JMP BEEP

```

```

1 *****
2 *
3 * AL28-BCD SUBTRACT ROUTINE *
4 *
5 *****
6 *
7 *
8 MEM1 EQU $06 ; 6,7
9 MEM2 EQU $08 ; 8,9
10 RSLT EQU $0A ; A,B
11 BEEP EQU $FBDD
12 *
8000: F8 13 ENTRY SED
8001: 38 14 SEC
8002: A5 06 15 LDA MEM1
8004: E5 08 16 SBC MEM2
8006: 85 0A 17 STA RSLT
8008: A5 07 18 LDA MEM1+1
800A: E5 09 19 SBC MEM2+1
800C: 85 0B 20 STA RSLT+1 ; RSLT = MEM1 - MEM2
800E: 90 02 21 BCC ERR ; UNDERFLOW
8010: D8 22 CLD
8011: 60 23 DONE RTS
8012: 4C DD FB 24 ERR JMP BEEP

```

Printing BCD Values

One of the biggest advantages of BCD is that the values are easily printed to the screen or disk. When using hexadecimal math, some sort of hex-to-ASCII string decimal conversion routine is required. This is then followed by the printing of the digits via some string print routine. In BCD, only a minimal conversion is needed, and the printing is done fairly easily.

The easiest way to print a number is to use one of the Monitor routines. PRBYTE (\$FDDA = PRint BYTE), for example, prints the contents of the Accumulator as a hex byte. Here's a routine that takes two BCD values from memory and prints the sum:

```

1 *****
2 * *
3 * AL28-BCD PRINT ROUTINE 1 *
4 * *
5 *****
6 *
7 *
8 MEM1 EQU $06
9 MEM2 EQU $07
10 PRBYTE EQU $FDDA
11 BEEP EQU $FBDD
12 *
8000: F8 13 ENTRY SED
8001: 18 14 CLC
8002: A5 06 15 LDA MEM1
8004: 65 07 16 ADC MEM2 ; ACC = MEM1 + MEM2
8006: B0 05 17 BCS ERR ; OVERFLOW
8008: D8 18 CLD
8009: 20 DA FD 19 JSR PRBYTE
800C: 60 20 DONE RTS
800D: 4C DD FB 21 ERR JMP BEEP

```

You can experiment by putting different values in \$06 and \$07 and calling the routine. For two-byte values (0 to 9999) one can use PRNTAX (\$F941 = PRiNT Accumulator and X-Register), which expects the Accumulator and X-Register to be loaded with the bytes to be printed prior to the call:

```

1 *****
2 * *
3 * AL28-BCD PRINT ROUTINE 2 *
4 * *
5 *****
6 *
7 *
8 MEM1 EQU $06 ; 6,7
9 MEM2 EQU $08 ; 8,9
10 PRNTAX EQU $F941
11 BEEP EQU $FBDD
12 *
8000: F8 13 ENTRY SED

```

```

8001: 18          14          CLC
8002: A5 06      15          LDA MEM1
8004: 65 08      16          ADC MEM2
8006: AA         17          TAX           ; STORE RSLT IN X
8007: A5 07      18          LDA MEM1+1
8009: 65 09      19          ADC MEM2+1    ; RSLT+1 IN ACC
800B: B0 05      20          BCS ERR       ; OVERFLOW
800D: D8         21          CLD           ; CLR FOR PRNTAX
800E: 20 41 F9   22          JSR PRNTAX
8011: 60         23  DONE    RTS
8012: 4C DD FB   24  ERR     JMP BEEP

```

It is important to notice that in each routine the CLD is used to clear the decimal mode *before* calling PRBYTE or PRNTAX. This is because the Monitor needs the normal binary mode to calculate screen addresses and positions properly. If you call the Monitor with the BCD mode set, strange things will happen when the text reaches the end of the line or the screen needs to be scrolled and the Monitor routines attempt to calculate where to put the next line of text.

If you don't want to use the Monitor byte print routines or, for whatever reason, just want to create the ASCII characters yourself, the conversions are straightforward and COUT (\$FDED = Character OUTput—usually pronounced “C-out”) can be used directly.

The only real obstacle is how to convert the BCD digits to their ASCII equivalents. As it happens, this is even easier to do than you might at first suppose. Consider the table at the right.

From looking at the table, we can see that the lower digit of the ASCII value corresponds to the digit encoded in the BCD format and, coincidentally enough, to the number itself to be printed. If

there was a way of adding \$B0 to the value for the digit to be printed, we'd have just the value we needed to send to COUT to print the appropriate character.

To add \$B0 to the BCD values shown would normally require the usual CLC, ADC instructions. There is a more elegant (that is, shorter) way, however. You may remember that the ORA (logical OR with Accumulator) can be used as a mask to perform an overlay-like operation.

Here's how a possible ORA operation would appear:

```

Accumulator: 0000 0110   ($06 BCD)
ORA #B0:      1011 0000
Result:       1011 0110   ($B6 = ASCII "6")

```

Letter	ASCII Value*	BCD Value
0	\$B0	\$00
1	\$B1	\$01
2	\$B2	\$02
3	\$B3	\$03
4	\$B4	\$04
5	\$B5	\$05
6	\$B6	\$06
7	\$B7	\$07
8	\$B8	\$08
9	\$B9	\$09

*high bit set

What if the upper BCD digit is involved? The procedure then is first to shift the upper four bits "down" to the lower nibble position:

```
BCD value:  0101 0000  ($50 BCD)
LSR         0010 1000
LSR         0001 0100
LSR         0000 1010
LSR         0000 0101
Result:     0000 0101  ($05 BCD)
```

Ah, you ask, what if both digits possible are indicated by the BCD value? The answer here is first to shift the upper nibble down to the lower nibble, as was just shown, and to print the ASCII character arrived at. Then the original value is reloaded into the Accumulator and the upper nibble is masked out. This can be done using the AND instruction, which has the ability to clear a designated portion of a byte to zeros. For example:

```
Accumulator: 0101 0110  ($56 BCD)
AND #$0F:    0000 1111
Result #1:   0000 0110  ($06)
ORA #$B0:    1011 0000
Result #2:   1011 0110  ($B6 = ASCII "6")
```

Here, then, is the complete routine:

```

1  *****
2  *
3  * AL28-BCD PRINT ROUTINE 3 *
4  *
5  *****
6  *
7  *
8  MEM      EQU  $06
9  COUT     EQU  $FDED
10 *
8000: D8    11  ENTRY   CLD           ; BCD MODE NOT NECC
8001: A5 06  12          LDA  MEM     ; GET BCD NUMBER
8003: 4A    13          LSR         ; SHIFT UPPER NIBBLE
8004: 4A    14          LSR         ; TO BOTTOM POSITION
8005: 4A    15          LSR
8006: 4A    16          LSR
8007: 09 B0  17          ORA  #$B0    ; %1011 0000
8009: 20 ED FD 18        JSR  COUT    ; PRINT DIGIT
800C: A5 06  19          LDA  MEM     ; RETRIEVE ORIG BCD
800E: 29 0F  20          AND  #$0F    ; %0000 1111
8010: 09 B0  21          ORA  #$B0    ; %1011 0000
8012: 20 ED FD 22        JSR  COUT
8015: 60    23  DONE   RTS
```

The CLD is done at the beginning just to emphasize that the BCD mode is not required here since the digit is presumed to exist already in MEM and no arithmetic operations are anticipated. Remember that the BCD mode is required only

during the actual addition or subtraction operations. Although the BCD mode would have no harmful effect on the AND and ORA operations, COUT would certainly take offense at being called while the BCD mode was still in effect.

Lines 12 through 16 get the original BCD value from memory and then shift it left four times to move the upper nibble to the lower position. At this point the ORA #B0 is done to convert the value in the Accumulator to the proper ASCII value, at which point the JSR COUT on line 18 prints the first digit. Line 19 retrieves the original value again, after which the AND #0F clears the upper digit to 0 and the ASCII conversion is completed and printed as before.

The remainder of the routine is identical to the previous example program.

Conclusion

The Binary Coded Decimal mode of the 6502 can be convenient for a variety of reasons. Its most frequent use is to facilitate input and output, particularly for scientific instrumentation.

A number of points should be kept in mind when using the BCD mode:

1. The mode should be set only for arithmetic processes that use BCD values, such as addition and subtraction.
2. Only legal values are allowed: 0–9 for each digit. Values outside the expected range will generate inaccurate results.
3. The BCD mode should be cleared as soon as possible when arithmetic operations are completed so as to avoid possible complications with other software in the Apple that neither expects, nor checks for, the BCD mode.
4. RESET does not clear the decimal mode of the 6502.⁵ Only the CLD instruction does. You can also clear the mode by means of a CALL -155 from BASIC or an FF59G from the Monitor.
5. The N and Z-flags are unreliable as a means of detecting the results of comparisons or of increment/decrement operations. Only the carry flag should be used to detect the results of such operations.
6. The carry flag will be set for results greater than 99 (overflow) and cleared for results less than 0 (underflow).
7. BCD operations do “wrap around.” That is, $99 + 01 = 00$ and $00 - 01 = 99$.

⁵ [CT] See footnote 1 earlier in the chapter.

Special Note: Counting Down

These are some general rules to help in programs using the BCD mode of the 6502. There is only one notable exception that may on occasion prove useful. The test for 0 (BNE, BEQ) can be used when counting *down* in the BCD mode. For example:

```
SED
SEC
LDA #$01
SBC #$01
BEQ DONE
```

would work, whereas

```
SED
CLC
LDA #$99
ADC #$01
BEQ DONE
```

would not.

It might be an interesting challenge for you to use the information given in this chapter and in previous chapters to try to write a routine that would add two Applesoft strings together using the BCD mode and return the result in a third string. This would provide a way of extending the normal precision of Applesoft for mathematical operations requiring more than nine digits, a problem that unfortunately does not hinder my personal checkbook program.

Intercepting Output

February 1983

I/O routines are responsible for handling the computer's communications with the outside world. Their design is also one of the more interesting aspects of assembly-language programming. We'll spend this chapter and the next learning how to intercept the I/O vectors of the Apple and implement our own routines.

It will make the next few demonstrations much easier if you disconnect DOS from the I/O system. That's most easily done by running this short Apple-soft BASIC program:

```
10 IN#0: PR#0: END
```

That will keep DOS out of the way for the upcoming exercises.

Output

In earlier chapters we discussed how COUT (\$FDED) could be used to print characters to the screen, to disk, or to other output devices. The general procedure was to load the Accumulator with the ASCII value for the character you wanted to print and then to do a JSR COUT.

To see what happens at \$FDED when you do this, enter the Monitor by means of the usual CALL -151. Then type in: FDEDL<RETURN>.

The first instruction listed should be a JMP (\$0036). This is an indirect jump to a location pointed to by the byte pair \$36, \$37. To see where these bytes are currently pointing, type in: 36.37<RETURN>.

You should get:

```
0036- F0 FD
```

This tells you that the jump will be made to \$FDF0, which in this case happens to be the next instruction after the JMP (\$0036). \$FDF0 is called COUT1 and is used only to print characters to the Apple's screen. When output is going to the disk, to the printer, or to some other device, \$36, \$37 will point somewhere other than \$FDF0.

If you are sending characters to a printer, for example, \$36, \$37 might point to \$C102. CSW (Character output SWitch) is the name given to the byte pair \$36, \$37. A pointer such as this is usually called a *vector*, in that it directs the flow of

program control to whatever routine (that is, whatever address in memory) is appropriate at the moment.

The changing of the CSW vector is what happens when you execute a `PR#n` command. CSW is pointed to the address `Cn00`, where `n` is the slot number given in the `PR#n` statement. If no device is present in the slot, then no program will be found at `$Cn00`. This explains why a BASIC program hangs when an improper `PR#` command is given: the computer is waiting for the final RTS from a non-existent routine. To verify for yourself that the lockup doesn't occur until a character is output, run this program in Applesoft BASIC:

```
10 HOME
20 PR#5: REM OR SOME OTHER EMPTY SLOT
30 FOR I = 1 TO 20
40 POKE 1024 + I, 192 + I
50 NEXT I
60 PRINT "YOU WON'T SEE THIS"
```

When you run this program, you should see the letters A through T printed on the screen, but the phrase on line 60 should not appear. Things happen this way because the loop on lines 20 through 40 puts the data directly into the screen memory without going through COUT. Remember that all this time CSW is pointing to `$C500`. It's only when the Y character gets sent to COUT that the computer hangs.

If DOS were installed and line 20 said `PRINT CHR$(4); "PR#5"`, the program would hang on that statement because of the carriage return sent at the end of the print statement. It's instructive to note that the carriage return is not actually needed for the `PR#` to work. Adding a semicolon to the print statement would restore the program to its original semi-functional state.

One would think from the preceding thoughts that hooking up a routine to the output hooks would be fairly simple. The problem is that most of the time you'll want to have DOS active, and DOS has been cleverly designed to do everything possible to keep itself connected. When DOS is installed, CSW actually points to `$9EBD`, a portion of DOS, and it's very difficult to get it to point elsewhere.

Specifically, whenever either input or output is done, both vectors are checked to make sure DOS is still hooked up. This means that, even though you could temporarily change CSW, any input-type action would cause DOS to restore itself to the output flow. Here's a program to show this. You'll need to reconnect DOS (pressing RESET will do that) to try it:

```
10 HOME
20 PR#0
30 PRINT CHR$(4); "CATALOG"
40 INPUT "TURN THINGS BACK ON"; I$
50 PRINT CHR$(4); "CATALOG"
```

The theory here is that the PR#0 sets CSW to point directly to \$FD0C rather than to DOS. This is why the CATALOG doesn't work in line 30. However, when the input is done, DOS is still hooked up to the input vector. Realizing that the output connection has been lost, DOS thus reconnects itself. Line 50 then performs as expected.

In general, DOS can be disconnected by executing both an IN#0 and a PR#0 within a BASIC program, provided that one is done immediately after the other with no input or output in between. The one-line BASIC program used at the beginning of this chapter to disconnect DOS employs this principle.

Pressing RESET will hook things back up anytime you want. Notice that these are not done as DOS commands such as:

```
10 PRINT CHR$(4);"IN#0": PRINT CHR$(4);"PR#0"
```

An IN#0 or PR#0 as a direct BASIC command redirects I/O to the Monitor. The same commands done as DOS commands set the I/O to DOS.

Let's see just how DOS does handle the output vectors. With DOS installed and active, enter the Monitor and type in:

```
36.37 AA53.AA54
```

You should get:

```
0036- BD 9E
AA53- F0 FD
```

With DOS active, CSW points to a main output entry point at \$9EBD. This is the beginning of the section that watches the output for DOS commands. Eventually it does its own indirect jump via the vector at \$AA53, \$AA54, which completes the path to COUT1 (\$FDF0). When you do a JSR COUT (\$FDED), then, here's the general flow of things:

1. With the appropriate value in the Accumulator, a JSR COUT (\$FDED) is done.
2. At \$FDED is a JMP to the address specified in CSW (\$36, \$37). With DOS installed, CSW points to DOS at \$9EBD.
3. When DOS is through looking at the character, it does a jump to the address held at \$AA53, \$AA54. This normally points to \$FDF0.
4. Eventually an RTS returns control to the calling program.

Intercepting Output

An obvious question now arises. How do we hook our routine to DOS? This basically depends on whether a slot is used. If you happened to be writing firmware for an interface card, for example, the PR# command when executed would automatically handle the setting up of CSW to make everything work. If,

however, you want to put a routine at a location other than the \$C000 space, another approach is needed.

The procedure actually is fairly simple. All you need to do is set CSW to where you want the output to be eventually sent and then call \$3EA.¹

For example, let's put a trivial routine at \$300 that merely jumps to COUT1 (\$FDF0). Go into the Monitor and enter:

```
300: 4C F0 FD
```

If you list this routine you should get:

```
300L
```

```
0300- 4C F0 FD    JMP  $FDF0
0303- 00          BRK
0304- 00          BRK
```

To hook it up, type in the following from the immediate mode of Applesoft

```
POKE 54,0: POKE 55,3: CALL 1002
```

This sets CSW to point to \$300 and then calls \$3EA. The same thing can be done from within an assembly-language program with:

```
LDA #$00
STA $36
LDA #$03
STA $37
JSR $3EA
RTS
```

Once connected in this way, everything will still look the same on the screen. In reality, however, every character going to the screen is now going through \$300. You can check the new routing by entering the Monitor while this routine is installed and typing in:

```
36.37 AA53.AA54
```

You should get:

```
0036- BD 9E
AA53- 00 03
```

The Monitor, DOS, and BASIC all send output via the jump at COUT. This still points to DOS, but now DOS points not to COUT1 (\$FDF0), but to \$300. There, our routine does a jump to COUT1 to complete the flow.

¹ [CT] The technique would be different for ProDOS, which doesn't have a hookup routine. Instead, you can manually change the output vector at \$BE30, \$BE31 to point to your output routine. See chapter seven of *Inside the Apple II*, by Gary B. Little.

To verify that characters are going through \$300, just type in POKE 768,0. Or, from the Monitor, type: 300:0.

The computer will immediately hang as program flow hits the 00 (BRK instruction) at \$300. The BRK routine in the Monitor will then try to send the break error message through COUT, at which point \$300 will be called again and the process will repeat itself indefinitely.

An interesting point here is that when COUT is turned off (for instance, a simple RTS at \$300 will do the trick), nothing appears on the screen despite the fact that the computer is still fully functional. Even though you can't see what you're typing, you could type in CATALOG and the disk drive would come on. The flashing cursor would remain on the screen since RDKEY (part of the input routine at \$FD1B) addresses the screen directly for the cursor.

To experiment with COUT some more, let's try a routine that's a little more interesting. Control characters are normally "invisible" in that they're not sent to the screen by COUT1. If we could detect the control character before it got to COUT1 and could change it to a different value, we could have it display as inverse or as some other visible character.

Normally all characters going through CSW have the high bit set. That is, all values are greater than \$80. Inverse and flashing characters are created by sending characters with a value less than \$80 to COUT. All characters in the range of \$00 to \$3F come out inverse, and all those from \$40 to \$7F are flashing. In general what this means is that, if the high bit is cleared, control characters will come out in inverse and "standard" characters in flashing.

This is, in fact, how the FLASH and INVERSE commands of Applesoft work. The routine at COUT1 includes a portion that does an AND operation on the value about to be stored on the screen and a mask value stored at location \$32 (called INVFLG, short for "INVERSE FLA^G"). INVFLG normally holds an \$FF, so no change takes place. However, the BASIC commands INVERSE and FLASH set the values to \$3F and \$7F, respectively, which produces the desired results.

The following diagram illustrates the INVFLG mask's effect on outgoing characters sent to COUT:

	Hex	Binary	Character
Character:	\$C1	%1100 0001	A (Normal)
INVFLG:	\$FF	%1111 1111	—
AND result:	\$C1	%1100 0001	A (Normal)
Character:	\$C1	%1100 0001	A (Normal)
INVFLG:	\$7F	%0111 1111	—
AND result:	\$41	%0100 0001	A (Flashing)
Character:	\$C1	%1100 0001	A (Normal)
INVFLG:	\$3F	%0011 1111	—
AND result:	\$01	%0000 0001	A (Inverse)

We can do our own specialized processing, though, so as to highlight just control characters. Here's the listing:²

```

1 *****
2 *AL29-CONTROL CHARACTER DISPLAY*
3 *****
4 *
5          ORG  $300
6 *
7 COUT1   EQU  $FDF0
8 *
0300: C9 A0  9  ENTRY   CMP  #$A0      ; FIRST NON-CTRL CHAR
0302: B0 06 10          BCS  PRINT    ; CHAR OKAY
0304: C9 8D 11          CMP  #$8D      ; LET 'CR' THRU
0306: F0 02 12          BEQ  PRINT
0308: 29 3F 13  MASK    AND  #$3F      ; CLEAR TOP 2 BITS
030A: 4C F0 FD 14 PRINT   JMP  COUT1    ; PRINT IT

```

This routine's operation is very straightforward. A comparison is done as each character reaches the routine at \$300. All "usual" characters are sent through to COUT1 unaltered. If a character is found to be a control character, though, a test is done to see if it's a carriage return. If so, that too is passed to COUT1. After all, we do want the screen to look somewhat normal. If a control character (other than a <RETURN>) is found, however, an AND #\$3F converts the character to an inverse character, at which point it will be forwarded to COUT1.

Any control characters generated by a program, with the exception of <RETURN> (<CTRL>M), will now be shown in inverse. When typed from the keyboard, <ESCAPE>, the right-arrow key (<CTRL>U), and <CTRL>X won't show up since they are intercepted by the Monitor input routine and never make it to COUT.

Other Output Devices

So far, all we've done is intercept COUT, filter the characters going through, and eventually return control to the Monitor screen routine COUT1. If we had our own output device, this would not be necessary. The point here is to demonstrate the possibility of alternate output devices. Ultimately this could include printer cards, terminals, analog devices such as motors, and more. Such projects are rather involved, however, so for now let's just see if we can write our own primitive screen routine.

² [CT] When running under DOS 3.3, you can hook up your routine by executing:

```
POKE 54,0: POKE 55,3: CALL 1002
```

Under ProDOS, you can directly modify the output vector at \$BE30, \$BE31 by running the following Applesoft program (do not run this as a direct command):

```
10 POKE 48688,0: POKE 48689,3
```

The basic model will be to set aside one line of the screen as our display window and to attempt to control text output within that window. To avoid having to create vertical scrolling routines and cursor management routines, we'll limit all output to the single line and scroll text only to the left as each new character is displayed on the right.

If this sounds suspiciously similar to a calculator display, you're right. It should be easy now to see why, with limited resources of display hardware and, more significantly, limited memory for management routines, such a display would be desirable.

Here's the summary of the design points:

1. Display will be limited to one line.
2. Characters will be output on the rightmost position.
3. The remainder of the line will scroll to the left to make room for each new character.
4. No control characters will be displayed.
5. The left-arrow key, <CTRL>H, will be designated as a "clear display" character.
6. No editing capabilities (that is, backspace, forward copy, and so on) will be provided for, except for number 5 above.

Before proceeding, let's digress for a moment to mention the value of the list as a programming technique. If you can't bring yourself to flowchart, at least make a list to clarify exactly what your program will do. This helps to organize your thoughts in a general way before you have to leap in and code the detailed parts. Even if you amend it as the coding progresses, such a list is helpful. Now back to our regularly scheduled program...

```

1 *****
2 * AL29-SPECIAL DISPLAY ROUTINE *
3 *****
4 *
5 *          OBJ  $300
6          ORG  $300
7 *
8 LINE     EQU  $700      ; $700-727
9 YSAV1    EQU  $35
10 *
0300: 84 35 11 ENTRY   STY  YSAV1      ; SAVE Y-REGISTER
0302: C9 A0 12         CMP  #$A0      ; FIRST NON-CTRL CHAR
0304: B0 11 13         BCS  SCROLL    ; DISPLAY THE CHAR
0306: C9 88 14 CHK     CMP  #$88      ; BACKSPACE
0308: D0 0A 15         BNE  DONE1
030A: A0 27 16 CLEAR   LDY  #$27
030C: A9 A0 17         LDA  #$A0      ; SPACE
030E: 99 00 07 18 LOOP1  STA  LINE,Y    ; ERASE A CHAR

```

```

0311: 88      19      DEY
0312: 10 FA   20      BPL LOOP1      ; UNTIL Y=$FF
0314: A4 35   21  DONE1  LDY YSAV1      ; RESTORE Y
0316: 60      22  OUT1   RTS          ; DON'T SHOW
                23  *
0317: 48      24  SCROLL PHA          ; SAVE THE CHAR
0318: A0 01   25      LDY #$01
031A: B9 00 07 26  LOOP2  LDA LINE,Y
031D: 99 FF 06 27      STA LINE-1,Y
0320: C8      28      INY
0321: C0 28   29      CPY #$28
0323: 90 F5   30      BCC LOOP2      ; UNTIL Y=$28
0325: 68      31  PRINT  PLA          ; RETRIEVE CHAR
0326: 8D 27 07 32      STA LINE+$27
0329: A4 35   33  DONE2  LDY YSAV1      ; RESTORE Y
032B: 60      34  OUT2   RTS
032C: 27      35      CHK

```

After the listing has been assembled, the routine is hooked up to COUT, just like the other routine. You will probably want to type in HOME to give you a clear screen for your display. Once your routine is installed, everything you type should scroll across a line in the upper half of the screen. Notice that all expected output from the Apple is now done on its own custom display. You can list programs, catalog a disk, or do any of the usual operations. Try typing in this command line in Applesoft:

```
FOR I = 1 TO 127: PRINT CHR$(I);: NEXT I
```

When you press return, you should see a whole series of characters go whizzing through the window, ending with the lowercase letters (although they may not look quite right if you don't have a lowercase display device). Remember, the left arrow will clear the display window.

The routine itself is fairly simple. The only memory locations defined are the memory range for the screen line at \$700, a temporary storage byte used by COUT1, and our routine to preserve the contents of the Y-Register. The program also contains some instructive points of style.

On entry, the Y-Register is saved. This is because the "official" output routine, COUT1, returns with all registers (A, X, and Y) intact when called. Many other routines in BASIC and DOS assume that all output will be done as safely, so we must honor that convention as well.

Once Y is saved, the value passed to this routine from the Accumulator is appropriate to the ASCII value for the character to be printed. As was done in the control-character display routine, a check is done for control characters. Remember that in this program all control characters, even <RETURN>, will be filtered out. If a control character is detected, the comparison on line 12 will fail and a check will be made for the left arrow (<CTRL>H). If the character is not a

<CTRL>H, we will immediately exit via DONE1, where the Y-Register will be restored and no character will be displayed.

If a <CTRL>H is detected, the CLEAR routine clears the display window to spaces. A note here about the BPL on line 20 to determine when the loop is done: You might think that we would want to use a BNE to find out when Y reached 0. The problem is that, when Y reached 0, the branch would fall through and we would not store a space at \$700, so the leading character could not be cleared from the display window.

Because we know that Y is started at \$27, we can test for Y reaching the value of \$FF as it “wraps around” after reaching 0. An alternate approach would have been to make line 18 say STA LINE-1, Y and to start Y with a value of \$28 on line 16. LINE-1 would evaluate to \$6FF, and thus we could use the BNE test. Either way works, but this second approach provides a way of showing another programming technique. After clearing the window, the routine returns via DONE1, again without displaying any new character.

If a legitimate character is detected on lines 12 and 13, control flows to SCROLL, which makes room for the new character to be displayed. Because we’ll need to use the Accumulator for the scrolling, the character to be printed is pushed onto the stack to save it for future use.

At that point, the Y-Register is set to \$01 in preparation for the memory move to follow. Line 26 loads a character from one position, after which line 27 will store the character in the position immediately to the left. For example, on the first pass through, the value will be loaded from \$701 (\$700, Y where Y = 1) and stored at \$700 (\$6FF, Y where Y still equals 1).

Notice the use of two different base addresses for the indexed addressing. This allows us to use the same value in the Y-Register to load and store at two different addresses. The loop is repeated until we have moved all the characters one position to the left. The routine then falls into PRINT.

PRINT first retrieves the character to be printed from the stack by means of the PLA on line 31. It then stores the character at \$727. The code is written this way (LINE + \$27) to show that you can, in most assemblers, add any amount to an address. You aren’t limited to the usual ADDR, ADDR+1 that’s most often seen. After the character has been stored at \$727, the Y-Register is restored and the routine returns via DONE2.

You should verify for yourself that the Accumulator and Y-Registers are always left in their original conditions regardless of whether the RTS is done through DONE1 or DONE2. Since we didn’t use the X-Register, it also will be preserved.

Conclusion

Here are the main points of our discussion on the output vector.

1. The main output vector is called CSW, which stands for Character output SWitch. CSW is the byte pair \$36, \$37.
2. DOS maintains its own output vector at \$AA53, \$AA54.
3. DOS can be disconnected by executing the BASIC statement IN#0: PR#0 (not as a DOS command).
4. DOS can be reconnected by pressing RESET.
5. Any attempt to alter CSW directly with DOS active will be undone by DOS on the first input statement following the attempt.
6. To hook a routine into the output vectors, execute the equivalent of

```
POKE 54, LB: POKE 55, HB: CALL 1002
```

where LB and HB are the low- and high-order bytes of the address you wish output to be directed to.³

7. If you're handling all of the final output, end the routine with the usual RTS. If you're merely filtering or watching the output, you must eventually pass control on to where the final output will be done, usually COUT1 (\$FDF0).

In the next chapter we'll look at the input hooks and at how to use your own routines on the listening side of the Apple.

³ [CT] The equivalent for ProDOS would be:

```
10 POKE 48688, LB: POKE 48689, HB
```

Intercepting Input

March 1983

It's time to examine the input system of the Apple. Many parallels can be drawn between it and the output system, discussed in the previous chapter. Though not required, some familiarity with that chapter's major points will help you understand our current topic.

The main demo routines in this installment involve lowercase text; therefore, it's strongly recommended that you acquire lowercase display hardware if you don't have it already. Lowercase chips for Apples with revision numbers greater than 7 can be purchased for \$20 to \$30. Earlier Apples require more than a single chip. Apple //e doesn't require any additional software or hardware; the lowercase display capability is built in. For serious study and exploration of text input/output methods, lowercase capability is essentially required.

The Input Vector: KSW

The byte pair \$38, \$39 constitutes the main input vector and is generally labeled KSW for Keyboard input SWitch. Like CSW (the Character output SWitch), KSW is used to switch input to BASIC and the Monitor from different sources. As is evident from the fact that an INPUT statement will read a DOS text file and the action of the EXEC command on text files, the keyboard isn't the only place from which the Apple can obtain ASCII data.

When you're writing an assembly-language program that needs a single-character input from the outside world, the usual procedure is to do a JSR RDKEY (\$FD0C) and then use the value that is returned in the Accumulator.

As we did with COUT (\$FD0E), let's see what RDKEY does to get that character. To examine the routine, enter the Monitor with the usual CALL -151 and list the code by typing \$FD0C<RETURN>.

Here, shown with labels and comments, is the code at that location:¹

```

FD0C-  A4 24      RDKEY  LDY  CH          ; Get horizontal cursor
FD0E-  B1 28      LDA   (BASL),Y      ; Get character from screen
FD10-  48                PHA                ; Store it
FD11-  29 3F      AND   #$3F          ; Clear bits 6,7
FD13-  09 40      ORA   #$40          ; Set bit 6 (flash)
FD15-  91 28      STA   (BASL),Y      ; Put on screen

```

¹ [CT] The code shown is for an Apple II or Apple II Plus. The code for an Apple //e is quite a bit different, but the entry points at RDKEY and KEYIN are the same.

```

FD17- 68          PLA          ; Get the original character
FD18- 6C 38 00    JMP   (KSW)    ; To 'real' input
FD1B- E6 4E      KEYIN  INC   RND      ; RND = RND + 1
FD1D- D0 02      BNE   KEYIN2
FD1F- E6 4F      INC   RND+1
FD21- 2C 00 C0   KEYIN2 BIT   KBD      ; Check for key
FD24- 10 F5      BPL   KEYIN    ; No, again
FD26- 91 28      STA   (BASL),Y ; Restore old character
FD28- AD 00 C0   LDA   KBD      ; Get input character
FD2B- 2C 10 C0   BIT   KBDSTRB ; Clear strobe
FD2E- 60          RTS          ; Return with character

```

On entry to RDKEY the first three instructions read the character on the Apple screen and put it onto the stack. Remember that what you see on-screen is the representation of a byte stored in the memory range of \$400 to \$7FF. To determine what byte corresponds to a screen position, you need only load the Y-Register with the horizontal cursor position (CH = \$24) and add this offset to the base address for the current line. This base address is always stored in \$28, \$29 (BASL, BASH).

Once the existing character on-screen has been read and stored (so we can put it back on-screen after the input), the next three instructions have the net effect of putting a flashing character on the screen equivalent to the character that was on-screen in the current cursor position.

The action of the ANDs and ORAs may not be intuitively obvious. Let's consider this example:

	Hex	Binary	Character
Original character:	\$C1	%1100 0001	A (Normal)
AND:	\$3F	%0011 1111	clear bits 6, 7
First result:	\$01	%0000 0001	A (Inverse)
ORA:	\$40	%0100 0000	set bit 6
Final result:	\$41	%0100 0001	A (Flashing)

Remember that the action of the AND is to clear any bits in the Accumulator that are matched by a 0 in the mask value. Bits in the Accumulator matched by 1s in the mask are left unchanged, whether they are 0s or 1s.

An ORA, on the other hand, sets to 1 any bits in the Accumulator that are matched by a 1 in the mask value. Bits in the Accumulator matched by 0s in the mask are left unchanged.

You might wonder at first why two instructions—the AND followed by the ORA—were needed. After all, in the previous chapter didn't we change control characters to inverse in just one step? Why not just use a different mask value to get flashing characters? The answer lies in the differences between the bit patterns for inverse and flashing characters. All inverse characters have the top two

bits clear (bits 6 and 7). Flashing characters, on the other hand, have one bit clear (bit 7 = 0) and the other set (bit 6 = 1).

When the cursor is on a character and the character is to be converted to flashing temporarily, we must not only clear the high bit (at least for all “normal” text), but also must on occasion set bit 6. This combination of a set and a clear requires two operations.

Once RDKEY has thus put a flashing character on-screen to show the cursor’s location, the character originally on the screen is retrieved from the stack in preparation for the jump to KEYIN (or to any other input routine that will want to restore the original character if no new character is entered). Finally, the actual indirect jump via KSW is done.

In COUT (\$FDED), the jump via CSW was made immediately. This extra portion in RDKEY preceding the actual jump explains the presence of the cursor on-screen during a text-file read. Although DOS is handling the input at that point, the call is still done via RDKEY, and thus the presence of the cursor is still somewhat unavoidable.

If DOS is not active, KSW ordinarily points to KEYIN (\$FD1B). KEYIN is the routine responsible for getting characters from the keyboard; it thus involves the keyboard memory hardware (\$C000 and \$C010) directly. If input was from a modem or some other external device installed in a peripheral slot, KSW would point to \$Cnxx, where n is the slot number and xx is the input routine entry point. Before considering the unusual situations, let’s see what happens most of the time, when KSW points to KEYIN.

KEYIN first increments the random-number byte pair, \$4E, \$4F. This is a part of the loop that will be repeated until a key is pressed. The theory is that the passage of time between key presses is random. This byte pair is used primarily by Integer BASIC. Applesoft has its own random-number registers and routines.

After incrementing the random byte pair, KEYIN2 then does the actual keyboard check, repeating the process by going back to KEYIN if no key has been pressed. Remember that the BIT instruction makes the test possible by setting the sign flag of the Status Register equal to bit 7 of the character value detected at the keyboard (\$C000). BPL thus can be used to detect (by failing) when bit 7 goes high (bit 7 = 1), indicating a keypress.

Once a key has been pressed, the value in the Accumulator is put back into screen memory. Remember that this is the value of the old character presumably there, *not* the new character input. If the character entered is a right arrow, this signifies that we want to move the cursor over the displayed character without changing that character. The LDA KBD is what puts the input character into the Accumulator, at which point the strobe is cleared by accessing \$C010 and the final return is done. The calling program then has the option of printing the input character to the screen.

Other Input Sources

KSW does not always point to RDKEY. In fact, it doesn't point there when DOS is installed. With DOS booted and active, enter the Monitor and type in:

```
38.39 AA55.AA56<RETURN>
```

You should get:

```
0036- 81 9E
AA55- 1B FD
```

You'll see that KSW actually points to DOS at \$9E81, which then eventually points to RDKEY (\$FD1B) at \$AA55, \$AA56. Like the output system, DOS is rather permanently made part of the input path. Any attempts to disconnect DOS by modifying KSW directly will be undone by DOS if any output is done. DOS has its own internal input vector at \$AA55, \$AA56. It alters this vector, not KSW, as needed to gain access to various slots (or to disk files, as appropriate).

You can install your own routine into the input path by means of a procedure similar to the one used in the previous chapter to intercept the output path. Put the low- and high-order bytes of the destination address into KSW (\$38, \$39 = 56, 57 decimal) and do a call to \$3EA (1002 decimal). This causes DOS to change its own vectors at \$AA55, \$AA56 to the address specified, and then to restore KSW so that it points to DOS again, usually at \$9E81.²

In Applesoft this would take the form:

```
10 POKE 56, LB: POKE 57, HB: CALL 1002
```

In this example, LB and HB are the low- and high-order bytes of the destination address. In assembly language, it would look like this:

```
LDA #LB
STA $38
LDA #HB
STA $39
JSR $3EA
```

Just as output has two basic classes of routines, there are two main types of input routines—those that intercept incoming characters and do some sort of processing, and those that entirely replace the input routines already being used. If you are doing the latter, things are fairly simple. Once installed, your routine is entirely in charge of getting the input character; when that character is “got,” your routine ends with an RTS to pass control back to the calling program. This approach is similar to our custom output routines from the previous chapter.

² [CT] Just like the output vector (described in chapter 29), when using ProDOS you can directly change the input vector at \$BE32, \$BE33 to point to your input routine. See chapter six of *Inside the Apple IIe*, by Gary B. Little.

The first class of input routines, in which incoming characters are to be intercepted, must be handled slightly differently than our output experiments were.

Interception Routines

When we were dealing with the output process, the point at which we intercepted the data flow really didn't matter. Because the calling program loads the Accumulator with the character to be output, the character can be examined at any point along the way. With input, the character input is not available until the very end of the procedure, when the RTS returns control to the calling program. Fortunately, there is a relatively easy way around this limitation.

In both the input and output systems, the links in the process are done by means of a series of Jumps (as opposed to JSRs). You'll recall from our output interception from the previous chapter that the final exit from the routine was a JMP \$FDF0 (or wherever) after the processing was done.

With input, the secret is to do a JSR to KEYIN (or wherever) first and then do your processing, followed by an eventual RTS to the calling program. For our first experiment, we'll try writing a routine to convert all incoming characters to lowercase:

```

1 *****
2 * AL30-SIMPLE CASE CONVERTER *
3 *****
4 *
5 *      OBJ  $300
6 *      ORG  $300
7 *
8 KEYIN  EQU  $FD1B
9 *
0300: 20 1B FD 10 ENTRY  JSR  KEYIN
0303: C9 C1 11          CMP  #$C1      ; ASCII 'A'
0305: 90 02 12          BCC  DONE
0307: 09 20 13 MASK    ORA  #$20      ; %0010 0000
0309: 60 14 DONE     RTS

```

In theory, anything you type in now should be displayed in lowercase. Numeric and control characters should be unaffected. The routine works by first calling KEYIN, which gets a character from the keyboard and puts it in the Accumulator. At that point our routine ensures that we've got a capital letter, rather than a numeric or control character. If we don't have an alphabetic character value less than \$C1, then the routine skips to DONE.

If what we have is an alphabetic character, the conversion to lowercase is done by forcing bit 5 of the ASCII value to 1. The values of all lowercase characters are equal to the values of the corresponding uppercase letters plus 32. This means, as an ASCII chart showing bit values reveals, that capital letters have bit 5

clear and lowercase letters have bit 5 set. Line 13 of our routine sets bit 5, thus converting the character to lowercase. Finally, line 14 returns us to the calling program.

Our routine should work from within Applesoft. Try this:³

```

6 INPUT "ENTER A STRING:"
10 POKE 56,0: POKE 57,3: CALL 1002
20 INPUT I$
30 PRINT I$
40 PRINT CHR$(4); "IN#0": REM DISCONNECT ROUTINE

```

Don't be surprised if this program doesn't work.⁴

Try changing line 20 to look like this:

```

20 GET A$: PRINT A$;: IF A$ <> CHR$(13) THEN I$ = I$ + A$: GOTO 20

```

Now run the program. The results this time should be more like you expected. Line 30 is used to confirm the fact that the lowercase data we typed in on line 20 actually made it to Applesoft.

The question now is, why didn't the first program work? In a sense it did. If you like, go back and run the first program without line 40. When the program ends, go into the Monitor and check the DOS input vector at \$AA55, \$AA56. It should indicate that our routine at \$300 is being used.

The problem lies in Applesoft's use of the GETLN (GET LiNe) routine for the INPUT statement. This routine is used to input entire lines at a time. Although GETLN does use the RDKEY routine to get individual characters, it unfortunately tampers with the characters entered before it returns the data to Applesoft, DOS, or the Monitor.

Specifically, GETLN converts any lowercase characters coming in to uppercase. Thus, even though our routine converts the uppercase characters coming in through the keyboard to lowercase, GETLN undoes every thing by converting them back before they're even echoed to the screen.

Another annoyance of GETLN is that it converts characters that you copy from the screen using the right arrow.

The reason the program works with the new version of line 20 is that the Applesoft GET statement uses a direct call to RDKEY and does not use GETLN.

One way to solve the problem of the INPUT statement not working is by writing your own input routine instead of using the GET sequence. The easiest thing to do here would probably be to copy the GETLN routine and eliminate the conversion portion starting at \$FD7E.⁵

³ [CT] Under ProDOS, you should change line 10 to POKE 48690,0: POKE 48691,3

⁴ [CT] On an Apple //e (under DOS) this program actually will work.

⁵ [CT] In the Apple II and Apple II Plus the code at \$FD7E checks whether the character is \geq \$E0 and if so, does an AND #\$DF, which converts from lowercase to uppercase. In the Apple //e this has been replaced with AND #\$FF, which does nothing.

Instead, let's see if we can improve on the simple input routine just shown, making it a little more flexible, without rewriting the GETLN routine.

Something More Useful: Lowercase Input

Although the routine just given illustrates the concept of intercepting input, it's not really that useful because it provides no way of switching between uppercase and lowercase letters at will. Why not create an input routine that allows us to shift between uppercase and lowercase letters as we input them? As we did for the output routine in the previous chapter, we'll first make a list of what we want the routine to do:

1. The routine should allow numeric and control characters to pass through unaltered.
2. The routine should be set up such that pressing <ESCAPE> once when in the lowercase mode will shift only the next letter to uppercase.
3. Pressing <ESCAPE> twice when in the lowercase mode should shift all successive input to the uppercase mode (this is sometimes called "caps lock").
4. Pressing <ESCAPE> once when in the uppercase mode should return the system to the lowercase mode.

The system of using <ESCAPE> as a shift key is somewhat standard. Before going on to the listing, though, let's think a little more about what is needed to implement this system. First off, we'll need some way to remember which mode (lowercase or uppercase) we're in. The most direct way of doing this is to use a flag, which we'll call CSFLG (CaSe FLaG). To avoid a zero-page conflict, we'll reserve a place for the flag at the end of the routine.

In order to fulfill the requirement stated in item three on our list, we need to store the value of the last character input—that is, the character just before the one currently being input, in another storage location. This will allow us to tell when <ESCAPE> has been hit twice in a row. We'll call this location LSTCHR (LaST CHaRacter).

The general pattern will be to do some brief tests each time a character is input and, if no conversion is necessary, to pass the uppercase letter through unaltered. Only when an <ESCAPE> sequence is coming through or when we're in the lowercase mode will we ever alter the input character. Here, then, is the improved listing:

```

1 *****
2 * AL30-LOWERCASE INPUT ROUTINE *
3 *****
4 *
5 *          OBJ  $300
6 *          ORG  $300

```

```

      7 *
      8 KEYIN EQU $FD1B
      9 ESC EQU $9B
     10 *
0300: 20 1B FD 11 ENTRY JSR KEYIN ; GET KEY
0303: 48 12 PHA ; SAVE CHAR
0304: C9 9B 13 CMP #ESC
0306: F0 19 14 BEQ ESC1
     15 *
0308: AD 3F 03 16 CHAR LDA LSTCHR
030B: C9 9B 17 CMP #ESC
030D: F0 0D 18 BEQ XFER ; CAP THIS CHAR
     19 *
030F: 2C 40 03 20 CASE BIT CSFLG
0312: 30 08 21 BMI XFER ; CAPS
     22 *
0314: 68 23 CVERT PLA ; RETRIEVE CHAR
0315: C9 C1 24 CMP #$C1 ; ASCII 'A'
0317: 90 02 25 BCC X2 ; DON'T CHANGE
0319: 09 20 26 ORA #$20 ; SET BIT 5
031B: 48 27 X2 PHA ; PUT CHAR BACK
     28 *
031C: 68 29 XFER PLA ; RETRIEVE CHAR
031D: 8D 3F 03 30 STA LSTCHR ; LSTCHR = CHR
     31 *
0320: 60 32 DONE RTS
     33 *
0321: AD 3F 03 34 ESC1 LDA LSTCHR
0324: C9 9B 35 CMP #ESC
0326: D0 10 36 BNE CASE2
     37 *
0328: A9 80 38 LOCK LDA #$80 ; BIT 7 = 1
032A: 8D 40 03 39 STA CSFLG ; UC
032D: D0 ED 40 BNE XFER ; ALWAYS
     41 *
032F: 68 42 UNLOCK PLA ; PULL CHAR
0330: A9 00 43 LDA #$00
0332: 48 44 PHA ; CHR = NULL
0333: 8D 40 03 45 STA CSFLG ; 0 = LC
0336: F0 E4 46 BEQ XFER ; ALWAYS
     47 *
0338: 2C 40 03 48 CASE2 BIT CSFLG
033B: 10 DF 49 BPL XFER ; LC NEEDS NO ACTION
033D: 30 F0 50 BMI UNLOCK ; UNLOCK UC
     51 *
033F: 00 52 LSTCHR DFB $00
0340: 00 53 CSFLG DFB $00 ; DEF = LC; #$80 = UC
     54 *
0341: 9C 55 CHK

```

After assembling and installing this routine at \$300, try the Applesoft program with the altered line 20 again. This time you should be able to enter a string containing both uppercase and lowercase letters, with the <ESCAPE> key functioning as described in the requirements list.

Note the use of EQU to define ESC in line 9. The label ESC is used as a value rather than a location. This way you can change the key used for shift by changing the value equated in line 9.

A look at the source listing reveals what's going on. First, a JSR KEYIN is done to get a character from the keyboard. KEYIN handles the flashing cursor and keyboard hardware for us. Next, the input character is pushed on the stack so we'll be free to use the Accumulator if necessary without losing the input character.

Next, a test is done to see whether the current character is an <ESCAPE> character. If so, a branch is done to the <ESCAPE>-handling routine, ESC1 (line 34). The first thing done at ESC1 is to see if the last character was an <ESCAPE> as well, in which case LOCK (line 38) sets caps-lock mode by putting a \$80 in CSFLG. If not, then CASE2 (line 48) checks CSFLG to see whether we're currently in lowercase or uppercase.

To simplify this test, we've used a value of \$00 for CSFLG to signify the lowercase mode. A value of \$80 signifies the uppercase mode in our example. These values were chosen to allow the use of the BIT command. Because the BIT instruction conditions the sign flag (bit 7) of the Status Register according to bit 7 of the memory location referenced, we can test the status of CSFLG without actually having to load the Accumulator with anything to do the test.

CASE2 uses the BIT instruction to test bit 7 of CSFLG. If bit 7 is clear, we're in lowercase mode and all that needs to be done is to pass this first <ESCAPE> character through to XFER, where it will be stored in LSTCHR. That way the <ESCAPE> can be used to signify a shift to uppercase if the next character is a letter.

If bit 7 is set, then we're in uppercase, and we need to "unlock" the uppercase mode. UNLOCK does this by putting a 0 value in CSFLG. You'll also notice that the current character is changed from an <ESCAPE> to a null. This is done so that after down-shifting, we can still press <ESCAPE> once more to capitalize the next letter. If we hadn't changed that <ESCAPE> to a null when we down-shifted, we'd be back in caps-lock mode.

For the next pass through, let's see what happens with a non-<ESCAPE> character. We'll resume tracing the routine right after ENTRY has decided that the current character is not an <ESCAPE> character.

The next section is CHAR, which checks to see whether the last character through was an <ESCAPE> character. If so, we need to make sure the current letter is capitalized, even though we're presumably in the lowercase mode. This is easily done, though: program flow proceeds directly to XFER. Remember, XFER simply stores the current input character in LSTCHR and then returns to the calling program. In this case, because all characters generated by KEYIN are always uppercase (except on the Apple //e), we'll just leave the capital letter input "as is" and pass it through.

If the last character was not an <ESCAPE>, program flow continues to the CASE section, which decides whether to convert the character coming through by checking to see whether we're in uppercase or lowercase mode.

CASE uses the BIT instruction to do this test. If we're in the uppercase mode (bit 7 = 1, therefore BMI works), no conversion of the incoming uppercase letter is needed and the program branches directly to the XFER routine. XFER retrieves the original input character stored on the stack, updates LSTCHR (since this will now be the "last character" on the next pass through), and then returns to the main calling program via the RTS.

If the CSFLG was set to 0, line 21 would not branch, and the CVERT (CONVERT) routine would be entered. CVERT first retrieves the input character from the stack and then checks to see if the character has an ASCII value less than that of the letter A. If so, the character coming through is a number or a control character and, as such, should not be converted to lowercase. If such a character is detected, the routine jumps over the conversion routine to line 27, which puts the character back on the stack (where XFER expects to find it) and goes through to the XFER section.

If the character has an ASCII value equal to or greater than that of the letter A, then the ORA #\$20 sets bit 5, thus converting the letter to lowercase. At that point the new character is put on the stack for the XFER routine.

Conclusion

This is definitely one of those programs that take a flow chart to design, so don't feel discouraged if everything's not immediately clear. Considering all the possible situations of <ESCAPE> sequences and current case, it may take a little time before you feel comfortable with it.

Even if the program never makes complete sense, remember that the important thing here is to understand the workings of the input system in general, rather than this particular little routine.

Of course, the best way to understand what's going on is to experiment with your own routines. Doing this always helps bring out the right and wrong assumptions about the way we think things work. You might want to try writing the generalized input routine suggested earlier, or perhaps you're one of those people who've hooked up a wire from the <SHIFT> key to pushbutton 2. If so, see whether you can improve the input routine to allow yourself to use the <SHIFT> key as well. Another interesting project would be to write your own KEYIN routine to be used by the input routine, then see if you can generate a different kind of cursor—or solve the problem of the cursor not looking quite right when it's on a lowercase letter.

Hi-Res Character Generator

April 1983

This chapter starts a discussion about how to write your own hi-res character generator, and thus how to use text on the hi-res screen in your own assembly-language programs.

The discussion will cover a number of points. First, we'll look at the memory mapping of the hi-res screen to see what considerations must be made to put the data for the appropriate characters on the screen.

Next, we'll look at the code needed to intercept the characters being output to the normal text screen, and how this information can be used to actually implement the hi-res character generator.

Last of all, a listing for a character editor will be presented, so you can make up your own character sets or even produce special characters for unusual graphics effects.

Text and Hi-Res Screen Mapping

The first consideration in creating our character generator is the topic of what actually will be required to put a character on the hi-res screen. In previous chapters we have seen how each dot on the graphics screen is related to an individual bit within a byte of memory assigned to the hi-res display. In earlier routines we created graphics by plotting dots using the routines built into Applesoft BASIC. This time the approach will be somewhat different.

To create a character on the hi-res screen, an entire array of dots will have to be turned on. Although the HLOT routines of Applesoft could be used, it turns out there is a much simpler way to achieve the desired result. This method is based on similarities between the normal text display page and the hi-res graphics display page. To fully understand this technique, though, a brief overview of the screen memory mapping will be required.

On the Apple, text display is normally confined to what is called text display page 1. This display corresponds to a block of memory in the address range \$400 to \$7FF (1024 to 2047 decimal). A character is printed on the screen by storing a single byte in this memory range. The computer hardware then takes care of converting this stored character into a video image on your monitor or television set.

Line #	Address (hex)	Address (dec)
0	400	1024
1	480	1152
2	500	1280
3	580	1408
4	600	1536
5	680	1664
6	700	1792
7	780	1920
8	428	1064
9	4A8	1192
10	528	1320
11	5A8	1448
12	628	1576
13	6A8	1704
14	728	1832
15	7A8	1960
16	450	1104
17	4D0	1232
18	550	1360
19	5D0	1488
20	650	1616
21	6D0	1744
22	750	1872
23	7D0	2000

The memory for the screen display is not mapped in a simple, continuous pattern. That is to say, if you were to fill memory sequentially with a certain value, the screen image would not be changed in a line-by-line, character-by-character pattern. Instead, a rather unusual pattern would be followed. The table at left gives the address of the first character on each line of the normal text display page. You may also wish to look at page 16 of the *Apple II Reference Manual* for a more complete chart.

You may recall from earlier chapters that it was not necessary to calculate the beginning address (sometimes called the *base address*) of each line ourselves. Instead, we can use a Monitor routine called VTAB (\$FC22).

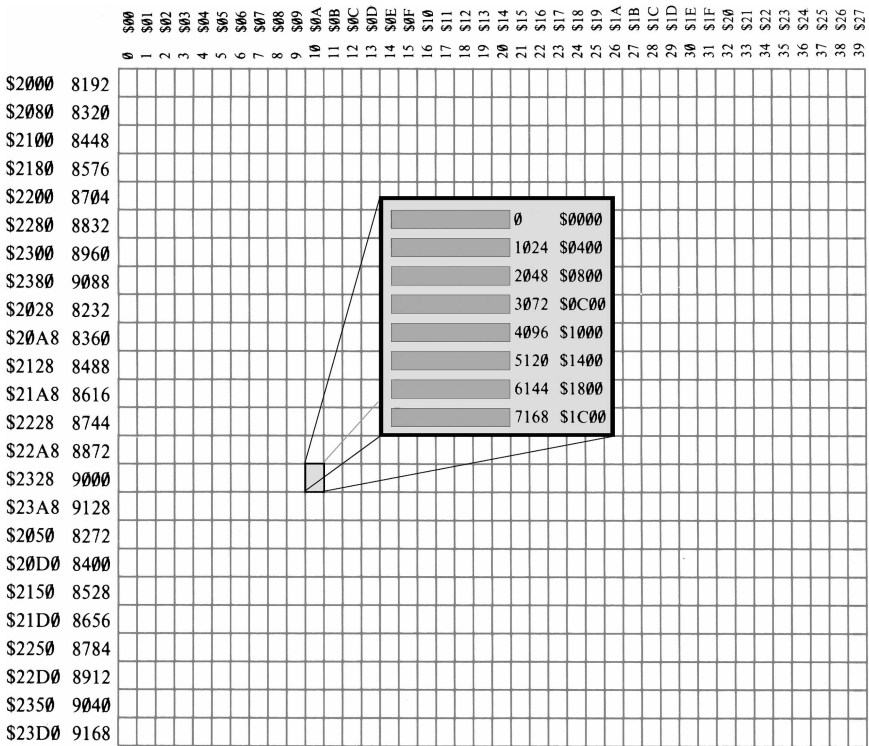
When this routine is called, it takes the value stored in location \$24 (called CV for Cursor Vertical position) and calculates the base address of the line corresponding to that vertical position. CV is assumed to be in the range of \$0 to \$17 (0 to 23 decimal) when VTAB is called.

This is what COUT (\$FDED) does whenever the cursor moves to a new line, such as when <RETURN> is pressed, or when a VTAB command is done in BASIC. The base address is returned in a zero-page pointer called BASL, BASH (\$28, \$29 = Base Address Low byte and High byte).

At first glance, there may seem to be too few horizontal rows to represent all 192 lines. However, if you look at the figure on the next page, in the middle you'll see a blow-up of one box of the map. Each of eight lines within the box is labeled with one of eight values. What this means is that each box on the main chart actually represents eight screen lines on the display. Twenty-four boxes times eight lines in each box gives us the total of 192 screen lines. To find the base address of the third screen line, for instance, you would add the correction for the third line within a box (\$800) to the base address for the primary box (\$2000) to get the actual base address (\$2800).

Looking at the horizontal rows, you'll notice that there are 40 bytes that make up the 280 horizontal dot positions. Seven bits in each byte are used to map the screen dots ($7 \times 40 = 280$).

At this point you may be getting discouraged thinking that a lot of complicated calculations are going to be required to even begin to know where to start drawing our character on-screen. Take heart, though! If you give it a little



thought, you should be able to see a remarkable similarity between the hi-res page and the text page in regard to their memory mapping.

The first similarity is in the number of bytes used for each horizontal line on the screen. In each case, 40 bytes are used for an entire line. Could there be even more similarity? Read on!

If you look at the first four lines of the text page, the base addresses are the values \$400, \$480, \$500, and \$580. If you examine the first four blocks of eight lines each on the hi-res screen, the base addresses are \$2000, \$2080, \$2100, and \$2180.

You'll notice that if you add the value \$1C00 to each of the text-screen values, you'll get the corresponding base address for the hi-res screen. This pattern continues throughout all twenty-four text screen lines.

What about the eight lines for each block? Each successive line within a block can be calculated by adding the value \$400 to the address for the line above it. This will turn out to be just perfect for creating a character.

As it happens, a character on the normal text screen is made up of dots in a matrix seven dots high by five dots wide. Around this matrix there is a boundary

of one dot position on either side and one dot position along the bottom. This permanently empty region is set up to provide a guaranteed separation between characters when printed on-screen. Thus, the final matrix is actually seven dots wide by eight dots high. The figure below, for example, shows the matrix pattern for the letter A.

A column of dot positions on each side of the character and a row on the bottom are left open. At this point, a little light in your mind is probably starting to glow. The seven dot positions across each character can correspond to seven bits in each of the 40 hi-res screen bytes used on each line. The eight horizontal rows will correspond to the eight bytes assigned to each primary box described earlier.




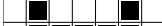




All this, then, brings us to the precipice. It is time to make the mental leap to understanding the concept of how a hi-res character can be created.

In a block of eight sequential bytes of memory, we can store all of the information needed to create a single character on the screen. Each byte will correspond to one of the eight rows in the matrix. Each bit within a byte will correspond to one possible dot position within a given row.

For example, to encode the letter A, we might store the following bytes: \$08, \$14, \$22, \$22, \$3E, \$22, \$22, \$00.

To illustrate how this really forms the letter A, take a look at the table to the right of the figure, which shows these same numbers in a different way.

In the right-hand column is the binary form of each number. You can see which bits are on and which are off. This relates directly to how the character is displayed on-screen. The bits are plotted in reverse order—that is, with bit 0 in the leftmost position. Bit 7 (the high bit) is never displayed on-screen. At most, bit 7 can be used only to shift the other dots one-half position. See the earlier chapters on hi-res plotting if you need a little refresher in this area.

Dot Matrix for A	Hex	Binary
	\$08	%0000 1000
	\$14	%0001 0100
	\$22	%0010 0010
	\$22	%0010 0010
	\$3E	%0011 1110
	\$22	%0010 0010
	\$22	%0010 0010
	\$00	%0000 0000

The Character Generator

Now to actually describe the character generator that will be used to put the appropriate ASCII character on the hi-res screen.

The process it will use is as follows:

1. A routine will be hooked up to the output vector to intercept each character to be printed to the normal text screen.
2. If the character is a control character, no special processing will be done and the character will be passed on to COUT1 (\$FDF0).

3. If the character is not a control character, an examination of CV (\$23 = Cursor Vertical position) and the current text-page address will be made. A value of \$1C00 will be added to BASL, BASH (\$28, \$29) to calculate the base address of the primary hi-res screen line. The contents of CH (\$22 = Cursor Horizontal position) will then be added to this base address to calculate the actual hi-res screen byte to be modified.

4. The ASCII value of the character to be printed will be used to determine the position in a character data table from which the eight bytes containing the data for the character will be retrieved. The position can be determined by first subtracting 32 from the ASCII value (to make up for the missing control characters in the table). The resulting value is then multiplied by eight (for eight bytes per character) to determine the correct starting position of the data for that particular character. The general formula, then, is:

$$\text{Position} = (\text{ASCII value} - 32) \times 8$$

5. The character will actually be produced by storing the first byte in the calculated base address. The next seven bytes will then be stored at the addresses determined by successively adding the value \$400 to the base address.

6. At that point the printing to the hi-res screen will be complete. The original character to be printed will then be sent to COUT1 (\$FDF0) so that the Monitor routines can handle carriage returns, backspaces, and so on. This action by the Monitor will automatically ensure that the BASL, BASH pair is maintained properly so that we can always rely on its accuracy in positioning the text output on the screen.

This last point may need a bit of explanation. If we never sent a character to COUT1, we would have to handle the entire screen management ourselves. This means that when we got to the end of the line, we would have to detect it and then advance CV and recalculate BASL, BASH accordingly. By passing each character to COUT1 (even though technically we never see the text screen), the Monitor will keep BASL, BASH, CH, and CV all maintained in a way consistent with the data printed to the screen.

Thus all we need to do is look at BASL, BASH, CH, and CV for each character printed to have the hi-res screen properly mimic what is going on with the text display page.

Here, then, is the listing for the hi-res character generator:

```

1 *****
2 *   AL31-CHARACTER GENERATOR   *
3 *****
4 *
5 *           OBJ  $300
6 *           ORG  $300
7 *
```

```

      8 CSW EQU $36
      9 BASL EQU $28
     10 CH EQU $24
     11 TABLE EQU $9000
     12 POSN EQU $3C ; (BAS2)
     13 SCRN EQU $3E ; (A4)
     14 VECT EQU $3EA
     15 COUT1 EQU $FDF0
     16 *
0300: A9 0B 17 HOOK LDA #ENTRY ; PRODUCES LOW BYTE
0302: 85 36 18 STA CSW
0304: A9 03 19 LDA #>ENTRY ; #> PRODUCES HIGH BYTE
0306: 85 37 20 STA CSW+1
0308: 4C 3A FF 21 JMP VECT
     22 *
030B: C9 A0 23 ENTRY CMP #$A0
030D: 90 51 24 BCC OUT ; CTRL CHARACTER
030F: 48 25 PHA ; STORE CHAR
0310: 29 7F 26 AND #$7F ; CLEAR HI BIT
0312: 85 3C 27 STA POSN
0314: A9 00 28 LDA #$00
0316: 85 3D 29 STA POSN+1
0318: 98 30 TYA
0319: 48 31 PHA ; SAVE Y
     32 *
031A: 38 33 CALC1 SEC
031B: A5 3C 34 LDA POSN
031D: E9 20 35 SBC #$20
031F: 85 3C 36 STA POSN ; CHAR < 96
0321: 06 3C 37 ASL POSN ; *2 = CHAR < 192
0323: 06 3C 38 ASL POSN ; *4 < 384
0325: 26 3D 39 ROL POSN+1
0327: 06 3C 40 ASL POSN ; *8 < 768
0329: 26 3D 41 ROL POSN+1
     42 *
     43 * POSN = (ASC - $20)*8 BYTES PER CHAR
     44 *
032B: 18 45 CLC
032C: A9 00 46 LDA #TABLE ; LOW BYTE
032E: 65 3C 47 ADC POSN
0330: 85 3C 48 STA POSN
0332: A9 90 49 LDA #>TABLE ; HIGH BYTE
0334: 65 3D 50 ADC POSN+1
0336: 85 3D 51 STA POSN+1 ; POSN = POSN + TABLE ADDR
     52 *
0338: 18 53 CALC2 CLC
0339: A5 28 54 LDA BASL
033B: 65 24 55 ADC CH
033D: 85 3E 56 STA SCRN
033F: A5 29 57 LDA BASL+1
0341: 69 1C 58 ADC #$1C
0343: 85 3F 59 STA SCRN+1 ; SCRN = BASL + CH + $1C00
     60 *
0345: A0 00 61 GETBYTE LDY #$00
0347: B1 3C 62 G1 LDA (POSN),Y
0349: 91 3E 63 STA (SCRN),Y

```

```

034B: C8      64  INC      INY
034C: 18      65          CLC
034D: A5 3E   66          LDA  SCRNI
034F: 69 FF   67          ADC  #$FF
0351: 85 3E   68          STA  SCRNI
0353: A5 3F   69          LDA  SCRNI+1
0355: 69 03   70          ADC  #$03
0357: 85 3F   71          STA  SCRNI+1      ; SCRNI = SCRNI + $3FF
72          *
73          * $3FF TO MAKE UP FOR GROWING VALUE OF 'Y'
74          *
0359: C0 08   75  DONE?   CPY  #$08
035B: 90 EA   76          BCC  G1
77          *
035D: 68      78  YES     PLA
035E: A8      79          TAY          ; RESTORE Y
035F: 68      80          PLA          ; RESTORE CHAR
0360: 4C F0 FD 81  OUT     JMP  COUT1
0363: D8      82          CHK

```

The routine is relatively short and is placed at location \$300 (768 decimal). When a call to \$300 is done by either a 300G from the Monitor or a CALL 768 from BASIC, the routine will set the output vectors to point to ENTRY and then call the DOS hookup routine described in earlier chapters. At this point, all future character output will pass through this routine, until it is disconnected either by a PR#0 or by pressing RESET.

At ENTRY, the first thing that is checked for is to see whether the character being output is a control character. Remember that at this point the high bit will be set on all text going to the screen. Therefore, even though \$20 is the more normal ASCII value for a space character, with the high bit set it will be sent through COUT as an \$A0.

If a control character is detected here, the CMP and BCC will pass control to the exit point of the routine, OUT. Remember that BCC is used to detect all values in the Accumulator less than the value used in the CMP instruction. All control characters will have an ASCII value less than that of the space character.

If the character is a non-control character, it's then pushed onto the stack in line 25. This is to save the character to be printed so that it eventually can be passed on to COUT1. The next line, 26, then clears the high bit of the character and stores the resulting value in POSN.

This resulting true ASCII value will be used shortly to calculate the needed position in our character table, so lines 28 and 29 store a 0 in the high-order-byte position of POSN. Because 96 characters times 8 bytes each will require a table 768 (\$300) bytes long, POSN will have to be able to include a two-byte value. Thus lines 28 and 29 take this opportunity to set the high byte of POSN to 0 now in anticipation of future calculations.

Another bit of programming technique appears on lines 30 and 31. Because the Apple assumes that all output routines will leave all of the registers (X, Y,

and A) unaltered, we must save the Y-Register so as to be able to restore it to its original condition later on exit. To avoid having to use another zero-page location for this, we've delayed saving the Y-Register until now so that its value can be put in the Accumulator and then pushed onto the stack. Prior to saving the character value in POSN, any attempt to put Y in the Accumulator would have erased the value for the character we wanted to print.

Now for the calculation phase. The first step is to subtract 32 from the ASCII value in preparation for calculating the table position. Lines 33 through 36 do this. The next step is to multiply by 8 to get the relative position in the table. Fortunately, 8 is an easy number by which to multiply. You may remember from earlier chapters that a left-shift operation is equivalent to multiplying by 2. Therefore, all we need do is shift left three times to get the effect of multiplying by 8 ($2 \times 2 \times 2 = 8$).

Normally, because POSN is a two-byte value, each shift would have to be a set of ASLs and ROLs. However, because we know we're starting with a value less than 96, we know the first shift cannot possibly give a result greater than 256. In looking at lines 37 through 41, you can see that line 37 does the first multiply by 2. It is then lines 38 through 41 that do the two-byte shifts to get the final result. Remember also that an ASL puts the bit pushed out the end into the carry flag. That allows ROL to pick up the carry when shifting the high-order byte.

Consider the example in the table below to see how the shifts work. The letter A has an ASCII value of \$41 (65 decimal). After subtracting \$20 (32 decimal) we'll have a result of \$21 (33 decimal). After multiplying by 8, we should get a result of \$108 (264 decimal).

Program Command	POSN+1 (hex)	POSN (hex)	POSN+1	Carry	POSN
36: Start	\$00	\$21	%0000 0000	0	%0010 0001
37: ASL POSN	\$00	\$42	%0000 0000	0	%0100 0010
38: ASL POSN	\$00	\$84	%0000 0000	0	%1000 0100
39: ROL POSN+1	\$00	\$84	%0000 0000	0	%1000 0100
40: ASL POSN	\$00	\$08	%0000 0000	1	%0000 1000
41: ROL POSN+1	\$01	\$08	%0000 0001	0	%0000 1000

Once the multiplication by 8 has been done, the only thing remaining is to take the relative offset position determined and add that to the base address of the table. In this case, we will assume that the table has been loaded at \$9000 (and presumably protected by setting HIMEM: 36864).

Once the table position is calculated, the screen byte to be modified must be calculated as well. This is done by CALC2. Lines 53 through 59 take the contents of BASL, BASH and add \$1Cxx to that, where xx is the value of CH at that point. Adding \$1C00 gives the base address of the hi-res screen line corresponding to the current text-page line. We could have used the Y-Register for CH, but that would have prevented us from easily using the Y-Register to index the character table data. Therefore, we add CH to make BASL, BASH the address of the first hi-res

screen byte to be modified. Note that an added advantage of this approach is that HTAB and VTAB commands will continue to work on the hi-res page. Scrolling, however, will not be available.

GETBYTE (line 61) is the section responsible for putting the character on the hi-res screen. This is done in a number of stages. The first step is to set the Y-Register to # 00 to prepare to retrieve the data bytes from the table. G1 then starts the retrieval loop by getting the first byte of the character from the table and storing it on the hi-res screen.

Now here's where it gets interesting. Normally, the next steps would be to increment Y to get the next character from the table, and to also add 400 to the POSN value to access the next horizontal line on the screen. The problem is that, if Y changes, we won't access the line directly below the one we just modified, but rather one byte to the right of where we want to be.

The solution is to add $3FF$, rather than 400 , to POSN. That way the value of POSN will grow in a way compatible with the increased value of the Y-Register. This part of the listing is worth studying until you understand the concept. It saves a lot of needless storing of the Y-Register and hence needless extra time and memory usage. The technique can be applied to many other situations as well.

Once the entire eight bytes have been put on the hi-res screen, lines 78 through 81 restore the Accumulator to the value of the original character to be printed and the Y-Register to its original value. The jump to COUT1 ($DF0$) is then done to complete the printing to the normal text screen. The advantages of this were discussed earlier (maintenance of BASL, BASH, CV, and so on).

A Hi-Res Character Set

The way to use the character generator is to load the assembled binary routine at 300 (768 decimal). In an Applesoft program, you would then execute an HGR command, followed by a CALL 768 to activate the routine.

If you were to use the routine entirely from assembly language, you would have to call HGR directly. See chapter 19 for more information on calling the hi-res subroutines.

There is, however, one minor detail still missing. That is the table that we assumed existed at 9000 . Since you don't yet have a means of easily creating your own character set, you'll need a table to use.

This data, although lengthy, will provide you with a complete character set to be loaded at 9000 . Although it will take a while to enter the data, it will probably be a little easier than creating each character with an editor, although you will have that opportunity in the next chapter.


```

1 *****
2 * AL31-ASCII CHARACTER SET *
3 *****
4 *
5 ORG $9000
6 *
9000: 00 00 00 00 00 00 00 00 7 HEX 0000000000000000 ; SPACE
9008: 08 08 08 08 08 00 08 00 8 HEX 0808080808000800 ; !
9010: 14 14 14 00 00 00 00 00 9 HEX 1414140000000000 ; "
9018: 14 14 3E 14 3E 14 14 00 10 HEX 14143E143E141400 ; #
9020: 08 3C 0A 1C 28 1E 00 00 11 HEX 083C0A1C281E0000 ; $
9028: 06 26 10 08 04 32 30 00 12 HEX 0626100804323000 ; %
9030: 04 0A 0A 04 2A 12 2C 00 13 HEX 040A0A042A122C00 ; &
9038: 0B 00 06 00 00 00 00 00 14 HEX 0808080000000000 ; '
9040: 08 04 02 02 02 02 04 08 00 15 HEX 080402020202040800 ; (
9048: 08 10 20 20 20 10 08 00 16 HEX 0810202020100800 ; )
9050: 08 2A 1C 08 1C 2A 08 00 17 HEX 082A1C081C2A0800 ; *
9058: 00 08 08 3E 08 08 00 00 18 HEX 0008083E08080000 ; +
9060: 00 00 00 00 00 00 00 04 19 HEX 0000000000000004 ; ,
9068: 00 00 00 3E 00 00 00 00 20 HEX 0000003E00000000 ; -
9070: 00 00 00 00 00 00 08 00 21 HEX 0000000000000800 ; .
9078: 00 20 10 08 04 02 00 00 22 HEX 0020100804020000 ; /
9080: 1C 22 32 2A 26 22 1C 00 23 HEX 1C22322A26221C00 ; 0
9088: 08 0C 08 00 08 08 1C 00 24 HEX 080C080808081C00 ; 1
9090: 1C 22 20 18 04 02 3E 00 25 HEX 1C22201804023E00 ; 2
9098: 3E 20 10 18 20 22 1C 00 26 HEX 3E20101820221C00 ; 3
90A0: 10 18 14 12 3E 10 10 00 27 HEX 101814123E101000 ; 4
90A8: 3E 02 1E 20 20 22 1C 00 28 HEX 3E021E2020221C00 ; 5
90B0: 18 04 02 1E 22 22 1C 00 29 HEX 1804021E22221C00 ; 6
90B8: 3E 20 10 08 04 04 04 00 30 HEX 3E20100804040400 ; 7
90C0: 1C 22 22 1C 22 22 1C 00 31 HEX 1C22221C22221C00 ; 8
90C8: 1C 22 22 3C 20 10 0C 00 32 HEX 1C22223C20100C00 ; 9
90D0: 00 00 08 00 08 00 00 00 33 HEX 0000080008000000 ; :
90D8: 00 00 08 00 08 08 04 00 34 HEX 0000080008080400 ; ;
90E0: 10 08 04 02 04 08 10 00 35 HEX 1008040204081000 ; <
90E8: 00 00 3E 00 3E 00 00 00 36 HEX 00003E003E000000 ; =
90F0: 04 08 10 20 10 08 04 00 37 HEX 0408102010080400 ; >
90F8: 10 22 10 08 08 00 08 00 38 HEX 1C22100808000800 ; ?
9100: 1C 22 2A 3A 1A 02 3C D0 39 HEX 1C222A3A1A023CD0 ; @
9108: 08 14 22 22 3E 22 22 00 40 HEX 081422223E222200 ; A
9110: 10 22 22 1E 22 22 1E 00 41 HEX 1E22221E22221E00 ; B
9118: 1C 22 02 02 02 22 1C 00 42 HEX 1C22020202221C00 ; C
9120: 1E 22 22 22 22 22 1E 00 43 HEX 1E22222222221E00 ; D
9128: 3E 02 02 1E 02 02 3E D0 44 HEX 3E02021E02023ED0 ; E
9130: 3E 02 02 1E 02 02 02 00 45 HEX 3E02021E02020200 ; F
9138: 3C 02 02 02 32 22 3C 00 46 HEX 3C02020232223C00 ; G
9140: 22 22 22 3E 22 22 22 00 47 HEX 2222223E22222200 ; H
9148: 1C 08 06 08 08 08 1C 00 48 HEX 1C08080808081C00 ; I
9150: 20 20 20 20 20 22 1C 00 49 HEX 2020202020221C00 ; J
9158: 22 12 0A 06 0A 12 22 00 50 HEX 22120A060A122200 ; K
9160: 02 02 02 02 02 02 3E 00 51 HEX 0202020202023E00 ; L
9168: 22 36 2A 2A 22 22 22 00 52 HEX 22362A2A22222200 ; M
9170: 22 22 26 2A 32 22 22 00 53 HEX 2222262A32222200 ; N
9178: 1C 22 22 22 22 22 1C 00 54 HEX 1C22222222221C00 ; O
9180: 1E 22 22 1E 02 02 02 00 55 HEX 1E22221E02020200 ; P
9188: 1C 22 22 22 2A 12 2C 00 56 HEX 1C2222222A122C00 ; Q

```

```

9190: 1E 22 22 1E 0A 12 22 00 57 HEX 1E22221E0A122200 ; R
9198: 1C 22 02 1C 20 22 10 00 58 HEX 1C22021C20221C00 ; S
91A0: 3E 08 03 08 03 08 08 00 59 HEX 3E08080808080800 ; T
91A8: 22 22 22 22 22 22 1C 00 60 HEX 2222222222221C00 ; U
91B0: 22 22 22 22 22 14 06 00 61 HEX 2222222222140800 ; V
91B8: 22 22 22 2A 2A 36 22 00 62 HEX 2222222A2A362200 ; W
91C0: 22 22 14 08 14 22 22 00 63 HEX 2222140814222200 ; X
91C8: 22 22 22 14 03 08 03 00 64 HEX 2222221408080800 ; Y
91D0: 3E 20 10 08 04 02 3E 00 65 HEX 3E20100804023E00 ; Z
91D8: 3E 06 06 0E 06 06 3E 00 66 HEX 3E06060606063E00 ; [
91E0: 00 02 04 06 10 20 00 00 67 HEX 0002040810200000 ; \
91E8: 3E 30 30 30 3D 30 3E 00 68 HEX 3E303030303D303E00 ; ]
91F0: 00 00 08 14 22 00 00 00 69 HEX 0000081422000000 ; ^
91F8: 00 00 00 00 00 00 00 7F 70 HEX 000000000000007F ; _
9200: 04 08 10 00 00 00 00 00 71 HEX 0408100000000000 ; '
9208: 00 00 1C 20 3C 22 3C 00 72 HEX 00001C203C223C00 ; a
9210: 02 02 1E 22 22 22 1E 00 73 HEX 02021E2222221E00 ; b
9218: 00 00 3C 02 02 02 3C 00 74 HEX 00003C0202023C00 ; c
9220: 20 20 3C 22 22 22 3C 00 75 HEX 20203C2222223C00 ; d
9228: 00 00 1C 22 3E 02 3C 00 76 HEX 00001C223E023C00 ; e
9230: 18 24 04 1E 04 04 04 00 77 HEX 1824041E04040400 ; f
9238: 00 00 1C 22 22 3C 20 1C 78 HEX 00001C22223C201C ; g
9240: 02 02 1E 22 22 22 22 00 79 HEX 02021E2222222200 ; h
9248: 08 00 0C 08 08 08 1C 00 80 HEX 08000C0808081C00 ; i
9250: 10 00 18 10 10 10 12 00 81 HEX 1000181010101200 ; j
9258: 02 02 22 12 0E 12 22 00 82 HEX 020222120E122200 ; k
9260: 0C 03 08 0B 08 08 1C 00 83 HEX 0C08080808081C00 ; l
9268: 00 00 36 2A 2A 2A 22 00 84 HEX 0000362A2A2A2200 ; m
9270: 00 00 1E 22 22 22 22 00 85 HEX 00001E2222222200 ; n
9278: 00 00 1C 22 22 22 1C 00 86 HEX 00001C2222221C00 ; o
9280: 00 00 1E 22 22 1E 02 02 87 HEX 00001E22221E0202 ; p
9288: 00 00 3C 22 22 3C 20 20 88 HEX 00003C22223C2020 ; q
9290: 00 00 3A 06 02 02 02 00 89 HEX 00003A0602020200 ; r
9298: 00 00 3C 02 1C 20 1E 00 90 HEX 00003C021C201E00 ; s
92A0: 04 04 1E 04 04 24 18 00 91 HEX 04041E0404241800 ; t
92A8: 00 00 22 22 22 32 2C 00 92 HEX 0000222222322C00 ; u
92B0: 00 00 22 22 22 14 08 00 93 HEX 0000222222140800 ; v
92B8: 00 00 22 22 2A 2A 36 00 94 HEX 000022222A2A3600 ; w
92C0: 00 00 22 14 08 14 22 00 95 HEX 0000221408142200 ; x
92C8: 00 00 22 22 14 08 08 06 96 HEX 0000222214080806 ; y
92D0: C0 00 3E 10 08 04 3E 00 97 HEX 00003E1008043E00 ; z
92D8: 38 0C 0C 06 0C 0C 30 00 98 HEX 380C0C060C0C3000 ; {
92E0: 08 0B 03 08 08 08 08 08 99 HEX 0808080808080808 ; |
92E8: 0E 18 1B 30 18 18 0E 00 100 HEX 0E18183018180E00 ; }
92F0: 2C 1A 00 00 00 00 00 00 101 HEX 2C1A000000000000 ; ~
92F8: 7F 7F 7F 7F 7F 7F 7F 7F 102 HEX 7F7F7F7F7F7F7F ; CURSOR
9300: 6F 103 CHK

```

As a side note, this is an odd program in that it doesn't actually do anything. It just creates a data table. Assemble it anyway and save the object code under the name AL31.ASCII.

To test all of this out, you can use this simple Applesoft program.¹ You probably should verify that you can at least get this much to work before diving in and trying to use the routines from within your own assembly-language programs.

```

10 PRINT CHR$(21): REM 40-COLUMN
20 PRINT "CHAR TABLE FILE, <RETURN> FOR DEFAULT": INPUT A$
30 IF LEN(A$) = 0 THEN A$ = "AL31.ASCII"
40 PRINT CHR$(4); "BLOAD "; A$
50 PRINT CHR$(4); "BLOAD AL31.CHARGEN,A$300
60 HGR: HCOLOR= 3
70 HPLOT 0,0 TO 279,0
80 HPLOT TO 279,159
90 HPLOT TO 0,159
100 HPLOT TO 0,0: REM DRAW FRAME
110 REM IF DOS 3.3 THEN SET UP CSW VECTOR
120 IF PEEK(1002) = 76 THEN CALL 768: GOTO 150
130 REM IF PRODOS, SET UP OUTPUT LINK AT $BE30,31
140 POKE 48688,11: POKE 48689,3
150 VTAB 1: HTAB 10
160 PRINT "HI-RES CHARACTER GENERATOR"
170 END
180 REM USE RESET OR PR#0 TO TURN OFF

```

Conclusion

At this point you should feel fairly comfortable with the idea of how a hi-res character generator works. The ideas presented here rely heavily on a general degree of familiarity with a variety of techniques discussed in earlier chapters, specifically, output vector use and interception, memory mapping of the hi-res and text screens, and of course general techniques of assembly-language programming. If you are having difficulty in any of these areas, you may wish to review previous chapters.

All in all, you should find the approach shown here to be much easier than you first thought. The similarities between the text and hi-res screens greatly reduce the amount of difficulty in creating a character generator.

In the next chapter, we'll develop a character editor to create your own hi-res character fonts (the term used for the character design), and also take a brief look at how hi-res graphics in arcade-style games can take advantage of these same techniques to create a wide variety of effects.

¹ [CT] For ProDOS, we manually change the output vector at \$BE30, \$BE31 to point to ENTRY (\$30B). See footnote 1 in chapter 29 for more discussion.

Hi-Res Character Editor

May 1983

In the previous chapter we presented a listing for a hi-res character generator along with the theory behind its operation. The generator used an existing character set, loaded at location \$9000 in memory, and contained the data for 96 ASCII characters.

To create your own character set, all that is needed is a utility for editing the existing character set and creating the new font, or character design, that you desire.

Before presenting the listing for the character editor, consider for a moment the information and techniques that must be provided for. This is a very important part of solving any problem, programming or otherwise, and is instrumental in directing and clarifying one's thought processes.

In discussing the character set, you'll recall that each character is represented by a series of eight bytes in the table, and that each dot in the character image is represented by a bit within one of those bytes. The first two considerations, therefore, are how to address the series of bytes that correspond to a given ASCII character and how to identify and alter the bit corresponding to the particular dot in the character image that we wish to modify.

In editing each character, we will want to be able to turn a given bit on or off (set it to 1 or 0) and to move a cursor from one bit to another. You'll also recall from the previous chapter that each byte of the character's data corresponds to one line of its image on the screen. Within each byte, seven bits are used to map the seven screen dots used to generate a given line of a character.

When we edit the individual screen dots, it would be nice if we could use the standard arrow keys to move the cursor around in a box containing the character image.

Speaking of the character box, some thought will have to be given to how the entire character itself will be displayed. We could just print the character on-screen each time a modification is done, but because of the small size this would become tedious after a while. A better approach would be to display a magnified image of the character, upon which our cursor can be positioned to edit any particular bit in the overall image.

To use the editor, we'll also have to be able to specify which character we want to edit, and then later to signify that we are done. To keep things simple,

we'll select a character by pressing the equivalent key and store the completed image back in the character table when <RETURN> is pressed.

Loading and saving of the complete table is not provided for in the editor but can be accomplished easily from the immediate mode with BLOAD and BSAVE. More on that later.

Here, then, is the complete listing, which will be explained in detail.¹ See you at the bottom!

```

1 *****
2 *      AL32-CHARACTER EDITOR      *
3 *              2/7/1983          *
4 *****
5          ORG  $8000
6 CSW      EQU  $36
7 BASL     EQU  $28
8 CV       EQU  $25
9 CH       EQU  $24
10 CR      EQU  $06
11 CC      EQU  $07
12 MASK    EQU  $08
13 CHR     EQU  $09
14 TABLE  EQU  $9000
15 POSN    EQU  $3C      ; (BAS2)
16 SCRN    EQU  $3E      ; (A4)
17 VECT    EQU  $3EA
18 COUT    EQU  $FDED
19 COUT1   EQU  $FDF0
20 HGR     EQU  $F3E2
21 HCOLOR  EQU  $F6F0
22 HPL0T   EQU  $F457
23 HLIN    EQU  $F53A
24 X1      EQU  $22      ; 34
25 X2      EQU  $54      ; 84
26 Y1      EQU  $17      ; 23
27 Y2      EQU  $58      ; 88
28 VTAB    EQU  $FC22
29 RDKEY   EQU  $FD0C
30 BELL    EQU  $FBDD
31 B1      EQU  %10101010
32 B2      EQU  %01010101
33 *
34 CURDAT  EQU  $FFFF
35 *
8000: A9 81 36 HOOK   LDA  #HCOUT   ; PRODUCES LOW BYTE
8002: 85 36 37       STA  CSW
8004: A9 81 38       LDA  #>HCOUT  ; #> PRODUCES HIGH BYTE
8006: 85 37 39       STA  CSW+1
8008: 4C DD FB 40       JSR  VECT
41 *
800B: 20 E2 F3 42 ENTRY  JSR  HGR
800E: A9 00 43       LDA  #$00
8010: 85 06 44       STA  CR      ; CR=0

```

¹ [CT] Lines 210 and 218–222 were modified to allow you to press <CTRL>Q to quit.

```

8012: 85 07    45          STA  CC          ; CC=0
8014: EA      46        TITLE  NOP
      47        *
8015: A9 03    48        CHRLIST LDA  #$03
8017: 85 25    49          STA  CV
8019: 20 22 FC 50          JSR  VTAB
801C: A2 20    51        START  LDX  #$20
801E: 8A      52        CH2    TXA
801F: 29 0F    53          AND  %#00001111 ; 2^4 - 1
      54        * RESULT = VALUE MOD 16
8021: D0 09    55          BNE  CONT          ; NOT MULT OF 16
8023: A9 8D    56          LDA  #$8D
8025: 20 ED FD 57          JSR  COUT          ; PRINT RETURN
8028: A9 14    58          LDA  #$14          ; MARGIN FOR NEW LINE
802A: 85 24    59          STA  CH
802C: 8A      60        CONT    TXA          ; RESTORE CHAR
802D: 09 80    61          ORA  #$80          ; SET HI BIT
802F: 20 ED FD 62          JSR  COUT          ; PRINT CHAR
8032: E8      63        NEXTC   INX
8033: E0 80    64          CPX  #$80
8035: 90 E7    65          BCC  CH2
      66        *
8037: A2 03    67        MATDSP  LDX  #$03
8039: 20 F0 F6 68          JSR  HCOLOR
803C: A2 22    69        BOX    LDX  #X1          ; LOW BYTE
803E: A0 00    70          LDY  #>X1          ; HIGH BYTE
8040: A9 17    71          LDA  #Y1
8042: 20 57 F4 72          JSR  HPLLOT          ; PLOT X1,Y1
8045: A9 54    73          LDA  #X2
8047: A2 00    74          LDX  #>X2
8049: A0 17    75          LDY  #Y1
804B: 20 3A F5 76          JSR  HLIN           ; TO X2,Y1
804E: A9 54    77          LDA  #X2
8050: A2 00    78          LDX  #>X2
8052: A0 58    79          LDY  #Y2
8054: 20 3A F5 80          JSR  HLIN           ; TO X2,Y2
8057: A9 22    81          LDA  #X1
8059: A2 00    82          LDX  #>X1
805B: A0 58    83          LDY  #Y2
805D: 20 3A F5 84          JSR  HLIN           ; TO X1,Y2
8060: A9 22    85          LDA  #X1
8062: A2 00    86          LDX  #>X1
8064: A0 17    87          LDY  #Y1
8066: 20 3A F5 88          JSR  HLIN           ; TO X1,Y1
8069: A9 03    89        MATD2   LDA  #$03
806B: 85 25    90          STA  CV
806D: 20 22 FC 91          JSR  VTAB
8070: A0 00    92        GETROW  LDY  #$00
8072: A9 05    93        GR1    LDA  #$05
8074: 85 24    94          STA  CH
8076: B9 69 81 95          LDA  MAT,Y
8079: A2 00    96        SCAN   LDX  #$00
807B: 4A      97        S1     LSR
807C: 48      98          PHA          ; SAVE RESULT
807D: A9 A0    99          LDA  #$A0          ; SPACE
807F: 90 02   100         BCC  PRINTM

```

```

8081: A9 FF      101          LDA  #$FF
8083: 20 ED FD   102 PRINTM JSR  COUT
8086: 68         103          PLA                      ; RESTORE ACCUM
8087: E8         104 NXTBIT  INX
8088: E0 07      105          CPX  #$07
808A: 90 EF      106          BCC  S1
808C: A9 8D      107          LDA  #$8D                ; RETURN
808E: 20 ED FD   108          JSR  COUT
8091: C8         109 NXTROW  INY
8092: C0 08      110          CPY  #$08
8094: 90 DC      111          BCC  GR1
      112          *
8096: 18         113 CURSOR  CLC
8097: A5 06      114          LDA  CR                  ; CURSOR ROW
8099: 69 03      115          ADC  #$03
809B: 85 25      116          STA  CV
809D: 20 22 FC   117          JSR  VTAB
80A0: 18         118          CLC
80A1: A5 07      119          LDA  CC                  ; CURSOR COLUMN
80A3: 69 05      120          ADC  #$05
80A5: 85 24      121          STA  CH
      122          *
80A7: 20 C2 81  123 CURCALC JSR  SCRNCALC
80AA: A4 06      124 STATUS  LDY  CR
80AC: B9 69 81  125          LDA  MAT,Y
80AF: A6 07      126          LDX  CC
80B1: 4A         127 ST1     LSR
80B2: CA         128          DEX
80B3: 10 FC      129          BPL  ST1
80B5: 90 02      130          BCC  CLEAR
80B7: B0 04      131          BCS  SET
80B9: A9 00      132 CLEAR   LDA  #$00
80BB: F0 02      133          BEQ  PRNTCURS
80BD: A9 08      134 SET     LDA  #$08
      135          *
80BF: 18         136 PRNTCURS CLC
80C0: 69 71      137          ADC  #CURSDATA        ; LOW BYTE
80C2: 85 3C      138          STA  POSN
80C4: A9 00      139          LDA  #$00
80C6: 69 81      140          ADC  #>CURSDATA        ; HIGH BYTE
80C8: 85 3D      141          STA  POSN+1
      142          *
80CA: 20 D0 81  143          JSR  PUTBYTE
80CD: 20 0C FD   144 CMD?   JSR  RDKEY
80D0: C9 A0      145          CMP  #$A0
80D2: 90 12      146          BCC  EDIT                ; CTRL CHAR
80D4: 85 09      147 CHAR   STA  CHR
80D6: 20 9B 81  148          JSR  POSNCALC
80D9: A0 07      149          LDY  #$07
80DB: B1 3C      150 MOVE   LDA  (POSN),Y
80DD: 99 69 81  151          STA  MAT,Y
80E0: 88         152          DEY
80E1: 10 F8      153          BPL  MOVE
80E3: 4C 37 80  154 CHRX   JMP  MATDSP
      155          *
80E6: C9 8D      156 EDIT   CMP  #$8D                ; RETURN

```

```

80E8: D0 14      157          BNE  E1
80EA: A5 09      158 ACCEPT LDA  CHR
80EC: 20 9B 81   159          JSR  POSNCALC
80EF: A0 07      160          LDY  #$07
80F1: B9 69 81   161 XFER   LDA  MAT,Y
80F4: 29 7F      162          AND  #$7F      ; CLEAR BIT 7
80F6: 91 3C      163          STA  (POSN),Y
80F8: 88         164          DEY
80F9: 10 F6      165          BPL  XFER
80FB: 4C 15 80   166 XFX    JMP  CHRLIST
      167 *
80FE: C9 9B      168 E1     CMP  #$9B      ; ESCAPE
8100: D0 18      169          BNE  E2
8102: 38         170 TOGGLE SEC
8103: A6 07      171          LDX  CC
8105: A9 00      172          LDA  #$00
8107: 2A         173 SHFT   ROL
8108: CA         174          DEX
8109: 10 FC      175          BPL  SHFT
810B: 85 08      176          STA  MASK
810D: A4 06      177          LDY  CR
810F: B9 69 81   178          LDA  MAT,Y
8112: 45 08      179          EOR  MASK
8114: 99 69 81   180          STA  MAT,Y
8117: 4C 37 80   181 TGX    JMP  MATDSP
      182 *
811A: C9 8B      183 E2     CMP  #$8B      ; <CTRL>K
811C: D0 0B      184          BNE  E3
811E: C6 06      185 UP     DEC  CR
8120: 10 04      186          BPL  UPX
8122: A9 07      187          LDA  #$07
8124: 85 06      188          STA  CR
8126: 4C 37 80   189 UPX    JMP  MATDSP
      190 *
8129: C9 8A      191 E3     CMP  #$8A      ; <CTRL>J
812B: D0 0F      192          BNE  E4
812D: E6 06      193 DOWN  INC  CR
812F: A5 06      194          LDA  CR
8131: C9 08      195          CMP  #$08
8133: 90 04      196          BCC  DX
8135: A9 00      197          LDA  #$00
8137: 85 06      198          STA  CR
8139: 4C 37 80   199 DX     JMP  MATDSP
      200 *
813C: C9 88      201 E4     CMP  #$88      ; <CTRL>H
813E: D0 0B      202          BNE  E5
8140: C6 07      203 LEFT  DEC  CC
8142: 10 04      204          BPL  LX
8144: A9 06      205          LDA  #$06
8146: 85 07      206          STA  CC
8148: 4C 37 80   207 LX     JMP  MATDSP
      208 *
814B: C9 95      209 E5     CMP  #$95      ; <CTRL>U
814D: D0 0F      210          BNE  E6      ; [CT] CHECK FOR QUIT
814F: E6 07      211 RIGHT INC  CC
8151: A5 07      212          LDA  CC

```



```

8153: C9 07    213      CMP  #$07
8155: 90 04    214      BCC  RX
8157: A9 00    215      LDA  #$00
8159: 85 07    216      STA  CC
815B: 4C 37 80  217  RX     JMP  MATDSP
815E: C9 91    218  E6     CMP  #$91      ; [CT] <CTRL>Q TO QUIT
8160: D0 01    219      BNE  ERR      ; UNKNOWN CTRL CHAR
8162: 60      220  QUIT    RTS           ; [CT] QUIT PROGRAM
8163: 20 DD FB  221  ERR     JSR  BELL
8166: 4C CD 80  222      JMP  CMD?
8169: 55 AA 55  223  MAT     DFB  B2,B1,B2,B1,B2,B1,B2,B1 ; WORKAREA
816C: AA 55 AA  224  *
      224  *
8171: 7F      225  CURSDATA DFB  %01111111
8172: 41      226      DFB  %01000001
8173: 41      227      DFB  %01000001
8174: 41      228      DFB  %01000001
8175: 41      229      DFB  %01000001
8176: 41      230      DFB  %01000001
8177: 41      231      DFB  %01000001
8178: 7F      232      DFB  %01111111
      233  *
8179: 00      234      DFB  %00000000
817A: 3E      235      DFB  %00111110
817B: 3E      236      DFB  %00111110
817C: 3E      237      DFB  %00111110
817D: 3E      238      DFB  %00111110
817E: 3E      239      DFB  %00111110
817F: 3E      240      DFB  %00111110
8180: 00      241      DFB  %00000000
      242  *
8181: C9 A0    243  HCOU    CMP  #$A0
8183: 90 13    244      BCC  OUT      ; DON'T PRINT CTRL CHARS
8185: 48      245      PHA           ; STORE CHAR
8186: 85 3C    246      STA  POSN
8188: 98      247      TYA
8189: 48      248      PHA           ; SAVE Y
      249  *
818A: A5 3C    250  CALC1   LDA  POSN      ; GET CHAR
818C: 20 9B 81  251      JSR  POSNCALC
      252  *
818F: 20 C2 81  253  CALC2   JSR  SCRNCALC
      254  *
8192: 20 D0 81  255  PRINT   JSR  PUTBYTE
      256  *
8195: 68      257      PLA
8196: A8      258      TAY           ; RESTORE Y
8197: 68      259      PLA           ; RESTORE CHAR
8198: 4C F0 FD  260  OUT     JMP  COUT1
      261  *
819B: 29 7F    262  POSNCALC AND  #$7F      ; CLEAR HI BIT
819D: 85 3C    263      STA  POSN
819F: A9 00    264      LDA  #$00
81A1: 85 3D    265      STA  POSN+1
81A3: 38      266      SEC
81A4: A5 3C    267      LDA  POSN

```

```

81A6: E9 20      268          SBC  #$20
81A8: 85 3C      269          STA  POSN      ; CHR < 96
81AA: 06 3C      270          ASL  POSN      ; *2 = CHR < 192
81AC: 06 3C      271          ASL  POSN      ; *4 < 384
81AE: 26 3D      272          ROL  POSN+1
81B0: 06 3C      273          ASL  POSN      ; *8 < 768
81B2: 26 3D      274          ROL  POSN+1
275 *
276 * POSN = (ASC - $20) * 8 BYTES PER CHAR
277 *
81B4: 18         278          CLC
81B5: A9 00      279          LDA  #TABLE    ; LOW BYTE
81B7: 65 3C      280          ADC  POSN
81B9: 85 3C      281          STA  POSN
81BB: A9 90      282          LDA  #>TABLE   ; HIGH BYTE
81BD: 65 3D      283          ADC  POSN+1
81BF: 85 3D      284          STA  POSN+1   ; POSN = POSN + TABLE ADDR
81C1: 60        285          RTS
286 *
81C2: 18         287  SCRNCLC  CLC          ; ENTER WITH BASL,CH SET UP
81C3: A5 28      288          LDA  BASL
81C5: 65 24      289          ADC  CH
81C7: 85 3E      290          STA  SCRNL
81C9: A5 29      291          LDA  BASL+1
81CB: 69 1C      292          ADC  #$1C
81CD: 85 3F      293          STA  SCRNL+1 ; SCRNL = BASL + CH + $1C00
81CF: 60        294          RTS
295 *
81D0: A0 00      296  PUTBYTE  LDY  #$00      ; ENTER WITH POSN,SCRNL SET UP
81D2: B1 3C      297  G1      LDA  (POSN),Y
81D4: 91 3E      298          STA  (SCRNL),Y
81D6: C8         299  INC      INY
81D7: 18         300          CLC
81D8: A5 3E      301          LDA  SCRNL
81DA: 69 FF      302          ADC  #$FF
81DC: 85 3E      303          STA  SCRNL
81DE: A5 3F      304          LDA  SCRNL+1
81E0: 69 03      305          ADC  #$03
81E2: 85 3F      306          STA  SCRNL+1 ; SCRNL = SCRNL + $3FF
307 * $3FF TO MAKE UP FOR GROWING VALUE
308 * OF 'Y'
309 *
81E4: C0 08      310  DONE?   CPY  #$08
81E6: 90 EA      311          BCC  G1        ; NO
81E8: 60        312  YES     RTS
81E9: E7         313          CHK

```

After assembling the listing, BLOAD the character set from chapter 31 at location \$9000. Then BLOAD the character editor at \$8000 (do not BRUN) and type CALL 32768 from Applesoft or 8000G from the Monitor (Applesoft must be the selected language).²

² [CT] An example BASIC program is given at the end of the chapter.

When the program is called, the screen will clear and a box with a matrix pattern inside it will appear, along with the complete character set loaded at \$9000. If the characters appear scrambled, recheck to make sure you have loaded the character set properly at \$9000.

To select a character to edit, simply press any non-control key. An enlarged image of that character should appear in the box. To move the editing cursor around, use the left and right arrows to move left and right, and <CTRL>J and <CTRL>K to move up and down. If you have an Apple //e, the four directional arrows will also work. Even on a standard Apple II, you may find it easier to hold down the <CTRL> key with the little finger of your left hand and then press the H, U, J, and K keys with your right hand to move around.

Pressing <ESCAPE> will toggle bits in the character on and off. To save a character back to the table, press <RETURN>. If you want to start over with a character, simply press the original letter key again.

To save the altered table back to disk, simply press <CTRL>Q, and then type:

```
BSAVE TABLENAME, A$9000, L$300
```

You can replace TABLENAME with any name you wish to give the new character set.

How it Works

Although the listing looks rather long, don't be discouraged. As it happens, much of the listing consists of routines that were presented in earlier chapters. For example, lines 243 through 313 (HCOUT) are the character generator that was described in chapter 31.

To see how the editor works, let's first consider this overview of the program:

HOOK: Hooks up the character generator, HCOUT, to the output vectors so that the hi-res characters can be printed.

ENTRY: Clears the hi-res screen and initializes the column and row counters to 0.

CHRLIST: Prints all 96 ASCII characters to the screen. We'll examine part of the process in detail shortly.

MATDSP: Draws the matrix pattern to indicate where the character will be edited. This is also the entry point for the editing loop for each character. This section can be broken down as follows:

BOX: The Applesoft hi-res routines are used to draw a box with four straight lines. This forms the boundary of the matrix area.

GETROW: Each byte of the matrix pattern is retrieved here, after which SCAN will process and display the individual bits.

SCAN: This section shifts each bit of the row into the carry and, depending on whether it's set, displays a solid or an empty block.

NXTROW : Increments the row counter (the Y-Register) until all eight rows have been displayed.

CURSOR: Calculates the current cursor position using CC (Cursor Column) and CR (Cursor Row).

CURCALC: This part, along with PRNTCURS, determines whether the bit at the cursor position is set. If it is set, a white cursor is printed; if not, an outline of the cursor is displayed.

CMD?: At this point we are ready to get a command from the keyboard. The general theory is to refresh the screen with the routines in MATDSP each time a command is entered. That way we don't have to update only part of the screen specifically.

If a control character is entered, it is assumed that it will either be a directional key or <RETURN>, so control is passed to EDIT.

If a non-control character is entered, it is assumed that this is a character to be edited. MOVE then retrieves the eight bytes for that character and moves them to the work area (MAT).

EDIT: If the user presses <RETURN>, ACCEPT will store the character data back in the table. If <ESCAPE> is pressed, the selected bit within the byte for that row will be toggled.

If one of the directional keys is pressed, the position counters CC and CR are adjusted accordingly.

Pressing a control key other than the legal command characters will generate a BELL sound. In any case, after a key is entered, a jump is made back to MATDSP to start the process over again.

And Now with the Magnifying Glass

The preceding overview showed in general how the editor works. Now we'll spend a little more time examining the particular techniques used in each routine. Some of the routines taken from earlier chapters will not be described in as much detail as those presented here for the first time. You may wish to refer to previous sections if some parts seem difficult. To help you scan through to just the parts that interest you, each section is keyed to the preceding overview.

HOOK

By storing the address of the HCOUNT routine in CSW and then calling VECT (\$3EA), all future output will pass through the HCOUNT routine, allowing us to print the hi-res characters on the screen.

ENTRY

This is the main entry point to the editor; it serves to clear the hi-res screen and initialize the column and row position of the cursor to 0, 0 (upper-left corner of the matrix).

CHRLIST

To display all of the existing characters, CHRLIST loops through the values \$20 through \$7F (32 through 127 = 96 characters). Because we can't print 96 characters on one line, some sort of aesthetic placement is desirable. The format chosen was a group of 6 lines of 16 characters each.

START is the beginning of this loop (X-Register set to \$20), and CH2 is the top of the printing loop. An interesting problem here is how to determine each time we have printed 16 characters. A separate counter could have been kept, but if it were possible to do a modulo function we could just test for our current character counter for multiples of 16. Because modulo returns the remainder of a division, we would expect a result of 0 each time the counter was at a multiple of 16 or, in other words, had just finished another line of 16 characters.

As it happens, the AND instruction can be used to perform the equivalent of a modulo for any power of 2. The technique is to do an AND with the value you want minus 1. Because 16 is a power of 2 ($2^4 = 16$), we need only do an AND #\$0F followed by a BNE to test for each completed line of 16 characters.

If a line has been finished, a carriage return is printed, followed by the equivalent of an HTAB 20.

Notice that as each character is printed the high bit is set with an ORA #\$80. This is to make COUT happy, as it always expects the high bit to be set on characters to be printed.

MATDSP

This section begins the part that creates the matrix display used in editing the individual characters. This section will be executed each time a command character is entered.

The first part, BOX, draws a box outlining the character image using the Applesoft HLIN routines.

Once the box is drawn, the individual bytes must be displayed with the status of each bit indicated. The algorithm is to scan through each bit position, printing a space if the bit is clear and printing a rubout (\$FF) if the bit is set. In the previous chapter's character table, a rubout was a solid block, so this

approach should work. (Note that if you edit the space or the rubout character, the matrix pattern will be altered accordingly.)

There are a total of eight bytes to be retrieved and displayed for each character. GR1 is the section that does the equivalent of an HTAB 5 (for proper screen placement) and then loads a byte from the work area MAT (see line 225). Once a byte is retrieved, SCAN uses the LSR instruction to shift a bit into the carry flag. If the carry is set, a rubout (\$FF) is printed; otherwise a space (\$A0) is printed.

Because the Accumulator will be used to print a character via COUT, the shifted byte is preserved by pushing it onto the stack on line 98 and later pulling it back off on line 103.

After each seven bits are “printed,” a carriage return is printed on lines 107 and 108 and the loop is repeated until all eight bytes have been displayed.

CURSOR

Once the character matrix has been displayed, we need to display the cursor. Lines 115 through 123 use the cursor row and column (CR and CC) to calculate the HTAB, VTAB position. Remember that since we are mirroring actions taken on the text page we can also use the text page as a frame of reference for hi-res screen operations.

STATUS is used to read the particular bit that corresponds to the current cursor position. Note that CR (Cursor Row) conveniently is equal to whichever byte in the individual character definition we will need to read, and that CC (Cursor Column) determines how many bits need to be shifted out to put the one of interest into the carry flag. Depending on whether the bit is clear or set, the Accumulator will be loaded with a \$00 or \$08, the purpose of which will become immediately obvious.

PRNTCURS

Since CH and CV (\$24, \$25) have been set up, we can use a special form of the HCOUT routine, called PRNTCURS, to print a smaller block or a block outline. You'll notice that the hi-res character generator at HCOUT has been modified slightly to use the pointer POSN (\$3C, \$3D) to point to the data table. Our original character generator always assumed that the table would be at \$9000. Normally HCOUT sets POSN to point at \$9000 on lines 278 through 285.

With POSN set up to point at a special two-character definition table on lines 227 through 243, the PUTBYTE routine will do the equivalent of printing one of the two necessary special characters at the cursor position.

You may wish to compare the HCOUT routine contained in the editor with the previous chapter's character generator to see what changes have been made to facilitate the calling by the PRNTCURS routine.

An interesting digression: By avoiding COUT and writing to the screen directly, we are on the verge of being able to do block shapes, a technique used in many hi-res arcade-type games.

CMD?

The processing of the command characters is done in this section. The character is read from the keyboard using the Monitor routine RDKEY (\$FD0C). This routine will place the ASCII value for the key pressed into the Accumulator.

The first major distinction to be made is whether a control character has been pressed. Lines 145 and 146 do this, forwarding any control characters to the EDIT section.

If a non-control character has been pressed, the user wants to edit that character. CHAR and MOVE use the ASCII value of the key pressed to calculate the position of the data of that character in the table, then move that data into the work area, MAT. After the move, a jump is made back to MATDSP to refresh the display with the new character and to get the next command key.

EDIT

If a control key is pressed, one of a number of functions must be performed. We will consider these in the order they are executed.

Return: This implies that the user wants to accept the character as displayed and copy it back into the character table. This is done by essentially reversing the process used by CHAR and MOVE (lines 147 through 153).

Toggle: If <ESCAPE> is pressed, the appropriate bit position must be switched to its opposite condition—off to on or on to off. This is done by creating a mask byte with the proper bit set. To do this, the carry flag is set and the Accumulator loaded with a 0. When an ROL is done, this set bit will be shifted through the Accumulator. By doing the ROL a given number of times (determined by CC) we can set a given bit in the MASK byte (\$08).

Once the mask has been created, we need only retrieve the proper byte from the work area (determined by CR) and then mask it with the MASK byte (lines 178 through 180). Once this is done, we again jump back to MATDSP to refresh the display and get the next character.

Cursor control: To move the cursor around, we'll use the four directional keys on the Apple //e keyboard. Even if you don't have a //e, you can generate the same characters in the manner mentioned earlier in this chapter. To refresh your memory, the keys we'll use will be <CTRL>H, <CTRL>U, <CTRL>J, and <CTRL>K, for left, right, down, and up respectively.

The code on lines 185 through 219 is fairly straightforward. The up and down motions are done by incrementing or decrementing the cursor row counter; left and right motions are done by incrementing or decrementing the cursor column counter. All motions wrap around.

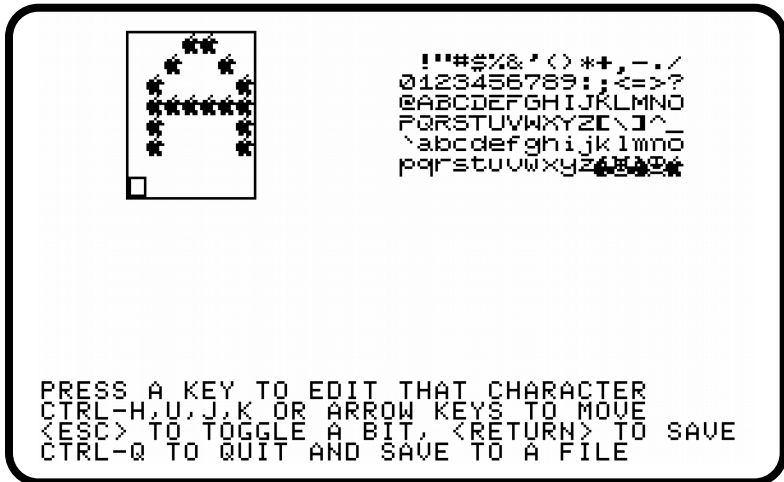
Running the Editor

The following Applesoft program can be used to load a character set, run the character editor, and then save the character set after exiting.³

```

10 PRINT CHR$(21): REM 40-COLUMN
20 PRINT "CHAR TABLE FILE, <RETURN> FOR DEFAULT": INPUT A$
30 IF LEN(A$) = 0 THEN A$ = "AL31.ASCII"
40 PRINT CHR$(4);"BLOAD ";A$
50 PRINT CHR$(4);"BLOAD AL32.CHAREDIT,A$8000"
60 VTAB 21
70 PRINT "PRESS A KEY TO EDIT THAT CHARACTER"
80 PRINT "<CTRL>H,U,J,K OR ARROW KEYS TO MOVE"
90 PRINT "<ESC> TO TOGGLE A BIT, <RETURN> TO SAVE"
100 PRINT "<CTRL>Q TO QUIT AND SAVE TO A FILE";
110 REM IF DOS 3.3 THEN SET UP CSW VECTOR
120 IF PEEK(1002) = 76 THEN CALL 32768: GOTO 150
130 REM IF PRODOS, SET UP OUTPUT LINK AT $BE30,31
140 POKE 48688,129: POKE 48689,129: CALL 32779
150 TEXT: PRINT
160 PRINT "FILENAME TO SAVE, OR <RETURN> TO EXIT": INPUT A$
170 IF LEN(A$) > 0 THEN PRINT CHR$(4);"BSAVE ";A$;" ,A$9000,L$300"
180 END

```



³ [CT] The Applesoft program is new to this edition. Similar to chapter 31, for ProDOS the output vector at \$BE30, \$BE31 is directly changed to point to the routine at HCOUT (\$8181). The main program at \$800B is then run.

Miscellaneous Notes

Although they can be displayed, lower-case characters may not be easy to edit because they are not easily generated from the Apple II keyboard. Apple //e owners will have no trouble. It is possible to use the lowercase input routine described in an earlier chapter to generate lowercase characters from a standard Apple II keyboard. Simply activate the routine prior to calling the character editor. The <ESCAPE> and <SHIFT> functions will continue to work properly, presumably with no ill effects on the editor routines.

It is worth noting that the character sets used and created by this editor are identical in format to the *DOS Tool Kit Animatrix* character sets, although the character editor provided with that package does have one or two minor, though not inconsequential, features not available in this editor.

Conclusion

This concludes our discussion of hi-res character generation and editing; it should provide you with the basic principles of these techniques. The idea can be extended into block graphics for arcade-style games or as improvements to the art of hi-res character generation. You might, for example, want to experiment with oversize letters, colored text, or simple animation.

The 65C02

June 1983

This last chapter deals with a new version of our beloved 6502 microprocessor known as the 65C02. Although the chip has just been released within the last few months and has yet to find its way into the mainstream of computers, it seems likely that we'll be hearing more about this item in the upcoming year.

Before jumping right into its new functions, though, let's first get a little background information out of the way.

The 6502 was designed by Chuck Peddle and Bill Mensch of MOS Technology (now owned by Commodore Business Machines) and, as of the present, 70 percent of its use is by Apple, Atari, and Commodore.¹ The current manufacturers of the 6502 are Rockwell International, MOS Technology, and Synertek. As sometimes happens with these things, though, some of the key persons involved with the 6502 went to work at a new company, Western Design Center.² This company, then, is the original source of the new 65C02 chip. But the story doesn't end there. Western Design Center has licensed the design to at least three independent manufacturers: Rockwell International, GTE, and NCR. These companies took the initial 65C02 design and added their own enhancements.

The picture at this point is that each of these companies will be marketing its own version of the 65C02. The chips are more or less the same, but the Rockwell chip has the largest instruction set.

"Largest instruction set," you ask? Yes! The new 65C02 has had the old 6502 instruction set appended with a variety of new instructions. Because the Rockwell chip appears to be a superset of all of the other chips, the bulk of this chapter will assume that it's the one that's being used. At the end of this chapter we'll describe differences among the three chips.

The Rockwell chip has a total of twelve new instructions and two new addressing modes. In addition, a number of addressing modes not previously available to an instruction (such as the immediate mode for the BIT instruction) are now available. There are a total of 59 actual new opcodes. The meaning of all of these numbers will become clear shortly.

¹ [CT] Corrected from the original article, which listed Commodore Business Machines as the original designer.

² [CT] Specifically, Bill Mensch, the designer of the 65C02.

New Addressing Modes

Since this is one of the smaller numbers, let's start here. You'll recall from many earlier discussions that each 6502 instruction has up to six addressing modes. That number is arrived at by counting some modes as mere variations of others and not including the value (relative addressing) associated with branch instructions (BEQ, BNE, BCC, BCS, and so on) as an addressing mode here. To refresh your memory, a list of modes and variations is provided in the table below for the LDA (LoaD Accumulator) instruction.

Addressing Mode	Common Syntax
1. Absolute	LDA \$1234
Zero Page	LDA \$12
2. Immediate	LDA #\$12
3. Absolute,X	LDA \$1234,X
Zero Page,X	LDA \$12,X
4. Absolute,Y	LDA \$1234,Y
5. (Indirect,X)	LDA (\$12,X)
6. (Indirect),Y	LDA (\$12),Y

Indirect Addressing

The first of the two new addressing modes is quite easy to explain because it is essentially another variation of an existing mode. The new mode is indirect addressing. This may sound very familiar because this is similar to the instructions used to access memory locations via a zero-page pointer. Usually, though, the Y-Register is set to 0 or some other value, which is then added to the address indicated by the zero-page pointer to determine the address of interest.

This is fine for addressing a large table of data, but many times we are interested in only one byte of memory and must then go through the obligatory LDY #\$00 to properly condition the Y-Register. (See entries 5 and 6 in the table above.)

The new instruction allows us to ignore the contents of the Y-Register and gain access to the memory location directly. This conserves two bytes of code for each reference, because the Y-Register does not have to be loaded. If you want to scan a block of memory, such as for a table, this instruction still can be used if you are willing to INC or DEC the zero-page pointer accordingly.

Addressing Mode	Common Syntax
7. Indirect	LDA (\$12)

This new addressing mode is available for the instructions listed below.

Instructions with Indirect Addressing	Opcode
ADC (\$12)	72
AND (\$12)	32
CMP (\$12)	D2
EOR (\$12)	52
LDA (\$12)	B2
ORA (\$12)	12
SBC (\$12)	F2
STA (\$12)	92

Indexed Absolute Indirect

The second new addressing mode has a name that obviously was not designed with easy recall in mind. Fortunately, this too is a variation on an existing theme and as such should be easy to remember. In the past, we had indexed indirect addressing. We called this mode pre-indexed for clarity's sake. An example would be LDA (\$22,X). Pre-indexing means that the contents of the X-Register are added to the address of the zero-page reference before using the sum of those numbers to determine which zero-page pair to use. For example, the instruction LDA (\$22,X), where the X-Register held the value 4, would actually use bytes \$26, \$27 to get the final destination address.

This differs from indirect indexed, which we referred to as post-indexing. In post-indexing, the value of the Y-Register is added after the base address is determined. For example, in the instruction LDA (\$22),Y, where the Y-Register holds the value 4 and \$22, \$23 point to location \$1000, the memory location accessed would be \$1004.

You'll recall also that pre- and post-indexing were limited in their use of the X- and Y-Registers. Pre-indexing could use only the X-Register and post-indexing only the Y-Register. Before you get too excited in anticipating the possibilities of the new instruction, restrain yourself: This much has not changed.

What has changed is that pre-indexing is no longer limited to zero-page pointers. The new mode allows any two-byte value to be used. This means that the X-Register can be added to the base address of a table of memory pointers that previously could have been located only on the zero page of memory.

Addressing Mode	Common Syntax
8. Indexed Absolute Indirect	JMP (\$1234,X)

For example, suppose you had a command interpreter that accepted a command value between 0 and 2. With the 65C02, such an interpreter can now be used in conjunction with a JMP table located anywhere in memory, constructed as in the following example:

JMP DATA TABLE:

1200: 80 10

1202: A0 10

1204: CO 11

```

1 *****
2 * AL33-SAMPLE COMMAND PROCESSOR*
3 *****
4         XC           ; MERLIN: ALLOW 65C02 OPCODES
5         ORG $1000
6 TABLE EQU $1200
7 *
1000: 20 00 40 8 ENTRY JSR GETCMD ; GET VALUE FROM 0-2
1003: 0A 9 ASL ; MULTIPLY BY 2
1004: AA 10 TAX ; PUT IN X-REGISTER
1005: 7C 00 12 11 GO JMP (TABLE,X) ; EXECUTE PROPER ROUTINE
12 *
13 * ...MORE CODE HERE...
1080: EA 50 CMD1 NOP ; FIRST ROUTINE
51 * ...MORE CODE HERE...
10A0: EA 100 CMD2 NOP ; SECOND ROUTINE
101 * ...MORE CODE HERE...
11C0: EA 150 CMD3 NOP ; THIRD ROUTINE
151 * ...MORE CODE HERE...

```

This is a very fast and effective technique. The following table shows the one instruction that can use this new mode.³

Indexed Absolute Indirect Addressing	Opcode
JMP (\$1234,X)	7C

New “Standard” Addressing Modes

There are a few instructions that have addressing modes that are new just to them. For example, two of the most exciting ones are INC and DEC.

Previously, any uses of INC and DEC were limited to memory locations. In addition (so to speak), using the X- and Y-Registers was the only way to maintain a simple loop counter without using a dedicated memory location. The surprise here is that INC and DEC will now work on the Accumulator. This is nice because you can now maintain a counter in the Accumulator, or even do fudging of flag values as they are being handled in the Accumulator.

Some future assemblers may require the “somewhat usual” (if not inconvenient) use of DEC A or INC A as they seem to prefer for LSR, ASL, and other operations on the Accumulator.

The BIT instruction also allows some additional addressing modes that may prove useful. Previously, the BIT instruction supported only absolute addressing.

³ [CT] The original article incorrectly listed eight other instructions: ADC, AND, CMP, EOR, LDA, ORA, SBC, STA. In addition, the code example has been corrected to use the new JMP instruction.

That is to say that a directly referenced memory location was used as the value against which the Accumulator was operated on.

Addressing Mode	Common Syntax
Absolute	BIT \$1234
Zero Page	BIT \$12

This is useful for testing a memory location for a given bit pattern, but not directly suitable for testing the bit pattern of the Accumulator. For many operations, this means you have to rather artificially load some memory location with the value you wanted to compare to the Accumulator.

The new 65C02 supports three new addressing modes for the BIT instruction:

Addressing Mode	Common Syntax	Opcode
Immediate	BIT # \$12	89
Absolute,X	BIT \$1234,X	3C
Zero Page	BIT \$12,X	34

At Last, the Real Scoop! New Instructions

Of course, the real question lurking in everyone's mind is: "But what are the new instructions?"

The great thing about the 65C02 is that not only are many of the old instructions enhanced, but there also are a number of absolutely terrific new instructions—twelve, to be exact. The new instructions are shown in the table below.

Instruction ⁴	Description	Opcode
BBR	Branch on Bit Reset (clear)	0F 1F 2F 3F 4F 5F 6F 7F
BBS	Branch on Bit Set	8F 9F AF BF CF DF EF FF
BRA	BRanch Always	80
PHX	Push X onto stack	DA
PHY	Push Y onto stack	5A
PLX	Pull X from stack	FA
PLY	Pull Y from stack	7A
RMB	Reset (clear) Memory Bit	07 17 27 37 47 57 67 77
SMB	Set Memory Bit	87 97 A7 B7 C7 D7 E7 F7
STZ	STore Zero	64 74 9C 9E
TRB	Test and Reset (clear) Bit	14 1C
TSB	Test and Set Bit	04 0C

⁴ [CT] The BBR, BBS, RMB, and SMB instructions apparently were never available on any 65C02 chips used by Apple.

So what exactly do these instructions do? Well, let's examine some of the easy ones first...

PHX, PHY, PLX, and PLY

Commands for pushing a byte onto the stack and pulling a byte off of the stack exist for the Accumulator but not for the X- and Y-Registers in the 6502. One of the more common uses for the stack is to save all of the registers prior to going into a routine so that everything can be restored just prior to exiting. Ordinarily, to save the Accumulator, X-Register, and Y-Register, we'd have to do something like this:

```

ENTRY   PHA           ; SAVE A
        TXA           ; PUT X IN A
        PHA           ; SAVE IT
        TYA           ; PUT Y IN A
        PHA           ; SAVE IT
WORK    NOP           ; YOUR PROGRAM HERE
DONE    PLA           ; GET Y
        TAY           ; PUT IT BACK
        PLA           ; GET X
        TAX           ; PUT IT BACK
        PLA           ; GET A
EXIT    RTS

```

The problem is complicated even further in programs like the character generator listed in chapter 31. There we had to refer to the original value of the Accumulator several times, and this interfered with any simple way to push all of the register data onto the stack.

With the new 65C02, this could all be resolved with the following:

```

ENTRY   PHX           ; SAVE X
        PHY           ; SAVE Y
        PHA           ; SAVE A
WORK    NOP           ; YOUR PROGRAM HERE
DONE    PLA           ; GET A
        PLY           ; GET Y
        PLX           ; GET X
EXIT    RTS

```

All four are one-byte commands, addressing only the indicated register.

BRA

BRA (branch always) is one of those instructions that will thrill writers of relocatable code. One of the techniques for writing code that is location-independent involves the use of a forced branch instruction, such as:

```

CLC           ; CLEAR CARRY
BCC LABEL    ; ALWAYS

```

Unfortunately, this means we must force some flag of the Status Register, which may not be convenient at the time. In addition, the process takes up an extra byte on most occasions.

BRA alleviates both of these problems by always branching to the desired address, subject of course to the usual limitations of plus or minus 128 bytes as the maximum branching distance.

It is worth mentioning, in the interest of programming style, that many people indiscriminately use a JMP to go back to the top of a loop when a branch instruction would do the trick; this only adds one more limitation to the final code in the process. Hopefully, this new branch instruction will encourage people to make their code more location-independent. BRA, like the rest of the branch instructions on the 65C02, uses only relative addressing.

STZ

STZ (STore Zero) is used for zeroing out memory bytes without changing the contents of any of the registers.

Many times it is necessary to set a number of internal program registers to 0 before proceeding with the routine. This is especially needed in mathematical routines such as multiplication and division.

Ordinarily this is done by loading the Accumulator with 0 and then storing that value in the appropriate memory locations. This is easy to do when you have to load the Accumulator, X-Register, or Y-Register with 0 anyway. The problem is that on occasion the only reason one of the registers is loaded with 0 is because of the need to zero a memory location.

STZ allows us to zero out any memory byte without altering current register contents. Not all of the addressing modes usually available to the STA, STX, or STY instructions are available with STZ, though. The following table shows what modes are available.

STZ Addressing Modes	Common Syntax
Absolute	STZ \$1234
Zero Page	STZ \$12
Absolute,X	STZ \$1234,X
Zero Page,X	STZ \$12,X

SMB and RMB

SMB and RMB (Set/Reset Memory Bit) will allow you to set or clear a given bit of a byte in memory. Previously this would have required three separate instructions to achieve the same result. For example:

```
LDA MEMORY ; LOAD VALUE FROM MEMORY
AND #$7F ; %0111 1111 IS PATTERN NEEDED TO CLEAR BIT 7
STA MEMORY ; PUT IT BACK
```


With the new instruction, we can accomplish the same thing with:

```
RMB7 MEMORY      ; RESET (CLEAR) BIT 7 OF MEMORY
```

or set the bit again with:

```
SMB7 MEMORY      ; SET BIT 7 OF MEMORY
```

There are two interesting things to note here. The first is that for some reason the term “reset” is used instead of “clear” to indicate the zeroing of a given bit. The second item is that we now have four-character instruction codes (mnemonics), the last character being the number of the bit being acted on. What problems this may cause in some assemblers remains to be seen, but this new species of instruction seems to have arrived.⁵ These instructions are limited to zero-page addressing only.

BBS and BBR

BBS and BBR (Branch on Bit Set/Reset) are two new branch instructions that make it possible to test any bit of a zero-page location and then branch depending on its condition. This instruction will be very useful for testing flags in programs that need to pack flag-type data into as few bytes as possible. By transferring I/O device registers to zero page, it is also possible to test bits in these registers directly for status-bit conditions.

These instructions are very similar in both appearance and usage to the SMB and RMB instructions just discussed. They, too, use four-character mnemonics. The difference, of course, is that we are testing bit status rather than changing it. These are three-byte instructions, the first byte being the opcode, the second being the byte to test, and the third being a relative branch value. In assembly, these commands actually will require two labels!⁶

One of the first applications is the testing of whether a number is odd or even. Previously, this had to be done with an LSR or ROR instruction, followed by a test of the carry flag, such as:

```
LDA MEMORY      ; LOAD A WITH VALUE
LSR              ; SHIFT BIT 0 INTO CARRY
BCS ODD         ; SET IF ODD
BCC EVEN        ; CLEAR IF EVEN
```

The equivalent can now be done without affecting the carry flag or the Accumulator:

```
BBR0 MEMORY,EVEN ; BRANCH IF BIT 0 = 0 = EVEN
BBS0 MEMORY,ODD  ; BRANCH IF BIT 0 = 1 = ODD
```

⁵ [CT] The problem is moot since SMB and RMB are not available on most Apple machines.

⁶ [CT] Again moot since BBS and BBR are not available.

This also could be useful in creating drivers for the new Apple //e 80-column extended memory board since this card uses one bank of memory or the other for the text screen, depending on whether the screen column position is odd or even.

TSB and TRB

TSB and TRB (Test and Set/Reset Bit) are the most complex of the new instructions. These instructions are rather like a combination of the BIT and AND/ORa instructions of the 6502. They seem primarily designed for controlling I/O devices but may have other interesting applications as things develop.

The action of these two instructions is to use a mask stored in the Accumulator to condition a memory location. The mask in the Accumulator is unaltered, but the Z-flag of the Status Register is conditioned based on the memory contents prior to the operation.

For example, to set both bits 0 and 7 of a memory location we could use the following set of instructions:

```
LDA #$81      ; %1000 0001 = MASK PATTERN
TSB MEM1     ; SET BITS 0,7 OF MEMORY
BNE PRSET    ; ONE OF THESE WAS 'ON' ALREADY
BEQ PRCLR    ; NEITHER OF THESE WAS 'ON' ALREADY
```

This would clear the bits:

```
LDA #$81      ; %1000 0001 = MASK PATTERN
TRB MEM2     ; CLR BIT 0,7 OF MEMORY
BNE PRSET    ; ONE OF THESE WAS 'ON' ALREADY
BEQ PRCLR    ; NEITHER OF THESE WAS 'ON' ALREADY
```

These instructions use only absolute and zero-page addressing.

Other Differences

There are a number of other differences between the 6502 and 65C02, most notably the power consumption. The power use of the 65C02 is one-tenth that of the 6502, so the chip runs considerably cooler. The lower power requirement opens new possibilities for portable computers and terminals.

One point of interest is that the old 6502 indirect jump problem has been fixed. If you're not aware of it, the 6502 has a well-documented problem with indirect jumps that use a pair of bytes that straddle a page boundary.

For example, consider these three instructions:

Instruction	Pointers Wanted	Pointers Used
JMP (\$36)	\$36, \$37	\$36, \$37
JMP (\$380)	\$380, \$381	\$380, \$381
JMP (\$3FF)	\$3FF, \$400	\$3FF, \$300

Notice that, in the third instance, the pointers used are not those anticipated. This is because the high byte of the pointer address is not properly incremented by the standard 6502.

This problem has been fixed in the 65C02. The only possible problem here is “clever” protection schemes that use this bug to throw off people trying to decode the protection method. Otherwise, this should not present any problems to existing software.

Are there any problems to be anticipated? In theory, no. The new 65C02 is compatible pin for pin with the old one, and also upwardly compatible in terms of software. Software for the Apple, PET, Atari, or other 6502-based microcomputers should work without problems with the new chip. Are there any exceptions? Unfortunately, yes.

The first big problem concerns internal microprocessor timing on the Apple II and II Plus computers. The Apple II and II Plus do not handle the microprocessor clock cycles in the same way the Apple //e does. On the surface, the 65C02 should directly replace the 6502; however, because the 65C02 is a faster chip, data is not available for as long and bits can get lost. What this means for now is that the 65C02 can be used only in the Apple //e and Apple /// machines. None of the manufacturers at this time produce a chip that works on the Apple II or II Plus. It can be expected, though, that revisions will be made in the near future that will allow the 65C02 to be implemented in the older machines.

There also is a possibility of problems with some existing software. A small percentage of software may be using undocumented bugs or “features” of the old 6502 chip, and these might not function as anticipated with the 65C02.

For example, a reasonable question might be, “Where did all the new opcodes come from? After all, wasn’t the chip full?” To answer this, consider how the instructions we use now are structured. The 6502 operates by scanning memory and performing specific operations based on the values that it finds in each memory location. You would then expect a total of 256 possible opcodes. As it happens, all 256 possible values are not used. It is this group of unused opcodes that allows for the new instructions and also creates the possibility of a small percentage of difficulties with existing programs.

Although rarely documented, the previously “unused” values in the 6502 will cause certain things to happen, much the same way that a legal value would. For instance, the code \$FF on a 6502 is labeled as an alternate NOP. This is one of the codes that have been converted to a new function in the 65C02, namely BBS7 (Branch on Bit 7 Set).

There are other unused codes, though, that have combination effects—usually of little use—such as loading the Accumulator and decrementing a register at the same time. Their main application is similar to the indirect jump problem: creating code that cannot be casually interpreted. If these instructions have been

used in existing software, problems could arise when that software is run on the 65C02.

With such difficulties, then, why bother to substitute the new 65C02 into an existing Apple? The answers are varied.

First of all, the 65C02 is likely to appear in upcoming releases of existing computers (in a new release of the Apple //e, perhaps?), and as such you can experiment now with the newest version of this versatile device.

Second, there likely will be specific applications where the advantages of the chip will make it worth supplying with the software, since the disadvantages are practically nonexistent for the Apple //e and Apple ///. Code rewritten to take advantage of the new instructions can be expected to be 10 to 15 percent smaller and run proportionally faster. In certain applications, even greater improvements could be expected.

At this writing, the Rockwell chip seems to have the largest set of instructions of the three varieties available. The GTE and NCR chips lack the bit-manipulation instructions but are otherwise identical.

As to assemblers supporting the instructions, the current version of *Merlin* supports all the new opcodes in both the assembly and *Sourceror* portions of the product. S-C Software is offering a 65C02 cross-assembler to registered owners of the *S-C Assembler* at a reduced rate. Hayden will be offering an update to *ORCA* to support the GTE version of the chip. Any assembler that supports macro capabilities should be able to be used immediately by defining the proper hex codes.

A note from Roger Wagner, June 1983:

This installment marks the last in this series. I want to thank the many readers of this column over the last several years for their enthusiastic support and valuable suggestions. I have always believed that the human element to this industry, and in fact any endeavor, is the truly rewarding part. I would also like to thank *Softalk* for giving me the opportunity to share the excitement of programming with its readers, and also thank Brian Britt for his help in researching this month's article.

For better or worse, though, you're not likely to be completely rid of me. There are rumors of other columns and projects, and I look forward to being a small part of the *Softalk* family for some years to come.

A note from *Softalk* editor Margot Comstock Tommervik, June 1983:

It was nearly three years ago that Roger Wagner's *Assembly Lines* began appearing in *Softalk*; the magazine was only one month old. In that first year, Wagner's column elicited more mail from *Softalk*'s readers than any other feature, and properly so: It was the first time assembly language had been explained from step one. In fact, in his first column, Wagner didn't even introduce a command.

With this issue, Roger Wagner's *Assembly Lines* ends. The first year's columns plus appendixes and revisions have been available for some time in *Assembly Lines: The Book*. Volume 2, covering the rest of the columns, will be released shortly by Softalk Books.

Roger Wagner will not fade away. He's planning occasional feature articles for *Softalk* and he's promised to remain available to answer questions from *Softalk* readers.

Appendix A: Contest

In the March 1981 edition of *Softalk* magazine, we challenged the readers of the “Assembly Lines” column to a contest. Using the commands discussed in the column from October 1980 through March 1981 (all material covered through chapter six in this book), contestants were asked to submit programs which would be judged by the staff, the shortest and most interesting program being the winner. Contest rules are reprinted here as they originally appeared in the March issue of *Softalk*.

Contest Rules

Create the shortest possible program using all and only the commands presented thus far in this series that does something interesting. The program must be entirely in machine language, and may not call any routines in Integer or Applesoft. It may call any of the Monitor routines from \$F800-\$FFFF.

The person who submits the shortest program of the most interest will be awarded \$50 worth of product from any advertiser in this issue of *Softalk* and the program will be published in *Softalk*.

Judging will be based on the opinions of a rather subjectively selected panel made up of people at *Softalk*, myself, and any other hapless passersby we can rope into this thing. Members of the staffs of *Softalk* and Southwestern Data Systems and professional programmers are not eligible to win. Entries should be submitted no later than April 15, 1981. Ties will be settled by Apple’s random number generator. (I promise not to seed it!)

Contest results were announced in the June 1981 edition of *Softalk*. The winning program for the contest is listed below. The commentary accompanies the listing.

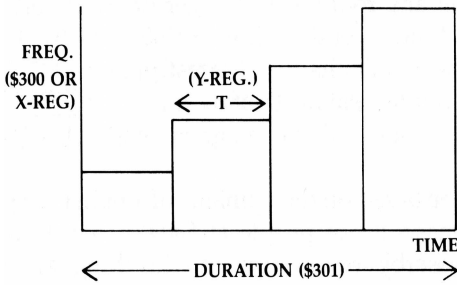
Contest Results

With the usual comments in mind about how hard it was to decide on a winner, I hereby announce the winner of the contest as Steven Morris, of Queens, New York. His program combines a number of the principles we’ve discussed so far and also shows some nice touches in programming. It’s an elegant use of all the given codes. Of particular interest is a self-modifying part wherein the program actually rewrites a small portion of itself upon user command.

I think it will be of interest, and also a good review, to go through Morris's listing to see what's been done. Before doing that, however, a little background on one more kind of tone routine is in order. This will make Morris's program that much more understandable.

In chapter eight, I discussed simple tone routines in which the speaker was accessed at a constant rate for a given length of time. These two factors determined the pitch and duration of the tone played. A variation on this is to have the pitch decrease or increase as the tone is played, creating effects rather like the

sound usually associated with a falling bomb or a rising siren, respectively. This requires three variables, and without getting too technical, let me take a moment to illustrate with the chart at left.



The vertical axis represents the frequency of the tone being played. Putting several tones together into a series over a period of time creates, in this case, a rising scale. As each tone is played,

the pitch is increased. Each individual tone lasts some arbitrary time, T, and put together, the series lasts an overall time period, labeled here as DURATION.

If the pitch is decreased by a certain amount each time, the pattern is reversed. This is sometimes called a *ramp* tone pattern. In parentheses, I have indicated how each of these values is determined in Morris's program.

Here is a listing of the program:

```

1 *****
2 * ASSEMBLY LINES CONTEST WINNER*
3 * BY STEVEN MORRIS *
4 *****
5 * OBJ $302
6 * ORG $302
7 *
8 PTCH EQU $300
9 DRTN EQU $301
10 SPKR EQU $C030
11 PREAD EQU $FB1E
12 PB0 EQU $C061
13 PB1 EQU $C062
14 GRSW EQU $C050
15 TXTSW EQU $C051
16 CLRSCR EQU $F832
17 *
0302: CA 18 LOOP DEX ; DEC THIS DELAY
0303: D0 06 19 BNE CYCLE ; DONE? NO = SKIP CLK
20 *
```

```

0305: AE 00 03 21 CLK      LDX PTCH      ; REFRESH X-REG
0308: AD 30 C0 22          LDA SPKR      ; CLK SPKR
                23 * SPKR CLKS ONLY ONCE
                24 * FOR EVERY ($300) PASSES
                25 *
030B: 88          26 CYCLE    DEY          ; # OF CYCLE CTR.
030C: D0 F4      27          BNE LOOP    ; DONE?
                28 *              NO = KEEP GOING
030E: CE 01 03 29          DEC DRTN     ;
0311: F0 06      30          BEQ CHKPDL ; DONE W/ RAMP?
                31 *              YES = CHK PDL S
0313: EE 00 03 32 RAMP     INC PTCH     ;
0316: 4C 02 03 33          JMP LOOP    ;
                34 *
0319: A2 00      35 CHKPDL   LDX #$00    ;
031B: 20 1E FB 36          JSR PREAD  ; READ PDL(0)
031E: 8C 00 03 37          STY PTCH   ; SET PTCH
0321: E8          38          INX
0322: 20 1E FB 39          JSR PREAD  ; READ PDL(1)
0325: 8C 01 03 40          STY DRTN   ; SET DRTN
0328: A0 7F      41          LDY #$7F
032A: CC 62 C0 42          CPY PB1    ; #1 PRESSED?
032D: 90 27      43          BCC TOGGLE ; BRCH IF YES
                44 *
032F: C8          45          INY          ; #$7F -> #$80; AN EXCUSE
0330: 98          46          TYA          ; TO USE THESE
0331: AA          47          TAX          ; COMMANDS.
0332: EC 61 C0 48          CPX PB0    ; #0 PRESSED?
0335: B0 CB      49          BCS LOOP    ; BRCH IF NO
                50 *
0337: 20 32 F8 51 SCREEN   JSR CLRSCR  ; CLR TOBLK
033A: 8D 50 C0 52 S1      STA GRSW    ; SHOW GRAPHICS MODE
033D: 8D 51 C0 53          STA TXTSW  ; SHOW TEXT MODE
0340: 4C 3A 03 54          JMP S1
                55 *
0343: A8          56 SETDEC   TAY          ; USE UP THIS CODE
0344: A2 CE      57          LDX #$CE    ; OPCODE FOR 'DEC'
0346: 8A          58          TXA
0347: CD 13 03 59          CMP RAMP   ; IS IT 'DEC' NOW?
034A: F0 04      60          BEQ SETINC ; BRCH IF YES.
034C: 8D 13 03 61          STA RAMP   ; NO. MAKE IT 'DEC'
034F: 60          62          RTS
                63 *
0350: A2 EE      64 SETINC   LDX #$EE    ; OPCODE FOR 'INC'
0352: 8E 13 03 65          STX RAMP
0355: 60          66          RTS
                67 *
0356: 20 43 03 68 TOGGLE   JSR SETDEC
0359: 4C 02 03 69          JMP LOOP
                70 *
035C: C0          71          CHK

```

I'll try to explain each part of the program, hopefully with a proper balance of enough detail to jog your memory and enough brevity to keep things reasonably short.

If all of this seems overwhelming, you're trying to read through it too fast. Go back through it slowly, taking your time. Have a nice cup of tea while you're at it.

Remember, we're packing six chapters' worth of subject matter into one program. Don't worry if the fine details of the tone routine escape you. The important part is to make sure that you at least recall the existence and general nature of each individual command used in the program.

To explain the program, the easiest place to start is actually at `CHKPDL`, where the paddles are checked for new values at the end of each ramp series (line 35 at address `$319`). The X-Register is loaded with a `$00` to tell the computer we want to read paddle 0 in the next step, then `JSR` to `$FB1E`. That returns with the Y-Register holding the value of the paddle (`$00` to `$FF`), which is then stored in location `$300`, labeled `PTCH` ("pitch"). The X-Register value is then incremented from `$00` to `$01` on line 38, and paddle 1 read. The returned value is stored at `$301` for the duration value.

If paddle pushbutton 1 is pressed, location `$C062` will hold a number greater than `$7F`. To check for this, the Y-Register is loaded with `$7F` and compared against `$C062`. If `$C062` holds a value greater than `$7F`, the Branch Carry Clear (BCC) will be taken (Y-Register < memory location = carry clear). We'll see what that does later.

If the value is less than `$7F`, program execution will fall through to line 45. Here the `$7F` is increased to `$80` and that value passed to the X-Register via the Accumulator. These steps are here to exercise the `INY`, `TYA`, `TAX` commands, and to allow us to use the `CPX` command next to fulfill the contest requirements. At line 48 the comparison is done. If the X-Register is greater (remember it holds a `$80` here), the button is not pressed and the Branch Carry Set (BCS) will be taken (X-Register > memory location = carry set) that sends us to the main tone loop.

At entry to this loop, the X-Register and the Y-Register hold rather arbitrary values, but the overall theory is that, starting at `CLK` on line 21, the X-Register is loaded with the pitch value and the speaker clicked once. At line 26 the Y-Register is decremented; this is a counter for the length of that pitch value. Jumping back to loop, the net effect is that the program will make n passes through before clicking the speaker once, where n is the pitch value held in `$300`. This creates the delay between clicks needed for a given tone.

The length of that particular tone is determined by the Y-Register. When it reaches a value of `$00`, the `BNE` (Branch Not Equal) fails and the counter for the overall duration is decremented. As long as there's time left (that is, `DRTN > $00`), the next test fails (`BEQ` = Branch if Equal to Zero) and the pitch value is incremented.

Going back to `LOOP` plays this next note until all of the notes in the series have been played. Incrementing pitch gives a descending note pattern. (Recall that the greater the pitch value, the lower the tone played.)

When DRTN does reach 0, the program branches to the paddle check routine that we started in. Let's see what happens when a button is pressed. If button 1 is pressed, the program goes via TOGGLE to SETDEC. This clever section (ignore the TAY) loads the X-Register with the value \$CE. This is the opcode for DEC (DECrement a memory location).

If the comparison fails, that is, there is not a \$CE currently there, the \$CE is stored at RAMP, the RTS (ReTurn from Subroutine) returns to TOGGLE and the JMP loop sends everything back into the tone loop, this time with a DEC PTCH there instead. This gives an ascending pitch series.

If the comparison is true, it means that a \$CE was put there earlier, and the BEQ goes to SETINC, which restores the code for INC at RAMP (\$313), and then returns with the RTS, JMP LOOP as in the previous case.

These two options give the program the ability to rewrite itself, an interesting and powerful idea.

If paddle button 0 is pressed, the branch at line 49 fails and the program falls into an infinite loop at SCREEN (\$337). In this loop, the screen is cleared to the color black by the Monitor routine at \$F832.

Locations \$C050 and \$C051 are soft-switches: accessing these changes the display mode of the Apple. The screen can be viewed either in a text mode or in a graphics mode. Accessing \$C050 on line 52 sets the graphics mode, so the screen appears black. Accessing \$C051 sets the display to text, which appears as inverse "@" signs.

The JMP S1 repeats this cycle back and forth so fast that you don't actually see the flicker, just an interesting pattern created by the screens switching faster than your screen monitor can display them.

At this point you have to hit RESET to end.

There were a number of other excellent entries. Honorable mention should be made of Steve Hawley, Ray Ransom, Stephen Gagola, Jr., and Matt Brookover for their efforts.

Appendix B: Assembly Commands

This section may well serve as the most often-used portion of this book. I have mentioned elsewhere that learning programming can be looked upon as merely familiarizing yourself with the available tools to accomplish a specified task. The following section summarizes the tools available to an assembly-language programmer.

When you are first learning to program, much can be gained simply by browsing through the following pages and casually noting the variety of instructions available when writing a routine. Each entry provides the usual technical data on the instruction and often a brief example of its use as well.

Please note that in some examples a percent sign (%) is used to indicate a binary form of a number. Some assemblers support this delimiter which can be very convenient, particularly when working with the logical operators and shift instructions. For example, the following representations are all equivalent: $100 = \$64 = \%0110\ 0100$.

When looking at addressing modes, it's easy to forget the subtleties of the differences between the X- and Y-Register as used with indirect addressing. Remember that the syntax ($\$FF, X$) indicates pre-indexing, while ($\$FF$), Y indicates post-indexing. See chapter seven for the "official" explanation of addressing modes.

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 binary and BCD (Binary Coded Decimal) modes.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓	✓					✓	✓	✓			

Addressing Modes

Common Syntax

Hex Coding

Immediate	ADC # $\$12$	69 12
Zero Page	ADC $\$12$	65 12
Zero Page,X	ADC $\$12, X$	75 12
Absolute	ADC $\$1234$	6D 34 12
Absolute,X	ADC $\$1234, X$	7D 34 12
Absolute,Y	ADC $\$1234, Y$	79 34 12

Addressing Modes	Common Syntax	Hex Coding
(Indirect),X	ADC (\$12,X)	61 12
(Indirect),Y	ADC (\$12),Y	71 12
(Indirect) [65C02] ¹	ADC (\$12)	72 12

Uses: Peculiarly enough, ADC is most often used to add numbers together. Here are some common examples:

1. Adding a constant to a register or memory location:

```
CLC
LDA MEM
ADC #$80
STA MEM ; MEM = MEM + #$80
```

2. Adding a constant (such as an offset) to a two-byte memory pointer:

```
CLC
LDA MEM
ADC #$80
STA MEM
LDA MEM+1
ADC #$00
STA MEM+1 ; MEM, MEM+1 = MEM, MEM+1 + #$80
```

3. Adding two (2) two-byte values together:

```
CLC
LDA MEM
ADC MEM2
STA MEM
LDA MEM+1
ADC MEM2+1
STA MEM+1 ; MEM, MEM+1 = MEM, MEM+1 + MEM2, MEM2+1
```

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 also ORA.)

AND means that if *both* bits are 1 then the result will be 1, otherwise the result will be 0.

¹ [CT] Opcodes in gray are only available on the 65C02.

Truth Table

	0	1
0	0	0
1	0	1

Example

Accumulator: 0 0 1 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:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓		✓			

Addressing Modes

Common Syntax

Hex Coding

Immediate	AND #\$12	29 12
Zero Page	AND \$12	25 12
Zero Page,X	AND \$12,X	35 12
Absolute	AND \$1234	2D 34 12
Absolute,X	AND \$1234,X	3D 34 12
Absolute,Y	AND \$1234,Y	39 34 12
(Indirect),X	AND (\$12,X)	21 12
(Indirect),Y	AND (\$12),Y	31 12
(Indirect) [65C02]	AND (\$12)	32 12

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 1s in each bit position where data is to be allowed through, and 0s where data is to be suppressed. For example, it is frequently desirable to mask out the high-order bit of ASCII data, such as would come from the keyboard or another input device (perhaps a disk file). The routine shown assures that no matter what value is gotten from the device, the high-order bit of the value in MEM will always be clear:

Code	Example 1	Example 2
LDA DEVICE	01010111	11010111
AND #7F	01111111	01111111
STA MEM	01010111	01010111

AND is also used when you know the high bit will be set and you want it cleared. This is the case when getting ASCII characters from the keyboard. A common routine to get a character from the keyboard is:

```
WATCH LDA KYBD ; $C000
      BPL WATCH ; AGAIN IF < #$80
      BIT STROBE ; CLEAR STROBE: $C010
      AND #$7F ; CLR HIGH BIT
      STA MEM
```

Another way of looking at this same effect is to say that AND can be used to force a 0 in any desired position in a byte's bit pattern. (See ORA to force 1s). A 0 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 AND'd with this mask, a 0 will be forced at each position marked with a 0 in the mask, while all other positions will be unaffected, remaining 0s or 1s, as in their original condition.

The Monitor uses the AND instruction in the GETLN routine (\$FD6A) to convert lowercase letters to uppercase:

```

FD7C- B1 28      807          LDA  (BASL),Y ; GET CHARACTER
FD7E- C9 E0      808 CAPTST CMP  #$E0    ; ALPHA?
FD80- 90 02      809          BCC  ADDINP  ; NO, DON'T XVERT
FD82- 29 DF      810          AND  #$DF    ; XVERT TO CAPS
FD84- 9D 00 02   811 ADDINP STA  IN,X     ; PUT CHAR BACK

```

There are also at least two other 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* of a division operation. For example: 12 MOD 4 = 0; 14 MOD 4 = 2; 18 MOD 4 = 2; 17 MOD 2 = 1; etc.

The general formula is: Accumulator MOD 2^n = 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:

```

LDA  MEM
AND  #$07    ; %00000111 = 2^3-1
STA  MEM    ; MEM = MEM MOD 8

```

This technique provides one of several ways of testing for the odd/even attribute of a number:

```

LDA  MEM
AND  #$01    ; %00000001 = 2^1-1
BEQ  EVEN
BNE  ODD

```

The result of the AND of any number and \$01 will always be either 0 or 1 depending on whether the number was odd or even.

The third application is in determining if a given bit pattern is present among the other data in a number. For example, to test if bits 0, 3 and 7 are on:

```

LDA  MEM
AND  #$89    ; %10001001
CMP  #$89
BEQ  MATCH
BNE  NOMATCH

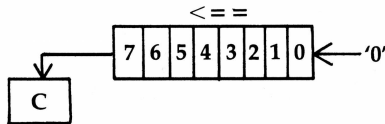
```

The general technique is to first AND the data against the value for the byte with just the desired bits set to 1 (all others 0), and then immediately do a CMP to the same value. If all the specified bits match, a BEQ will succeed.

Note: BIT (described later) can be used to test for one or more matches, but the AND technique described here confirms that *all* of the bits of interest match.

ASL: Arithmetic Shift Left

Description: This instruction moves each bit of the Accumulator or specified memory location one position to the left. A 0 is forced at the bit 0 position, and bit 7 (the high-order bit) falls into the carry. The result is left in the Accumulator or memory location. (See also LSR , ROL, and ROR.)



Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓	✓	✓			✓

Addressing Modes

Common Syntax

Hex Coding

Accumulator	ASL	0A
Zero Page	ASL \$12	06 12
Zero Page,X	ASL \$12,X	16 12
Absolute	ASL \$1234	0E 34 12
Absolute,X	ASL \$1234,X	1E 34 12

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: $123 \times 10 = 1230$ (shift left). For example:

```
LDA MEM
ASL      ; TIMES 2
ASL      ; TIMES 2 AGAIN
STA MEM ; MEM = MEM*4 (4 = 2^2)
```

The other use is to check a given bit position by conditioning the carry flag. For example, checking bit 4 would look like this:

```
LDA MEM
ASL
ASL
```

```

ASL
ASL BIT 4 NOW IN CARRY
BCS SET
BCC NOTSET

```

NOTE: This technique destroys the contents of the Accumulator in the process of checking the bit. AND or BIT instructions are generally preferred instead of this technique.

If testing bits 0 through 3, LSR or ROR may be more appropriate (fewer shifts needed). ROL also can be used instead of ASL depending on whether the data is to be preserved.

BCC: Branch Carry Clear

Description: Executes a branch if the carry flag is clear. Ignored if carry is set. Many assemblers have an equivalent pseudo-op called BLT (Branch Less Than, not to be confused with the sandwich), since BCC is often used immediately following a comparison to see whether the Accumulator is less than the specified value.

Flags & Registers Affected: None

Addressing Modes	Common Syntax	Hex Coding
Relative only	BCC Address	90 FF

Note: The carry flag, upon which this depends, is conditioned by ADC, ASL, CLC, CMP, CPX, CPY, LSR, PLP, ROL, RTI, SBC, and SEC.

Uses: As mentioned, BCC is used to detect when the Accumulator holds a value that is less than a specified value. The usual appearance of the code is listed below. Note that in a two-byte comparison the high-order bytes are checked first.

One-Byte Comparison:

```

ENTRY  LDA  MEM
        CMP  MEM2
        BCC LESS      ; Goes to LESS if MEM < MEM2
        BCS EQ/GRTR

```

Two-Byte Comparison :

```

ENTRY  LDA  MEM+1
        CMP  MEM2+1
        BCC LESS      ; MEM, MEM+1 < MEM2, MEM2+1
        BEQ  CHK2      ; MEM+1 = MEM2+1

```



```

          BCS GRTR      ; MEM, MEM+1 > MEM2, MEM2+1
CHK2    LDA MEM
          CMP MEM2
          BCC LESS      ; MEM, MEM+1 < MEM2, MEM2+1
          BCS EQ/GRTR   ; MEM, MEM+1 >= MEM2, MEM2+1

```

BCS: Branch Carry Set

Description: Executes a branch only if the carry flag is set. Some assemblers support the pseudo-op BGT (“Branch Greater Than”), since this command is used to test whether the Accumulator is equal to or greater than the specified value.

Flags & Registers Affected: None

Addressing Modes	Common Syntax	Hex Coding
Relative only	BCS Address	B0 FF

Note: The carry flag, upon which this depends, is conditioned by ADC, ASL, CLC, CMP, CPX, CPY, LSR, PLP ROL, RTI, SBC, and SEC.

Uses: BCS is used to detect whether the Accumulator is greater than or equal to a specified value. BCS can be combined with BEQ to detect a greater-than relationship. Note that in the two-byte comparison, the high-order bytes are checked first.

One-Byte Comparison:

```

ENTRY   LDA MEM
          CMP MEM2
          BCC LESS      ; Goes to LESS if MEM < MEM2
          BEQ EQUAL     ; Goes to EQUAL if MEM = MEM2
          BCS GREATER   ; Goes to GREATER if MEM > MEM2

```

Two-Byte Comparison:

```

ENTRY   LDA MEM+1
          CMP MEM2+1
          BCC LESS      ; MEM+1 < MEM2+1
          BEQ CHK2      ; MEM+1 = MEM2+1
          BCS GRTR      ; MEM+1 > MEM2+1
CHK2    LDA MEM
          CMP MEM2
          BCC LESS      ; MEM, MEM+1 < MEM2, MEM2+1
          BEQ EQUAL     ; MEM, MEM+1 = MEM2, MEM2+1
          BCS GRTR      ; 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 0. See BCS to see how a comparison for the Accumulator equal to a given value is done.

Flags & Registers Affected: None

Addressing Modes	Common Syntax	Hex Coding
Relative only	BEQ Address	F0 FF

Note: The zero flag, upon which this depends, is conditioned by: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, RTS, SBC, TAX, TAY, TXA, and TYA.

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 was 0. The common classes of these operations are increment and decrement, logical operators, shifts, and register loads. Even easier to remember is the general principle that whenever you've done something that results in 0, chances are good that the Z-flag has been set. Likewise, any nonzero result of an operation is likely to clear the Z-flag. One of the most common instances is when checking an input string for a 0, usually used as a delimiter:

Example:

```

ENTRY  LDA  DEVICE
        BEQ  DONE    ; DATA = 0
WORK   (... )
        JMP  ENTRY
DONE   RTS

```

BIT: compare Accumulator BITs with memory

Description: Performs a logical AND on the bits of the Accumulator and the contents of the memory location. The opposite of the result is stored in the Z-flag. 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:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
M ₇	M ₆					✓					

Addressing Modes

Common Syntax

Hex Coding

Zero Page	BIT \$12	24 12
Absolute	BIT \$1234	2C 34 12
Immediate [65C02]	BIT #\$12	89 12
Zero Page,X [65C02]	BIT \$12,X	34 12
Absolute,X [65C02]	BIT \$1234,X	3C 34 12

Uses: BIT provides a means of testing whether a given bit is *on* in a byte of data.

Important: BIT will indicate only 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 test mask can be held either in the Accumulator (if testing a memory location), or in a memory location (when testing the Accumulator). The mask is created by setting a 1 in each bit position you are interested in, and leaving all remaining positions set to 0.

Examples:

1. Showing the results of the bit operation:

Acc: 10011011
 Mem: 01010101
 Result: 00010001 → 1 → (opposite) → 0 BNE works, BEQ not taken

Status Register:

N	V	—	B	D	I	Z	C
0	1					0	

2. Acc: 10011011
 Mem: 01000100
 Result: 00000000 → 0 → (opposite) → 1 BEQ works, BNE not taken

Status Register:

N	V	—	B	D	I	Z	C
0	1					1	

3. Sample routines:

Test Accumulator for bit 4 on

```
ENTRY LDA  #$10      ; %00010000
      STA  MEM
      LDA  DEVICE
      BIT  MEM
      BNE  MATCH
      BEQ  NOMATCH
```

Test memory for bit 4 on

```
ENTRY LDA  #$10      ; %00010000
      BIT  MEM
      BNE  MATCH
      BEQ  NOMATCH
```

BIT also sets the N- and V-flags, and thus provides a very fast way of testing bits 6 and 7. Since bit 7 is the high-order bit and is frequently used to indicate certain conditions, this can be quite handy. Here is an example of how to watch for a keypress:

```
LOOP  BIT  KYBD      ; $C000
      BPL  LOOP      ; VAL < 128 = NOT PRESS
      BIT  STROBE    ; $C010
DONE  RTS
```

Note 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 example also provides an illustration of a second application of BIT, which is to access a hardware location (often called a *soft-switch*) without damaging the contents of the Accumulator.

BMI: Branch on Minus

Description: Executes the branch only if the N-flag (sign flag) is set. The N-flag is set by any operation producing a result in the range \$80 to \$FF (i.e. high bit set).

Flags & Registers Affected: None

Addressing Modes

Relative only

Common Syntax

BMI Address

Hex Coding

30 FF

Note: The sign flag, upon which this depends, is conditioned by: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, TAX, TAY, TXS, TXA, and TYA.

Uses: BMI is most commonly used to detect negative numbers when signed binary math is used, but is also equally common in testing for a set high bit, such as in watching the keyboard for a keypress. (See also BIT.) For example:

```

LOOP   LDA   KYBD
        BMI  PRESS    ; DATA > $7F
        BPL  LOOP     ; DATA < $80

```

BMI is also useful for terminating a loop that you want to reach 0 and which otherwise will stay out of the \$80 to \$FF range:

```

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.

Flags & Registers Affected: None

Addressing Modes

Relative only

Common Syntax

BNE Address

Hex Coding

D0 FF

Note: The zero flag, upon which this depends, is conditioned by: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, RTS, SBC, TAX, TAY, TXA, and TYA.

Uses: Often used in loops to branch until the counter reaches 0. Also used in data input loops to verify the nonzero nature of the last byte in, as when checking for the end-of-data delimiter.

Examples:

1. Simple loop

```

ENTRY  LDX   #$20    ; WILL COUNT 32 TIMES
LOOP   DEX
        BNE  LOOP    ; UNTIL X = 0
DONE   RTS

```

2. Data input routine

```

ENTRY  LDA   DEVICE
        BNE  CONTINUE
DONE   RTS

```

3. As used in a two-byte increment routine

```

ENTRY   LDA   MEM
        ADC   #$01
        STA   MEM
        BNE   DONE    ; UNLESS MEM = 0
        LDA   MEM+1
        ADC   #$00    ; MEM+1 = MEM+1 + 1
        STA   MEM+1
DONE    RTS

```

BPL: Branch on Plus

Description: Executes the 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). See also BMI.

Flags & Registers Affected: None

Addressing Modes	Common Syntax	Hex Coding
Relative only	BPL Address	10 FF

Note: The sign flag, upon which this depends, is conditioned by: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, TAX, TAY, TXS, TXA, and TYA.

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:

```

LOOP    LDA   KYBD
        BMI   PRESS    ; DATA > $7F
        BPL   LOOP    ; DATA < $80

```

BPL is also useful for terminating a loop that you want to reach 0 and which otherwise will stay out of the \$80 to \$FF range:

```

ENTRY   LDX   $20    ; TO LOOP 33 TIMES
LOOP    DEX
        BMI   DONE    ; WHEN X = $FF
        BPL   LOOP    ; WHILE X > $FF
DONE    RTS

```

BRA: BRanch Always [65C02]

Description: Always executes the branch (65C02 only).

Flags & Registers Affected: None

Addressing Modes	Common Syntax	Hex Coding
Relative only [65C02]	BRA Address	80 FF

Uses: BRA (branch always) is useful for writing relocatable code. Normally, if you had a loop with a JMP back to the top you would make this relocatable by forcing a branch. This would involve setting or clearing a Status Register flag and then issuing the appropriate branch instruction. Instead, you can simply issue BRA without changing the Status Register flags. The only limitation is the maximum branching distance of plus or minus 128 bytes.

Example:

```

8000: A9 12    2    LOOP    LDA  #$12
8002: EA      3      NOP                      ; MORE CODE HERE
8003: 80 FB    4      BRA   LOOP

```

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:

```
0302-   A=A0 X=00 Y=01 P=36 S=E7
```

What actually happens is that the Program Counter, plus two, is saved on the stack, immediately followed by the Status Register, in which the BRK bit has been set. The processor then jumps to the address at \$FFFE, \$FFFF. On the Apple II Plus and Apple //e this routine (at \$FA40) jumps to a vector at \$3F0, \$3F1 which points to the BRK handler routine (at \$FA59) which produces the output.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Addressing Modes	Common Syntax	Hex Coding
Implied only	BRK	00

Uses: BRK can be very useful in debugging assembly-language programs. A BRK is simply inserted into the code at strategic points in the routine. When the program comes to a screeching halt, one can examine the status of various memory locations and registers to see if everything is as you think it should be. This process can be formalized, and hence considerably improved on, by using a software utility called a debugger which allows you to step through a program one instruction at a time. *Munch-A-Bug*, along with others, provides this option. On Integer Apples, a primitive Step and Trace function is provided as part of the Monitor.

BVC: Branch on oVerflow Clear

Description: Executes the 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. The overflow flag also can be cleared with a CLV command.

Flags & Registers Affected: None

Addressing Modes	Common Syntax	Hex Coding
Relative only	BVC Address	50 FF

Note: The overflow flag, upon which this depends, is conditioned by: ADC, BIT, CLV, PLP, RTI, and SBC.

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:

```

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 relocatable code. The advantage is that the carry remains unaffected, thus allowing it to be tested later in the conventional manner.

```

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. Note that there is not a command to specifically set the overflow flag (as would correspond to a SEC command for the carry) but, in the Apple, the instruction BIT \$FF58 is often used to set the overflow flag.

Flags & Registers Affected: None

Addressing Modes

Relative only

Common Syntax

BVS Address

Hex Coding

70 FF

Note: The overflow flag, upon which this depends, is conditioned by: ADC, BIT, CLV, PLP, RTI, and SBC.

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

```

CLC: CLear Carry

Description: Clears the carry bit of the Status Register.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
							0				

Addressing Modes

Implied only

Common Syntax

CLC

Hex Coding

18

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. A CLC also can be used to force a branch when writing relocatable 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:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
				0							

Addressing Modes

Implied only

Common Syntax

CLD

Hex Coding

D8

Uses: The arithmetic mode of the 6502 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: CLear Interrupt mask

Description: This instruction enables interrupts.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
					0						

Addressing Modes

Implied only

Common Syntax

CLI

Hex Coding

58

Uses: CLI tells the 6502 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 6502 until a CLI is re-issued. As a matter of interest, timing-dependent routines like the DOS RWTS (Read/Write Track Sector) routine disable interrupts while on and then allow them again with a CLI at exit.

CLV: CLear oVerflow flag

Description: This clears the V-flag (overflow flag) by setting the V bit of the Status Register to 0.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
	0										

Addressing Modes

Implied only

Common Syntax

CLV

Hex Coding

B8

Uses: Because the overflow flag is, in fact, cleared by a non-overflow result of an ADC instruction, it usually is not necessary to clear the flag prior to an addition. It is, however, occasionally used as a relatively unobtrusive way of forcing a branch when writing relocatable code.

This is done in a manner similar to the CLC, BCC or SEC, BCS pairs discussed in chapter 15. The general form is:

```
CLV
BVC ADDRESS
```

This technique has the advantage of not affecting the carry flag, should the user want to test the carry after the forced break.

CMP: CoMPare to Accumulator

Description: CMP compares the Accumulator to a specified value or memory location. The N-flag (sign flag), Z-flag (zero flag), and C-flag (carry flag) 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:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓	✓				

Addressing Modes	Common Syntax	Hex Coding
Immediate	CMP #\$12	C9 12
Zero Page	CMP \$12	C5 12
Zero Page,X	CMP \$12,X	D5 12
Absolute	CMP \$1234	CD 34 12
Absolute,X	CMP \$1234,X	DD 34 12
Absolute,Y	CMP \$1234,Y	D9 34 12
(Indirect,X)	CMP (\$12,X)	C1 12
(Indirect),Y	CMP (\$12),Y	D1 12
(Indirect) [65C02]	CMP (\$12)	D2 12

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:      Accumulator < value
BCS:      Accumulator ≥ value
BEQ, BCS: Accumulator > value

```

See the section on BCC through BCS for specific examples.

Important: A CMP #\$00 should never be done.² Consider this example:

```

LOOP   DEC   MEM
        LDA   MEM
        CMP   #$00
        BCS   LOOP      ; (ALWAYS TAKEN!)
        BCC   DONE
DONE   RTS

```

Because \$01 through \$FF are greater than \$0, the branch will be taken while MEM is in this range. Since \$0 = \$0, when MEM reaches \$0 the branch will still be taken. Therefore, the example creates an endless loop which will never terminate.

Similarly, if the BCC is done first it will never be taken because there is no value less than 0 to trigger it.

² [CT] This should probably state “it should not be used with BCS or BCC.” It is fine to use CMP #\$00 with BEQ and BNE.

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

Immediate

Zero Page

Absolute

Common Syntax

CPX #\$12

CPX \$12

CPX \$1234

Hex Coding

E0 12

E4 12

EC 34 12

Uses: CPX is used primarily 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 0 and then incremented until it reaches the length of the data stream to be read. For example:

```

ENTRY  LDX  #$00
LOOP   LDA  DATA,X
        JSR  PRINT
        INX
        CPX  #$05
        BCC  LOOP
DONE   RTS
DATA   ASC  "TEST!"

```

For the same reasons discussed under CMP, a CPX #\$00 should not be used.³

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:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓	✓				

³ [CT] Similar to CMP, it is fine to use CPX #\$00 with BEQ and BNE.

Addressing Modes	Common Syntax	Hex Coding
Immediate	CPY #\$12	C0 12
Zero Page	CPY \$12	C4 12
Absolute	CPY \$1234	CC 34 12

Uses: The Y-Register usually is used when reading a stream of data from a zero-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 the first five bytes of an Applesoft program line:

```

ENTRY  LDY  #$00
LOOP   LDA  ($67),Y    ; PROG BEG + Y
        STA  ($06),Y    ; TEMP STORAGE AREA
        INY
        CPY  #$05
        BCC  LOOP      ; LOOP FOR 5 BYTES
DONE   RTS

```

For the same reasons discussed under CMP, a CPY #\$00 should not be used.⁴

DEC: DECrement a memory location

Description: The contents of the specified memory location are decremented by one. If the original contents were equal to \$00, then the result will *wrap around* and give a result of \$FF.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓		✓			✓

Addressing Modes	Common Syntax	Hex Coding
Zero Page	DEC \$12	C6 12
Zero Page,X	DEC \$12,X	D6 12
Absolute	DEC \$1234	CE 34 12
Absolute,X	DEC \$1234,X	DE 34 12
Accumulator [65C02]	DEC A	3A

Uses: DEC usually is used when decrementing a one-byte memory value (such as an offset) or a two-byte memory pointer. Here are the common examples:

⁴ [CT] Similar to CMP, it is fine to use CPY #\$00 with BEQ and BNE.

One-Byte Value:

```

ENTRY  DEC  MEM
DONE   RTS

```

Two-Byte Pointer:

```

ENTRY  DEC  MEM
        LDA  MEM
        CMP  #$FF      ; WRAP-AROUND?
        BNE  DONE      ; NO
        DEC  MEM+1     ; YES: DEC MEM+1
DONE   RTS

```

After the DEC operation, the N- and/or Z-flags often are checked to see if the result was negative or a zero/nonzero value, respectively.

The technique shown for the two-byte DEC operation is not necessarily the most efficient. See the SBC entry for an alternative method.

DEX: DEcrement the X-Register

Description: The X-Register is decremented by one. When the original value is \$00, the result will *wrap around* to give a result of \$FF. See also DEC.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓			✓		

Addressing Modes

Implied only

Common Syntax

DEX

Hex Coding

CA

Uses: DEX often is used in reading a data block via indexed addressing, i.e. Absolute,X. Here is a simple example:

```

ENTRY  LDX  #$05
LOOP   LDA  DATA-1,X
        JSR  PRINT
        DEX
        BNE  LOOP
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 *backwards* from \$05 to \$01. The loop is terminated when the X-Register reaches 0. 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 one. When the original value is \$00, the result will *wrap around* to give a result of \$FF. See also DEC.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓				✓	

Addressing Modes

Implied only

Common Syntax

DEY

Hex Coding

88

Uses: DEY usually is used when decrementing a reverse scan of a data block, using a zero-page pointer via indirect indexed addressing (such as LDA (\$FF), Y). Reverse scans often are 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
LOOP   LDA  #$8D      ; <RETURN>
        JSR  COUT     ; $FDED
        DEY
        BNE  LOOP     ; UNTIL Y=0
DONE   RTS

```

EOR: Exclusive OR with Accumulator

Description: The value in the Accumulator is exclusive OR'd with the specified data. The N-flag (sign flag) and Z-flag (zero flag) are also conditioned depending on the result. The result is put back in the Accumulator. The memory location (if specified) is unaffected.

EOR means that if either bit, but *not both*, is 1 then the result will be 1, otherwise the result will be 0.

Truth Table

	0	1
0	0	1
1	1	0

Example

Accumulator: 0 0 1 1 0 0 1 1
 Memory: 0 1 0 1 0 1 0 1
 Result: 0 1 1 0 0 1 1 0

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓		✓			

Addressing Modes

Common Syntax

Hex Coding

Immediate	EOR #\$12	49 12
Zero Page	EOR \$12	45 12
Zero Page,X	EOR \$12,X	55 12
Absolute	EOR \$1234	4D 34 12
Absolute,X	EOR \$1234,X	5D 34 12
Absolute,Y	EOR \$1234,Y	59 34 12
(Indirect),X	EOR (\$12,X)	41 12
(Indirect),Y	EOR (\$12),Y	51 12
(Indirect) [65C02]	EOR (\$12)	52 12

Uses: EOR has a wide variety of uses:

(1) The most common 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.

```

CODE    LDX  #$05
LOOP    LDA  DATA1,X
        EOR  #$7D      ; ARBITRARY "KEY"
        STA  $300,X    ; REWRITE TABLE
        DEX
        BNE  LOOP      ; UNTIL X=0
DONE    RTS
DATA    ASC  "TEST!"
DECODE  LDX  #$05
LOOP    LDA  $300,X    ; RETRIEVE CODED DATA
        EOR  #$7D
        STA  $380,X    ; PUT IN NEW LOC
        DEX
        BNE  LOOP
DONE    RTS
    
```

(2) Another application is to reverse any given bit or bits of a data byte. The mask is created by putting a one in the positions which you wish to have reversed. A 0 is put in all remaining positions. When the EOR with the mask is

done, bits in the specified positions will *reverse*, i.e. ones will become zeros, and vice versa. See the first example in this entry to verify this effect.

(3) The N-flag (sign flag) can be used to detect if both memory and the Accumulator have bit 7 set:

```
ENTRY   LDA  MEM
        EOR  MEM2
        BMI  MATCH      ; BOTH SET
        BPL  NOMATCH    ; BOTH NOT SET
```

(4) The Z-flag (zero flag) flag will be set if either the Accumulator or memory, or both, equal 0:

```
ENTRY   LDA  MEM
        EOR  MEM2
        BEQ  ZERO      ; MEM=0 AND/OR MEM2=0
        BNE  NOTZ      ; NEITHER MEM NOR MEM2 = 0
```

(5) EOR is also useful in producing the two's complement of a number for use in signed binary arithmetic.

```
ENTRY   LDA  #$34      ; %00110100 = +52
        EOR  #$FF      ; TO BE CONVERTED TO -52
        EOR  #$FF      ; %11111111 = $FF
        CLC
        ADC  #$01      ; RESULT = RESULT + 1
        STA  MEM      ; = %11001100 = $CC
        STA  MEM      ; STORE RESULT
DONE    RTS
```

(5a) And to convert signed negative numbers back:

```
ENTRY   LDA  #$CC      ; %11001100 = $CC = -52
        SEC
        SBC  #$01      ; TO BE CONVERTED BACK
        SBC  #$01      ; ACC = ACC - 1
        EOR  #$FF      ; = %11001011 = $CB
        EOR  #$FF      ; REVERSE ALL BITS
        STA  MEM      ; RESULT = %00110100 = $34 = +52
        STA  MEM      ; STORE RESULT
DONE    RTS
```

INC: INCRement memory

Description: The contents of a specified memory location are incremented by one. If the original value is \$FF, then incrementing will result in a *wrap around* giving a result of \$00. The N-flag (sign flag) and Z-flag (zero flag) are conditioned depending on the result.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓		✓			✓

Addressing Modes

Common Syntax

Hex Coding

Zero Page	INC \$12	E6 12
Zero Page,X	INC \$12,X	F6 12
Absolute	INC \$1234	EE 34 12
Absolute,X	INC \$1234,X	FE 34 12
Accumulator [65C02]	INC A	1A

Uses: INC is used most often for incrementing a one-byte value (such as an off-set) or a two-byte pointer. Here are the most common forms:

One-Byte Value:

ENTRY	INC	MEM
DONE	RTS	

Two-Byte Pointer:

ENTRY	INC	MEM
	BNE	DONE
	INC	MEM+1
DONE	RTS	

After the INC operation, the N- and/or Z-flags often are checked to see whether the result was negative or a zero/nonzero value, respectively.

INX: INcrement the X-Register

Description: The contents of the X-Register are incremented by one. If the original value is \$FF, then incrementing will result in a *wrap around* giving a result of \$00. The N-flag (sign flag) and Z-flag (zero flag) are conditioned depending on the result.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓			✓		

Addressing Modes

Common Syntax

Hex Coding

Implied only	INX	E8
--------------	-----	----

Uses: INX is used in forward-scanning loops which digest a data stream as shown here:

```

ENTRY   LDX  #$00
LOOP    LDA  DATA,X
        BEQ  DONE      ; DELIMITER?
        JSR  COUT      ; $FDED
        INX
        JMP  LOOP      ; NEXT CHAR
DONE    RTS
DATA    ASC  "TEST!"
        HEX  00      ; END OF DATA

```

Note that in forward-scanning loops, the base address can be DATA itself (see DEX for another situation).

INX also can be used as a general-purpose counter for miscellaneous routines:

```

ENTRY   LDX  #$00
        LDA  #$8D      ; <RETURN>
LOOP    JSR  COUT      ; $FDED
        INX
        CPX  #$05
        BCC  LOOP      ; UNTIL X = 5
DONE    RTS           ; PRINTS 5 <CR>S

```

INX: INcrement the Y-Register

Description: The contents of the Y-Register are incremented by one. If the original value is \$FF, then incrementing will result in a *wrap around* giving a result of \$00. The N-flag (sign flag) and Z-flag (zero flag) are conditioned depending on the result.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓				✓	

Addressing Modes

Implied only

Common Syntax

INX

Hex Coding

CB

Uses: INY is used in forward-scanning loops which use the indirect indexed addressing mode, for example LDA (\$FF),Y. This is quite common in routines which process strings for certain characters, search routines, etc. Here is a routine which scans the input buffer for the first carriage return:

```

ENTRY  LDY  #$00
        STA  PTR
        LDA  #$02
        STA  PTR      ; PTR,PTR+1 = $200
        LDY  #$00
LOOP   LDA  (PTR),Y
        CMP  #$8D      ; CHR = <CR>?
        BEQ  FOUND
        INY
        BNE  LOOP      ; UNTIL Y = $00
DONE   RTS
FOUND  STY  MEM
        BEQ  DONE      ; (ALWAYS)

```

JMP: JuMP to address

Description: Causes program execution to jump to the address specified.

Flags & Registers Affected: None

Addressing Modes	Common Syntax	Hex Coding
Absolute	JMP \$1234	4C 34 12
Indirect	JMP (\$1234)	6C 34 12
(Absolute Indirect,X) [65C02]	JMP (\$1234,X)	7C 34 12

Note: The 6502 has a well-documented *bug* regarding the indirect jump.⁵ If the jump specified uses pointers which do not cross a page boundary (for example, \$3C0, \$3C1), then all will go as predicted. If, however, the pointers cross a boundary (such as \$3FF, \$400), then the assumed bytes *will not be used*. Instead, the address data will be retrieved (in our example) from locations \$3FF and \$300. That is to say that the high-order byte is not properly incremented and both bytes are retrieved from the same page of memory. This should be taken into account if such a situation can possibly arise in your routine.

Uses: Besides the obvious application of the usual absolute addressed JMP instruction, the indirect JMP is used when creating vectored jumps. The Apple uses many such indirect jumps, the most notable of which are:

Function	Routine	Jumps to Address at Vector Location
Interrupt Vector	IRQ (\$FA40)	IRQLoc (\$3FE,\$3FF)
Break Vector	BREAK (\$FA4C)	BRKV (\$3F0,\$3F1)
Input Vector	RDKEY (\$FD0C)	KSWL (\$38,\$39)
Output Vector	COU (\$FDED)	CSWL (\$36,\$37)

⁵ [CT] This bug is fixed in the 65C02.

An indirect JMP also can be used when writing relocatable code. If the current location of the code can be determined, then an offset can be calculated and the vectors set up so that the JMP will be relative to the current location of the code. See chapter 15 for more information about these techniques.

JSR: Jump to SubRoutine

Description: The address of the instruction following the JSR is pushed onto the stack. The address following the JSR is then jumped to. When an RTS in the called subroutine is encountered, a return to the location on the stack (the one after the JSR) is done. This is analogous to a GOSUB in BASIC.

Flags & Registers Affected: None

Addressing Modes	Common Syntax	Hex Coding
Absolute only	JSR \$1234	20 34 12

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 references an address within the code (as opposed to routines external to the program, such as in the Monitor ROM), the code can be executed only at the location for which the code was originally assembled.

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 chapter 15 for more details about both of these topics.

LDA: Load Accumulator

Description: Loads the Accumulator with either the specified value or the contents of the designated memory location. The N-flag (sign flag) and Z-flag (zero flag) are conditioned when a value with the high bit set is loaded, or when a 0 value is loaded.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓		✓			

Addressing Modes	Common Syntax	Hex Coding
Immediate	LDA #\$12	A9 12
Zero Page	LDA \$12	A5 12
Zero Page,X	LDA \$12,X	B5 12
Absolute	LDA \$1234	AD 34 12
Absolute,X	LDA \$1234,X	BD 34 12
Absolute,Y	LDA \$1234,Y	B9 34 12
(Indirect,X)	LDA (\$12,X)	A1 12
(Indirect),Y	LDA (\$12),Y	B1 12
(Indirect) [65C02]	LDA (\$12)	B2 12

Uses: LDA is probably *the most used* instruction. The vast majority of operations center around the Accumulator, and this instruction is used to get data into this important register.

LDX: LoaD the X-Register

Description: Loads the X-Register with either the specified value or the contents of the designated memory location. The N-flag (sign flag) and Z-flag (zero flag) are conditioned when a value is loaded that has the high bit set, or when a 0 value is loaded.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓			✓		

Addressing Modes	Common Syntax	Hex Coding
Immediate	LDX #\$12	A2 12
Zero Page	LDX \$12	A6 12
Zero Page,Y	LDX \$12,Y	B6 12
Absolute	LDX \$1234	AE 34 12
Absolute,Y	LDX \$1234,Y	BE 34 12

Uses: This is the primary way in which data is placed into the X-Register. What more can I say?

LDY: Load the Y-Register

Description: Loads the Y-Register with either the specified value or the contents of the designated memory location. The N-flag (sign flag) and Z-flag (zero flag) are conditioned when a value with the high bit set is loaded, or when a 0 value is loaded.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓				✓	

Addressing Modes

Immediate

Zero Page

Zero Page,X

Absolute

Absolute,X

Common Syntax

LDY #\$12

LDY \$12

LDY \$12,X

LDY \$1234

LDY \$1234,X

Hex Coding

A0 12

A4 12

B4 12

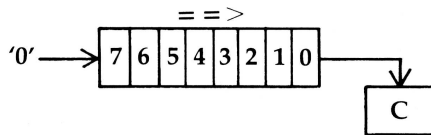
AC 34 12

BC 34 12

Uses: This is the primary way in which data is placed into the Y-Register. See LDX for additional comments.

LSR: Logical Shift Right

Description: This instruction moves each bit of the Accumulator or specified memory location one position to the right. A 0 is forced at the bit 7 position (the high-order bit), and bit 0 falls into the carry. The result is left in the Accumulator or memory location. (See also ASL, ROL, and ROR.)

**Flags & Registers Affected:**

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
∅						✓	✓	✓			✓

Addressing Modes

Accumulator

Zero Page

Common Syntax

LSR

LSR \$12

Hex Coding

4A

46 12

Addressing Modes	Common Syntax	Hex Coding
Zero Page,X	LSR \$12,X	56 12
Absolute	LSR \$1234	4E 34 12
Absolute,X	LSR \$1234,X	5E 34 12

Uses: LSR provides an easy way of dividing by two. The corresponding effect in decimal arithmetic is well known: $123/10 = 12.3$ (shift right). As an example:

```

ENTRY   LDA  MEM
        LSR          ; DIV BY 2
        LSR          ; DIV BY 2 AGAIN
        STA  MEM     ; MEM = MEM / 4

```

LSR also provides a fast way of detecting whether a number is odd or even:

```

ENTRY   LDA  MEM
        LSR
        BCS  ODD
        BCC  EVEN

```

Because 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.

NOP: No Operation

Description: Does nothing for one instruction (two cycles). May remind you of some people you know.

Flags & Registers Affected: None

Addressing Modes	Common Syntax	Hex Coding
Implied only	NOP	EA

Uses: NOP is used primarily to disable portions of code written by other programmers that you have decided you can live without. A classic example of this is the placing of three NOPs at bytes \$D3, \$D4, and \$D5 on Track 0, Sector 9, of a standard DOS 3.3 diskette. By the strategic placement of these NOPs, a boot will not force a clear of the language card, thus avoiding the rather monotonous LOADING LANGUAGE CARD message on every boot.⁶

Additionally, NOPs may be used during debugging to disable certain steps or to create certain timing periods.

⁶ [CT] You can accomplish the same task by adding three NOPs at \$BFD3:

```
POKE -16427,234: POKE -16428,234: POKE -16429,234
```

and then initializing a disk. When the disk is booted it will not erase the language card.

ORA: Inclusive OR with the Accumulator

Description: This instruction takes each bit of the Accumulator and performs a logical inclusive 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-flag (sign flag) and Z-flag (zero flag) depending on the result. (See also AND and EOR.) ORA means if *either* or *both* bits are 1 then the result is 1. Only when both bits are 0 is the result 0.

Truth Table	0	1	Example
0	0	1	Accumulator: 0 0 1 1 0 0 1 1
1	1	1	Memory: <u>0 1 0 1 0 1 0 1</u>
			Result: 0 1 1 1 0 1 1 1

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓		✓			

Addressing Modes

Common Syntax

Hex Coding

Immediate	ORA #\$12	09 12
Zero Page	ORA \$12	05 12
Zero Page,X	ORA \$12,X	15 12
Absolute	ORA \$1234	0D 34 12
Absolute,X	ORA \$1234,X	1D 34 12
Absolute,Y	ORA \$1234,Y	19 34 12
(Indirect),X	ORA (\$12,X)	01 12
(Indirect),Y	ORA (\$12),Y	11 12
(Indirect) [65C02]	ORA (\$12)	12 12

Uses: ORA is used primarily as a mask to force 1s in specified bit positions. (See AND to force 0s.) To create the mask, a 1 is put in each bit position which is to be forced. 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 COUT:

```
ENTRY   LDA  DEVICE
        ORA  #$80      ; %10000000, SET HIGH BIT
        JSR  COUT      ; $FDED
```

ORA also can be used to convert uppercase characters to lowercase:

```
ENTRY   LDA  CHAR      ; GET CHARACTER
        CMP  #$C1      ; (A) IS IT ALPHABETIC?
        BCC  DONE      ; NO, DON'T CONVERT
        CMP  #$E0      ; IS IT ALREADY LOWERCASE?
        BCS  DONE      ; YES, DON'T CONVERT
        ORA  #$20      ; UPPERCASE TO LOWERCASE
        STA  CHAR      ; PUT CHARACTER BACK
```

PHA: PusH Accumulator

Description: This pushes the contents of the Accumulator onto the stack. The Accumulator and Status Register are unaffected. (See also PLA.)

Flags & Registers Affected: None

Addressing Modes	Common Syntax	Hex Coding
Implied only	PHA	48

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 off into the weeds, as the saying goes.

Here is an example of a simple store/retrieve operation:

```

ENTRY  LDA  #$80      ; TEST VALUE
        PHA          ; STORE IT
        LDA  #$FF      ; DESTROY ACC.
        PLA          ; RETRIEVE VALUE
        STA  MEM      ; SAVE IT TO LOOK AT
DONE   RTS

```

Another more obscure use of PHA is to set up an artificial JMP by executing a n RTS for which a JSR was never done. Providing that two PHAs have been done prior to the RTS, the pseudo-jump will be executed. See chapter 15 for more details about this.

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 affected.

Flags & Registers Affected: None

Addressing Modes	Common Syntax	Hex Coding
Implied only	PHP	08

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

```

As with the PHA instruction, PHP always should be accompanied by an equal number of PLP instructions to keep the Apple happy. Remember: It's not nice to fool the stack!

PHX: PusH X-Register [65C02]

Description: This pushes the contents of the X-Register onto the stack (65C02 only). The X-Register and Status Register are unaffected. (See also PLX.)

Flags & Registers Affected: None

Addressing Modes	Common Syntax	Hex Coding
Implied only [65C02]	PHX	DA

Uses: PHX is useful for temporarily storing the X-Register without having to transfer it to the Accumulator first. It is combined with PLX to retrieve the data. Just like PHA/PLA, each PHX should normally be matched by a PLX (or another pull instruction) later in the routine.

Example: With the 65C02, you can easily save and restore all of the registers using code similar to the following:

```

ENTRY   PHX           ; SAVE X
        PHY           ; SAVE Y
        PHA           ; SAVE A
WORK    NOP           ; YOUR PROGRAM HERE
DONE    PLA           ; GET A
        PLY           ; GET Y
        PLX           ; GET X

```

PHY: PusH Y-Register [65C02]

Description: This pushes the contents of the Y-Register onto the stack (65C02 only). The Y-Register and Status Register are unaffected. (See also PLY.)

Flags & Registers Affected: None

Addressing Modes	Common Syntax	Hex Coding
Implied only [65C02]	PHY	5A

Uses: Just like PHX, PHY is useful for temporarily storing the Y-Register without having to transfer it to the Accumulator first. It is combined with PLY (or another pull instruction) to retrieve the data. See PHX for a usage example.

PLA: Pull Accumulator

Description: This is the converse of the PHA instruction. PLA retrieves one byte from the stack and places it in the Accumulator. This accordingly conditions the N-flag (sign flag) and Z-flag (zero flag), just as though an LDA instruction had been done.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓		✓			

Addressing Modes	Common Syntax	Hex Coding
Implied only	PLA	68

Uses: This is combined with PHA to retrieve data from the stack. See PHA for an example of this.

Additionally, PLA can be used to cancel a current RTS, much like a POP in Applesoft BASIC. To cancel the most recent RTS, two PLAs are required:

```

ENTRY   JSR   LEVEL1
        RTS           ; WOULD EXIT HERE NORMALLY
LEVEL1  LDA   #$00    ; ARBITRARY OPERATION
        PLA
        PLA           ; 'POP' RTS
EXIT    RTS           ; WILL EXIT ENTIRELY HERE

```

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 of the flags are conditioned corresponding to the status of each bit in the byte placed there. The Accumulator and other registers are unaffected. (See PHP.)

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓	✓	✓	✓	✓	✓	✓	✓				

Addressing Modes

Implied only

Common Syntax

PLP

Hex Coding

28

Uses: PLP is used to retrieve the Status Register *after* a PHP has stored the flags on the stack. See PHP for an example.

As with the PHA/PLA set, PLPs always should 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: Pull X-Register [65C02]

Description: PLX retrieves one byte from the stack and places it in the X-Register (65C02 only). This conditions the N-flag (sign flag) and Z-flag (zero flag), just as though a LDX instruction had been done.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓			✓		

Addressing Modes

Implied only [65C02]

Common Syntax

PLX

Hex Coding

FA

Uses: This is combined with PHX to retrieve data from the stack. See PHX for a usage example.

PLY: PuLL Y-Register [65C02]

Description: PLY retrieves one byte from the stack and places it in the Y-Register (65C02 only). This conditions the N-flag (sign flag) and Z-flag (zero flag), just as though a LDY instruction had been done.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓				✓	

Addressing Modes
Common Syntax
Hex Coding

Implied only [65C02]

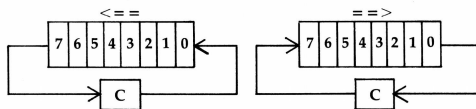
PLY

7A

Uses: This is combined with PHY to retrieve data from the stack. See PHY for details.

ROL: ROTate Left

Description: This instruction moves each bit of the Accumulator or the specified memory location 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-flag (sign flag) and Z-flag (zero flag) are conditioned depending on the result of the shift. (See also ASL, LSR, and ROR.)



ROL – Rotate One Bit Left

ROR – Rotate One Bit Right

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓	✓	✓			✓

Addressing Modes
Common Syntax
Hex Coding

Accumulator

ROL

2A

Zero Page

ROL \$12

26 12

Zero Page,X

ROL \$12,X

36 12

Absolute

ROL \$1234

2E 34 12

Absolute,X

ROL \$1234,X

3E 34 12

Uses: ROL can be used as one of the various methods to test for a set high bit. The disadvantage to testing for the high bit in this way is that the contents must then be restored with a corresponding ROR instruction.

ROR is used more often in combination with ASL in multiply and divide routines.

ROR: ROTate Right

Description: This instruction moves each bit of the Accumulator or the specified memory location one position to the right. The carry bit is pushed into position 7 (the high-order bit), and is replaced by bit 0. The N-flag (sign flag) and Z-flag (zero flag) are also conditioned depending on the result of the shift. (See also ASL, LSR, and ROL.)

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓	✓	✓			✓

Addressing Modes

Accumulator
Zero Page
Zero Page,X
Absolute
Absolute,X

Common Syntax

ROR
ROR \$12
ROR \$12,X
ROR \$1234
ROR \$1234,X

Hex Coding

6A
66 12
76 12
6E 34 12
7E 34 12

Uses: ROR provides an alternate way of testing for the odd/even nature of a number. The carry is tested after the shift to detect whether the number was odd or even.

ROR finds greater use when combined with the shift operations in creating multiply and divide routines.

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 their original values.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓	✓	✓	✓	✓	✓	✓	✓				

Addressing Modes

Implied only

Common Syntax

RTI

Hex Coding

40

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 performed, the return is done via the RTI command. Usually the user will want to restore the Accumulator, the X-Register, and the Y-Register prior to returning.

RTI also is equivalent to a PLP RTS in that the Status Register is restored from the stack and a return is done to the address on the stack.

RTS: ReTurn from Subroutine

Description: This restores the Program Counter to the address stored on the stack, 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 JSR.)

Flags & Registers Affected: None

Addressing Modes

Implied only

Common Syntax

RTS

Hex Coding

60

Uses: RTS is, surprisingly enough, most often used to return from subroutines. On occasion it can be used to simulate a JMP instruction by using two PHA instructions to put a false return address on the stack and then executing the RTS. See the section on PHA, and also chapter 15 for more details.

An RTS can be POP'd one level by the execution of two PLA instructions.

SBC: SuBtract with Carry

Description: Subtracts the contents of the memory location or the 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-flag (sign flag), V-flag (overflow flag), Z-flag (zero flag), and C-flag (carry flag) are all conditioned by this operation, and they often are used to detect the nature of the result of the subtraction. The result of the subtraction is put back in the Accumulator. The memory loca-

tion, if specified, is unchanged. SBC works for both the binary and the BCD arithmetic modes.

Important: An SEC should always be done before the first SBC operation. This is equivalent to clearing the borrow and is analogous to the CLC done before an ADC instruction.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓	✓					✓	✓	✓			

Addressing Modes

Common Syntax

Hex Coding

Immediate	SBC #\$12	E9 12
Zero Page	SBC \$12	E5 12
Zero Page,X	SBC \$12,X	F5 12
Absolute	SBC \$1234	ED 34 12
Absolute,X	SBC \$1234,X	FD 34 12
Absolute,Y	SBC \$1234,Y	F9 34 12
(Indirect),X	SBC (\$12,X)	E1 12
(Indirect),Y	SBC (\$12),Y	F1 12
(Indirect) [65C02]	SBC (\$12)	F2 12

Uses: SBC is used most often for subtracting a constant or memory value from either a one-byte memory location or a two-byte memory location.

One-byte subtraction:

```

ENTRY   SEC
        LDA  MEM
        SBC  #$80
        STA  MEM      ; MEM = MEM - #$80
DONE    RTS

```

Two-byte subtraction:

```

ENTRY   SEC
        LDA  MEM
        SBC  #$80
        STA  MEM
        LDA  MEM+1
        SBC  #$00
        STA  MEM+1    ; MEM, MEM+1 = MEM, MEM+1 - #$80
DONE    RTS

```

SEC: SET Carry

Description: This sets the carry flag of the Status Register.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
							✓				

Addressing Modes

Implied only

Common Syntax

SEC

Hex Coding

38

Uses: SEC usually is used just prior to a SBC operation. The carry is occasionally used though to indicate error (or other) conditions, as is done by RWTS (Read/Write Track Sector) in DOS. In these instances SEC is used to set the carry to indicate an error. This would be detected sometime later in program execution, after a return from RWTS has already been made.

SEC is also sometimes used to force a branch. For example:

```
SEC
BCS ADDRESS ; (ALWAYS)
```

SED: SET Decimal mode

Description: SED sets the 6502 to the Binary Coded Decimal (BCD) mode, in preparation for an ADC or SBC operation.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
				✓							

Addressing Modes

Implied only

Common Syntax

SED

Hex Coding

F8

Uses: BCD math is used when a greater degree of precision is required. In this mode each four bits of a byte represent one digit of a base-ten number. Here is a brief example of a BCD addition operation:

```
ENTRY  SED           ; SET DEC MODE
        CLC
        LDA #$25     ; %00101001 = DECIMAL 25
        ADC #$18     ; %00011000 = DECIMAL 18
```

```

          STA  MEM          ; RSLT = %01000011 = DECIMAL 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 = RESET).

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
					✓						

Addressing Modes

Implied only

Common Syntax

SEI

Hex Coding

78

Uses: SEI is automatically set whenever an interrupt occurs so that no further interrupts can disturb the system while it is going through the vector path from \$FFFE, \$FFFF to \$3FE, \$3FF. The user is expected to use CLI to re-enable interrupts upon entry to his or her own interrupt routine. DOS typically does a SEI/CLI operation upon entrance to and exit from RWTS so that interrupts do not interfere with the highly timing-dependent disk read/write routines.

STA: STore Accumulator

Description: Stores the contents of the Accumulator in the specified memory location. The contents of the Accumulator are not changed, nor are any of the Status Register flags.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
											✓

Addressing Modes	Common Syntax	Hex Coding
Zero Page	STA \$12	85 12
Zero Page,X	STA \$12,X	95 12
Absolute	STA \$1234	8D 34 12
Absolute,X	STA \$1234,X	9D 34 12
Absolute,Y	STA \$1234,Y	99 34 12
(Indirect,X)	STA (\$12,X)	81 12
(Indirect,Y)	STA (\$12),Y	91 12
(Indirect) [65C02]	STA (\$12)	92 12

Uses: STA is another frequently used instruction, being employed at the end of many operations to put the final result into a specified memory location.

In general, the LDA/STA combination is used to transfer bytes from one location to another.

STX: STore the X-Register

Description: STX 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.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Addressing Modes	Common Syntax	Hex Coding
Zero Page	STX \$12	86 12
Zero Page,Y	STX \$12,Y	96 12
Absolute	STX \$1234	8E 34 12

Uses: It is occasionally useful to be able to store the contents of the X-Register for later reference. Another fairly common use of STX is in Applesoft's determination of string lengths. After getting data from the input buffer (\$200 to \$2FF) the length of the input string is held in the X-Register and is saved to a string *descriptor* for later use. See chapter 13 for a listing of a simple input routine.

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.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
											✓

Addressing Modes**Common Syntax****Hex Coding**

Zero Page

STY \$12

84 12

Zero Page,X

STY \$12,X

94 12

Absolute

STY \$1234

8C 34 12

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           ; CHR = 0 = END
        CMP   #$20           ; 'SPC'
        BCS   NXT           ; CHR > CTRL'S
FOUND   STY   POS           ; SAVE Y-REG
DONE    RTS
NXT     INY                   ; Y = Y + 1
        BNE   LOOP          ; UNTIL Y = 0 AGAIN
        BEQ   DONE
NOTF    LDY   #$FF           ; FLAG NOT FOUND
        BNE   FOUND

```

STZ: STore Zero in memory [65C02]

Description: STZ stores a 0 in a zero-page memory location (65C02 only). None of the Status Register flags are affected.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
											✓

Addressing Modes	Common Syntax	Hex Coding
Zero Page [65C02]	STZ \$12	64 12
Zero Page,X [65C02]	STZ \$12,X	74 12
Absolute [65C02]	STZ \$1234	9C 34 12
Absolute,X [65C02]	STZ \$1234,X	9E 34 12

Uses: STZ is used to store a 0 in a memory location. Using STZ avoids having to load a 0 in the Accumulator just to set a memory location.

TAX: Transfer Accumulator to X-Register

Description: Puts the contents of the Accumulator into the X-Register. TAX does not affect the Accumulator.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓			✓		

Addressing Modes	Common Syntax	Hex Coding
Implied only	TAX	AA

Uses: Most simply, TAX is used for transferring data from the Accumulator to the X-Register. Equally important, however, is its combination with TYA to transfer data from the Y-Register to the X-Register:

```
ENTRY  LDY  #$00      ; LOAD Y
        TYA                ; PUT IN A
        TAX                ; PUT IN X
```

TAY: Transfer Accumulator to Y-Register

Description: Puts the contents of the Accumulator into the Y-Register. TAY does not affect the Accumulator.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓				✓	

Addressing Modes	Common Syntax	Hex Coding
Implied only	TAY	A8

Uses: Most simply, TAY is used for transferring data from the Accumulator to the Y-Register. Equally important, however, is its combination with TXA to transfer data from the X-Register to the Y-Register:

```
ENTRY   LDX  #$00      ; LOAD X
         TXA                ; PUT IN A
         TAY                ; PUT IN Y
```

TRB: Test and Reset Bits [65C02]

Description: TRB uses the Accumulator as a mask to clear bits in a specified memory location (65C02 only). The Accumulator is unchanged, but the Z-flag (zero flag) flag is conditioned based on the value of those memory bits prior to the operation.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Addressing Modes	Common Syntax	Hex Coding
Zero Page [65C02]	TRB \$12	14 12
Absolute [65C02]	TRB \$1234	1C 34 12

Uses: TRB is like a combination of BIT and AND, with the added bonus that the new value is stored back in the memory location.

For example, to set both bits 0 and 7 of a memory location, we could use the following set of instructions:

```
LDA  #$81      ; %1000 0001 = MASK PATTERN
TSB  MEM1      ; SET BITS 0,7 OF MEMORY
BNE  PRSET     ; ONE OF THESE WAS 'ON' ALREADY
BEQ  PRCLR     ; NEITHER OF THESE WAS 'ON' ALREADY
```

This would clear the bits:

```
LDA  #$81      ; %1000 0001 = MASK PATTERN
TRB  MEM2      ; CLR BIT 0,7 OF MEMORY
BNE  PRSET     ; ONE OF THESE WAS 'ON' ALREADY
BEQ  PRCLR     ; NEITHER OF THESE WAS 'ON' ALREADY
```

TSB: Test and Set Bits [65C02]

Description: TSB uses the Accumulator as a mask to set bits in a specified memory location (65C02 only). The Accumulator is unchanged, but the Z-flag (zero flag) flag is conditioned based on the value of those memory bits prior to the operation.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
						✓					✓

Addressing Modes
Common Syntax
Hex Coding

Zero Page [65C02]	TSB \$12	04 12
Absolute [65C02]	TSB \$1234	0C 34 12

Uses: TSB is like a combination of BIT and ORA, with the added bonus that the new value is stored back in the memory location. See TRB for an example.

TSX: Transfer Stack to X-Register

Description: This puts the contents of the Stack Pointer into the X-Register. The N-flag (sign flag) and Z-flag (zero flag) are conditioned. The Stack Pointer is unchanged.

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓			✓		

Addressing Modes
Common Syntax
Hex Coding

Implied only	TSX	BA
--------------	-----	----

Uses: The most obvious use of TSX is in preserving the value of the stack at a certain point. Similar to the use of PLAs with RTS, this could be used to duplicate BASIC's POP command—that is to say, a direct return to a different level than the one which had actually called a subroutine. For example:

```

ENTRY   LDA   #$00      ; DUMMY OPERATION
        TSX   ; SAVE CURRENT RETURN PTR
        JSR   LEVEL1
        RTS   ; NORMAL EXIT, BUT IT WILL NEVER BE CALLED
LEVEL1  TXS   ; PUT PTR TO 1ST RETURN BACK
DONE    RTS   ; EXIT TO MAIN CALLING PROGR

```

Note that this is somewhat dangerous in that you must be very certain as to the actual contents of the stack, and in the knowledge that the data has not been changed by intermediate PHAs and PLAs for instance. Remember that the Stack Pointer is only a *pointer* to the stack and does not preserve the return address as such, but only its position in the stack.

Another use for TSX is in retrieving data from the stack without having to do a PLA instruction. Although a PLA/PHA/TAX sequence would be transparent to the stack, and accomplish the same results, TSX can be used to retrieve information that is *officially* lost at that point. What I am alluding to is retrieving data that is lower in memory than the current Stack Pointer, and that would be overwritten by the next PHA instruction. One of the prime examples 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 transparent return address. This is done so that relocatable code has a way of finding out where it's currently located. See chapter 15 for a thorough explanation of the technique. For quick reference, here's the basic routine:

```

ENTRY   JSR  RETURN    ; $FF58
        TSX
        LDA  STACK,X   ; $100,X
        STA  PTR+1
        DEX
        LDA  STACK,X   ; $100,X+1
        STA  PTR       ; PTR,PTR+1 = ENTRY+2
DONE    RTS

```

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. *Caveat emptor!*

TXA: Transfer X to Accumulator

Description: This puts the contents of the X-Register into the Accumulator, and thus conditions the Status Register just as if an LDA had been executed. The X-Register is unaffected by the operation. (See also TAX.)

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓		✓			

Addressing Modes	Common Syntax	Hex Coding
Implied only	TXA	8A

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.

As discussed under TAY, TXA can be combined with TAY to form a TXY-like (transfer X to Y) function like so:

```
ENTRY  LDX  MEM      ; GET DATA
        TXA          ; PUT IN A
        TAY          ; MOVE TO Y
```

TXS: Transfer X to Stack

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.

Flags & Registers Affected: None

Addressing Modes	Common Syntax	Hex Coding
Implied only	TXS	9A

Uses: TXS is used to put data directly into the Stack Pointer. Because there is no TAS (Accumulator to Stack) or even TYS (Y-Register to Stack), this is the only way to get a specific byte into the Stack Pointer. This usually is used in conjunction with TSX to restore previously saved data. In the case of the Applesoft stack-fix program, it is used to avoid problems that otherwise would occur if a RESUME were not used after an error had occurred within a FOR-NEXT loop or a GOSUB:

```
ENTRY  PLA          ; GET LOW BYTE OF CURRENT RETURN ADDR.
        TAY          ; SAVE INTO Y
        PLA          ; GET HIGH BYTE OF RETURN ADDR.
        LDX  ERRSTK ; $DF = S PTR BEFORE ERROR
        TXS          ; PUT BEFORE-ERR PTR BACK
        PHA          ; PUT HIGH BYTE BACK
        TYA          ; GET LOW BYTE IN ACC.
        PHA          ; PUT LOW BYTE BACK.
DONE   RTS          ; RETURN TO APPLESOFT WITH STACK FIXED
```

See also TSX for other applications of TXS.

TYA: Transfer Y to Accumulator

Description: This puts the contents of the Y-Register into the Accumulator, and thus conditions the Status Register just as if an LDA had been executed. The Y-Register is unaffected by the operation. (See also TAY.)

Flags & Registers Affected:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
✓						✓		✓			

Addressing Modes

Implied only

Common Syntax

TYA

Hex Coding

98

Uses: TYA provides a way of retrieving the value in the Y-Register for appropriate processing by the program. This comes in handy in scanning a data block when information regarding 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 and logical operators available.

TYA also is combined with TAX to form the equivalent of a TYX (Transfer Y to X). The operation has the form of:

```
ENTRY  LDY  MEM      ; GET DATA
        TYA                ; PUT IN A
        TAX                ; MOVE TO X
```

Appendix C: 6502 Instruction Set

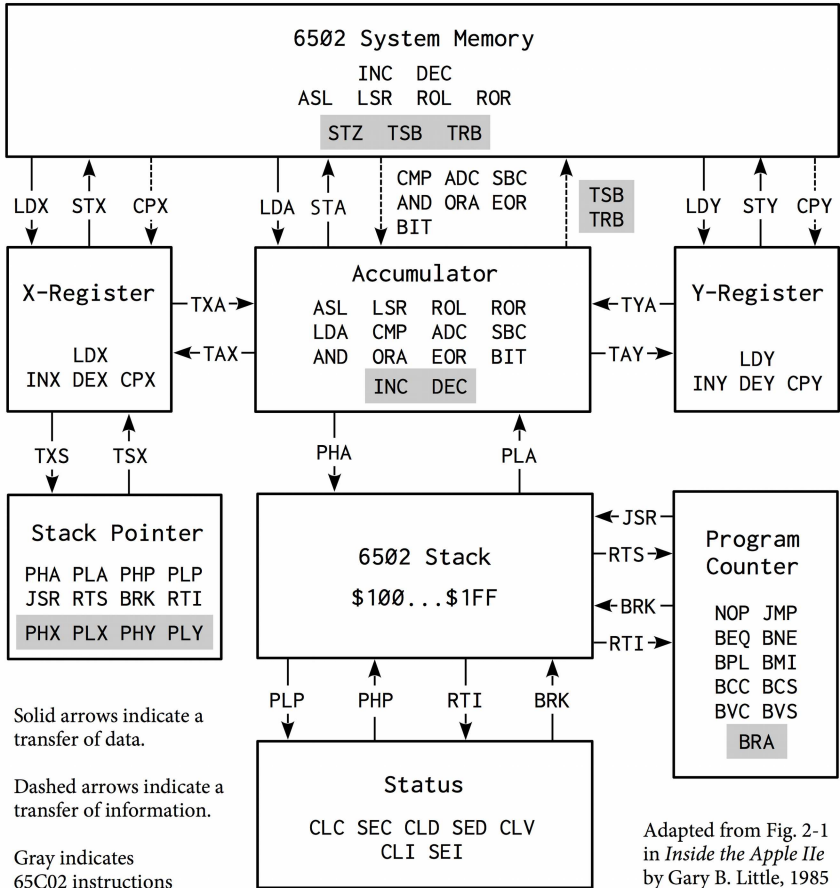
Portions of Appendices C, D, and E are reprinted from the *Apple II Reference Manual*, courtesy Apple Computer, Inc.

6502 Microprocessor Instructions

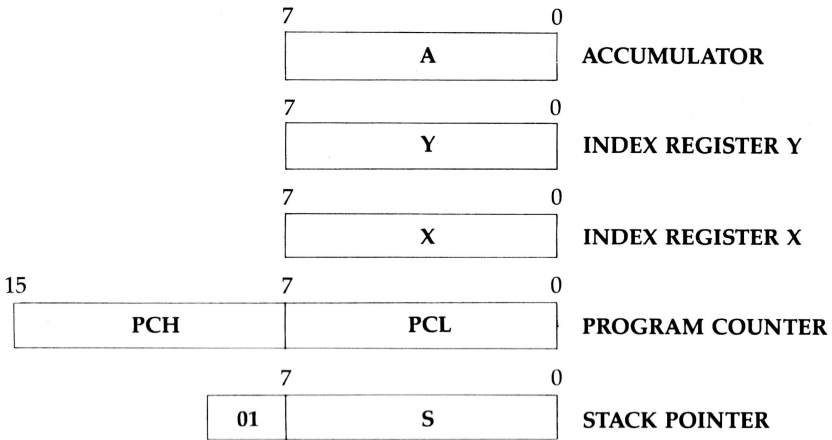
ADC	Add memory to Accumulator with carry	LDX	Load X-Register with memory
AND	AND memory with Accumulator	LDY	Load Y-Register with memory
ASL	Shift left one bit (memory or Accumulator)	LSR	Shift right one bit (memory or Accumulator)
BCC	Branch on carry clear	NOP	No operation
BCS	Branch on carry set	ORA	OR Accumulator with memory
BEQ	Branch on result = zero	PHA	Push Accumulator onto stack
BIT	Test bits in memory with Accumulator	PHP	Push processor status onto stack
BMI	Branch on result = minus	PHX	Push X-Register onto stack
BNE	Branch on result = not zero	PHY	Push Y-Register onto stack
BPL	Branch on result = plus	PLA	Pull Accumulator from stack
BRA	Branch always ¹	PLP	Pull processor status from stack
BRK	Force break	PLX	Pull X-Register from stack
BVC	Branch on overflow clear	PLY	Pull Y-Register from stack
BVS	Branch on overflow set	ROL	Rotate left one bit (memory or Accumulator)
CLC	Clear carry flag	ROR	Rotate right one bit (memory or Accumulator)
CLD	Clear decimal mode	RTI	Return from interrupt
CLI	Clear interrupt disable bit	RTS	Return from subroutine
CLV	Clear overflow flag	SBC	Subtract memory from Accumulator with borrow
CMP	Compare memory and Accumulator	SEC	Set carry flag
CPX	Compare memory and X-Register	SED	Set decimal mode
CPY	Compare memory and Y-Register	SEI	Set interrupt disable status
DEC	Decrement memory by one	STA	Store Accumulator in memory
DEX	Decrement X-Register by one	STX	Store X-Register in memory
DEY	Decrement Y-Register by one	STY	Store Y-Register in memory
EOR	Exclusive OR Accumulator with memory	STZ	Store zero in memory
INC	Increment memory by one	TAX	Transfer Accumulator to X
INX	Increment X-Register by one	TAY	Transfer Accumulator to Y
INY	Increment Y-Register by one	TRB	Test and reset bits
JMP	Jump to new location	TSB	Test and set bits
JSR	Jump to new location saving return address on Stack	TSX	Transfer Stack Pointer to X
LDA	Load Accumulator with memory	TXA	Transfer X to Accumulator
		TXS	Transfer X to Stack Pointer
		TYA	Transfer Y to Accumulator

¹ [CT] Opcodes in gray are for the 65C02.

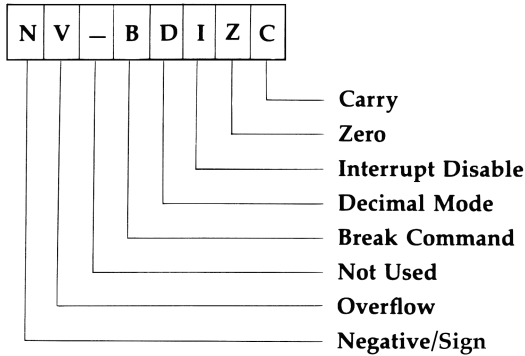
Usage Chart of 6502 Instructions



Programming Model



Processor Status Register



Notation

The following notation applies to the 6502 Instruction Codes table:

A	Accumulator	↑	Transfer from Stack
X, Y	Index Register	↓	Transfer to Stack
M	Memory	→	Transfer to
C	Carry	←	Transfer to
\bar{C}	Borrow	PC	Program Counter
P	Processor Status Register	PCH	Program Counter High
S	Stack Pointer	PCL	Program Counter Low
^	Logical AND	#\$FF	Immediate Addressing Mode
∨	Logical inclusive OR	\$FF	Two-byte (zero page) operand
⋈	Logical exclusive OR	\$FFFF	Four-byte (absolute) operand

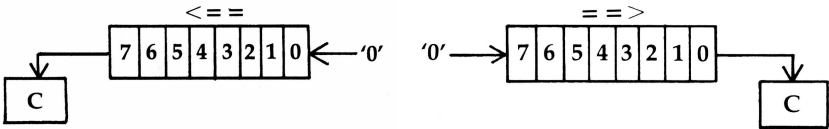


Figure C-1: ASL (shift one bit left) and LSR (shift one bit right)

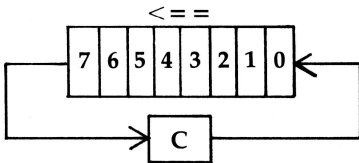


Figure C-2: ROL – Rotate one bit left (memory or Accumulator)

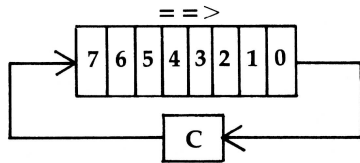


Figure C-3: ROR – Rotate one bit right (memory or Accumulator)

6502 Instruction Codes

The Time is given in clock cycles (1 μ s at 1 MHz). For times with a “+”, add 1 if a page boundary is crossed. For branch instructions with a “*”, add 1 if the branch is taken, and add 1 more if the branch crosses a page boundary. For times with a “d”, add 1 if in decimal mode on the 65C02 (but not on the 6502).

Name Description	Operation	Addressing Mode	Assembly Language	Op- code	Bytes	Time	P status NZCIDV
ADC Add Accumulator to memory with carry	A+M+C \rightarrow A, C	Immediate	ADC # $\$FF$	69	2	2d	NZC---
		Zero Page	ADC $\$FF$	65	2	3d	
		Zero Page,X	ADC $\$FF, X$	75	2	4d	
		Absolute	ADC $\$FFFF$	6D	3	4d	
		Absolute,X	ADC $\$FFFF, X$	7D	3	4d+	
		Absolute,Y	ADC $\$FFFF, Y$	79	3	4d+	
		(Indirect,X)	ADC ($\$FF, X$)	61	2	6d	
		(Indirect,Y)	ADC ($\$FF, Y$)	71	2	5d+	
AND AND Accumulator with memory	A \wedge M \rightarrow A	Immediate	AND # $\$FF$	29	2	2	NZ----
		Zero Page	AND $\$FF$	25	2	3	
		Zero Page,X	AND $\$FF, X$	35	2	4	
		Absolute	AND $\$FFFF$	2D	3	4	
		Absolute,X	AND $\$FFFF, X$	3D	3	4+	
		Absolute,Y	AND $\$FFFF, Y$	39	3	4+	
		(Indirect,X)	AND ($\$FF, X$)	21	2	6	
		(Indirect,Y)	AND ($\$FF, Y$)	31	2	5+	
ASL Shift left one bit (memory or Accumulator)	see Fig C-1	Accumulator	ASL	0A	1	2	NZC---
		Zero Page	ASL $\$FF$	06	2	5	
		Zero Page,X	ASL $\$FF, X$	16	2	6	
		Absolute	ASL $\$FFFF$	0E	3	6	
		Absolute,X	ASL $\$FFFF, X$	1E	3	7 ²	
BCC Branch on carry clear	Branch C=0	Relative	BCC $\$FF$	90	2	2*	-----
BCS Branch on carry set	Branch C=1	Relative	BCS $\$FF$	B0	2	2*	-----
BEQ Branch on result zero	Branch Z=1	Relative	BEQ $\$FF$	F0	2	2*	-----
BIT Test with Accumulator ³ with bits in memory	A \wedge M M ₇ \rightarrow N M ₆ \rightarrow V	Zero Page	BIT $\$FF$	24	2	3	NZ---V
		Absolute	BIT $\$FFFF$	2C	3	4	
BMI Branch on result minus	Branch N=1	Relative	BMI $\$FF$	30	2	2*	-----
BNE Branch on result not zero	Branch Z=0	Relative	BNE $\$FF$	D0	2	2*	-----
BPL Branch on result plus	Branch N=0	Relative	BPL $\$FF$	10	2	2*	-----
BRK Force break interrupt ⁴	PC+2 \downarrow P \downarrow	Implied	BRK	00	1	7	---I--

² [CT] On the 65C02, ASL Abs, X takes 6 cycles if a page boundary is not crossed.

³ Bits 6 and 7 are transferred to the Status Register. If the result of A \wedge M is 0, then Z = 1; otherwise Z = 0.

⁴ A BRK command cannot be masked by setting interrupt disable I.

[CT] On the 6502, BRK does not clear the decimal flag; on the 65C02, it does.

Name Description	Operation	Addressing Mode	Assembly Language	Op- code	Bytes	Time	P status NZCIDV
BVC Branch on overflow clear	Branch V=0	Relative	BVC \$FF	50	2	2*	-----
BVS Branch on overflow set	Branch V=1	Relative	BVS \$FF	70	2	2*	-----
CLC Clear carry flag ⁵	0 → C	Implied	CLC	18	1	2	--C---
CLD Clear decimal mode	0 → D	Implied	CLD	D8	1	2	----D-
CLI Clear interrupt disable	0 → I	Implied	CLI	58	1	2	---I--
CLV Clear overflow flag	0 → V	Implied	CLV	B8	1	2	----V
CMP Compare memory and Accumulator	A ←→ M	Immediate	CMP #\$FF	C9	2	2	NZC---
		Zero Page	CMP \$FF	C5	2	3	
		Zero Page,X	CMP \$FF,X	D5	2	4	
		Absolute	CMP \$FFFF	CD	3	4	
		Absolute,X	CMP \$FFFF,X	DD	3	4+	
		Absolute,Y	CMP \$FFFF,Y	D9	3	4+	
		(Indirect,X)	CMP (\$FF,X)	C1	2	6	
		(Indirect),Y	CMP (\$FF),Y	D1	2	5+	
CPX Compare memory and X- Register	X ←→ M	Immediate	CPX #\$FF	E0	2	2	NZC---
		Zero Page	CPX \$FF	E4	2	3	
		Absolute	CPX \$FFFF	EC	3	4	
CPY Compare memory and Y- Register	Y ←→ M	Immediate	CPY #\$FF	C0	2	2	NZC---
		Zero Page	CPY \$FF	C4	2	3	
		Absolute	CPY \$FFFF	CC	3	4	
DEC Decrement memory by one	M-1 → M	Zero Page	DEC \$FF	C6	2	5	NZ----
		Zero Page,X	DEC \$FF,X	D6	2	6	
		Absolute	DEC \$FFFF	CE	3	6	
		Absolute,X	DEC \$FFFF,X	DE	3	7	
DEX Decrement X by 1	X-1 → X	Implied	DEX	CA	1	2	NZ----
DEY Decrement Y by 1	Y-1 → Y	Implied	DEY	88	1	2	NZ----
EOR Exclusive OR Accumulator with memory	A ∨ M → A	Immediate	EOR #\$FF	49	2	2	NZ----
		Zero Page	EOR \$FF	45	2	3	
		Zero Page,X	EOR \$FF,X	55	2	4	
		Absolute	EOR \$FFFF	4D	3	4	
		Absolute,X	EOR \$FFFF,X	5D	3	4+	
		Absolute,Y	EOR \$FFFF,Y	59	3	4+	
		(Indirect,X)	EOR (\$FF,X)	41	2	6	
		(Indirect),Y	EOR (\$FF),Y	51	2	5+	
INC Increment memory by one	M+1 → M	Zero Page	INC \$FF	E6	2	5	NZ----
		Zero Page,X	INC \$FF,X	F6	2	6	
		Absolute	INC \$FFFF	EE	3	6	
		Absolute,X	INC \$FFFF,X	FE	3	7	
INX Increment X by 1	X+1 → X	Implied	INX	E8	1	2	NZ----
INY Increment Y by 1	Y+1 → Y	Implied	INY	C8	1	2	NZ----
JMP Jump to new location	PC+1 → PCL PC+2 → PCH	Absolute (Indirect)	JMP \$FFFF JMP (\$FFFF)	4C 6C	3 3	3 5/6 ⁶	-----

⁵ [CT] CLC, CLD, and CLV had the wrong Status Register flags.

⁶ [CT] Indirect JMP takes 5 cycles on the 6502 and 6 cycles on the 65C02.

Name Description	Operation	Addressing Mode	Assembly Language	Op- code	Bytes	Time	P status NZCIDV
JSR Jump to new location saving return address	PC+2 ↓ PC+1 → PCL PC+2 → PCH	Absolute	JSR \$FFFF	20	3	6	-----
LDA Load memory into Accumulator	M → A	Immediate Zero Page Zero Page,X Absolute Absolute,X Absolute,Y (Indirect,X) (Indirect),Y	LDA #FFF LDA \$FF LDA \$FF,X LDA \$FFFF LDA \$FFFF,X LDA \$FFFF,Y LDA (\$FF,X) LDA (\$FF),Y	A9 A5 B5 AD BD B9 A1 B1	2 2 2 3 3 3 2 2	2 3 4 4 4+ 4+ 6 5+	NZ----
LDX Load memory into X-Register	M → X	Immediate Zero Page Zero Page,Y Absolute Absolute,Y	LDX #FFF LDX \$FF LDX \$FF,Y LDX \$FFFF LDX \$FFFF,Y	A2 A6 B6 AE BE	2 2 2 3 3	2 3 4 4 4+	NZ----
LDY Load memory into Y-Register	M → Y	Immediate Zero Page Zero Page,X Absolute Absolute,X	LDY #FFF LDY \$FF LDY \$FF,X LDY \$FFFF LDY \$FFFF,X	A0 A4 B4 AC BC	2 2 2 3 3	2 3 4 4 4+	NZ----
LSR Shift right one bit (memory or Accumulator)	see Fig C-1	Accumulator Zero Page Zero Page,X Absolute Absolute,X	LSR LSR \$FF LSR \$FF,X LSR \$FFFF LSR \$FFFF,X	4A 46 56 4E 5E	1 2 2 3 3	2 5 6 6 7 ⁷	NZC---
NOP No operation		Implied	NOP	EA	1	2	-----
ORA Logical OR Accumulator with memory	A ∨ M → A	Immediate Zero Page Zero Page,X Absolute Absolute,X Absolute,Y (Indirect,X) (Indirect),Y	ORA #FFF ORA \$FF ORA \$FF,X ORA \$FFFF ORA \$FFFF,X ORA \$FFFF,Y ORA (\$FF,X) ORA (\$FF),Y	09 05 15 0D 1D 19 01 11	2 2 2 3 3 3 2 2	2 3 4 4 4+ 4+ 6 5+	NZ----
PHA Push Accumulator onto stack	A ↓	Implied	PHA	48	1	3	-----
PHP Push processor status onto stack	P ↓	Implied	PHP	08	1	3	-----
PLA Pull Accumulator from stack	A ↑	Implied	PLA	68	1	4	NZ----
PLP Pull processor status from stack	P ↑	Implied	PLP	28	1	4	from Stack

⁷ [CT] On the 65C02, LSR Abs, X takes 6 cycles if a page boundary is not crossed.

Name Description	Operation	Addressing Mode	Assembly Language	Op- code	Bytes	Time	P status NZCIDV
ROL Rotate one bit left (memory or Accumulator)	see Fig C-2	Accumulator Zero Page Zero Page,X Absolute Absolute,X	ROL ROL \$FF ROL \$FF,X ROL \$FFFF ROL \$FFFF,X	2A 26 36 2E 3E	1 2 2 3 3	2 5 6 6 7 ⁸	NZC---
ROR Rotate one bit right (memory or Accumulator)	see Fig C-3	Accumulator Zero Page Zero Page,X Absolute Absolute,X	ROR ROR \$FF ROR \$FF,X ROR \$FFFF ROR \$FFFF,X	6A 66 76 6E 7E	1 2 2 3 3	2 5 6 6 7 ⁸	NZC---
RTI Return from interrupt	P↑ PC↑	Implied	RTI	40	1	6	from Stack
RTS Return from subroutine	PC↑ PC+1 → PC	Implied	RTS	60	1	6	-----
SBC Subtract memory from Accumulator with borrow	A-M- \bar{C} → A	Immediate Zero Page Zero Page,X Absolute Absolute,X Absolute,Y (Indirect,X) (Indirect),Y	SBC #\$FF SBC \$FF SBC \$FF,X SBC \$FFFF SBC \$FFFF,X SBC \$FFFF,Y SBC (\$FF,X) SBC (\$FF),Y	E9 E5 F5 ED FD F9 E1 F1	2 2 2 3 3 3 2 2	2d 3d 4d 4d 4d+ 4d+ 6d 5d+	NZC--V
SEC Set carry flag	1 → C	Implied	SEC	38	1	2	--C----
SED Set decimal mode	1 → D	Implied	SED	F8	1	2	----D-
SEI Set interrupt disable	1 → I	Implied	SEI	78	1	2	---I--
STA Store Accumulator in memory	A → M	Zero Page Zero Page,X Absolute Absolute,X Absolute,Y (Indirect,X) (Indirect),Y	STA \$FF STA \$FF,X STA \$FFFF STA \$FFFF,X STA \$FFFF,Y STA (\$FF,X) STA (\$FF),Y	85 95 8D 9D 99 81 91	2 2 3 3 3 2 2	3 4 4 5 5 6 6	-----
STX Store X-Register in memory	X → M	Zero Page Zero Page,Y Absolute	STX \$FF STX \$FF,Y STX \$FFFF	86 96 8E	2 2 3	3 4 4	-----
STY Store Y-Register in memory	Y → M	Zero Page Zero Page,X Absolute	STY \$FF STY \$FF,X STY \$FFFF	84 94 8C	2 2 3	3 4 4	-----
TAX Transfer A to X	A → X	Implied	TAX	AA	1	2	NZ----
TAY Transfer A to Y	A → Y	Implied	TAY	A8	1	2	NZ----
TSX Transfer stack to X	S → X	Implied	TSX	BA	1	2	NZ----
TXA Transfer X to A	X → A	Implied	TXA	8A	1	2	NZ----
TXS Transfer X to stack	X → S	Implied	TXS	9A	1	2	-----
TYA Transfer Y to A	Y → A	Implied	TYA	98	1	2	NZ----

⁸ [CT] On the 65C02, ROL/ROR Abs, X take 6 cycles if a page boundary is not crossed.

65C02 Instruction Codes

The Time is given in clock cycles (1 μ s at 1 MHz). For times with a “+”, add 1 if a page boundary is crossed. For times with a “d”, add 1 if in decimal mode. This table does not include the bit-manipulation instructions BBR, BBS, RMB, and SMB, which are only available on the Rockwell and WDC chips.

Name Description	Operation	Addressing Mode	Assembly Language	Op- code	Bytes	Time	“P” status NZCIDV
ADC Add Accumulator to memory with carry	A+M+C→A,C	(Indirect)	ADC (\$FF)	72	2	5d	NZC---
AND AND Accumulator with memory	A∧M→A	(Indirect)	AND (\$FF)	32	2	5	NZ----
BIT Test Accumulator with bits in memory	A∧M M ₇ →N M ₆ →V	Immediate	BIT #\$FF	89	2	2	-Z----
		Zero Page,X	BIT \$FF,X	34	2	4	NZ---V
		Absolute,X	BIT \$FFFF,X	3C	3	4+	NZ---V
BRA Branch always	Branch	Relative	BRA \$FF	80	2	3+	-----
CMP Compare memory and Accumulator	A←M	(Indirect)	CMP (\$FF)	D2	2	5	NZC---
DEC Decrement A	M-1→M	Accumulator	DEC	3A	1	2	NZ----
EOR Exclusive OR Accumulator with memory	A∨M→A	(Indirect)	EOR (\$FF)	52	2	5	NZ----
INC Increment A	M+1→M	Accumulator	INC	1A	1	2	NZ----
JMP Jump to new location	PC+1→PCL PC+2→PCH	(Absolute Indirect,X)	JMP (\$FFFF,X)	7C	3	6	-----
LDA Load Accumulator with memory	M→A	(Indirect)	LDA (\$FF)	B2	2	5	NZ----
ORA Logical OR Accumulator with memory	A∨M→A	(Indirect)	ORA (\$FF)	12	2	5	NZ----
PHX Push X onto stack	X↓	Implied	PHX	DA	1	3	-----
PHY Push Y onto stack	Y↓	Implied	PHY	5A	1	3	-----
PLX Pull X from stack	X↑	Implied	PLX	FA	1	4	NZ----
PLY Pull Y from stack	Y↑	Implied	PLY	7A	1	4	NZ----
SBC Subtract memory from A with borrow	A-M-C→A	(Indirect)	SBC (\$FF)	F2	2	5d	NZC--V
STA Store Accumulator in memory	A→M	(Indirect)	STA (\$FF)	92	2	5	-----
STZ Store zero in memory	0→M	Zero Page	STZ \$FF	64	2	3	-----
		Zero Page,X	STZ \$FF,X	74	2	4	
		Absolute	STZ \$FFFF	9C	3	4	
		Absolute,X	STZ \$FFFF,X	9E	3	5	
TRB Test and reset bits	A∧M→M	Zero Page	TRB \$FF	14	2	5	-Z----
		Absolute	TRB \$FFFF	1C	3	6	
TSB Test and set bits	A∨M→M	Zero Page	TSB \$FF	04	2	5	-Z----
		Absolute	TSB \$FFFF	0C	3	6	

Hex Operation Codes

Note: Table entries in gray are opcodes for the 65C02.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	BRK	ORA (zp,x)			TSB zp	ORA zp	ASL zp		PHP	ORA #	ASL A		TSB abs	ORA abs	ASL abs	
1x	BPL rel	ORA (zp),y	ORA (zp)		TRB zp	ORA zp,x	ASL zp,x		CLC	ORA abs,y	INC A		TRB abs	ORA abs,x	ASL abs,x	
2x	JSR abs	AND (zp,x)			BIT zp	AND zp	ROL zp		PLP	AND #	ROL A		BIT abs	AND abs	ROL abs	
3x	BMI rel	AND (zp),y	AND (zp)		BIT zp,x	AND zp,x	ROL zp,x		SEC	AND abs,y	DEC A		BIT abs,x	AND abs,x	ROL abs,x	
4x	RTI	EOR (zp,x)			EOR zp	LSR zp			PHA	EOR #	LSR A		JMP abs	EOR abs	LSR abs	
5x	BVC rel	EOR (zp),y	EOR (zp)		EOR zp,x	LSR zp,x			CLI	EOR abs,y	PHY			EOR abs,x	LSR abs,x	
6x	RTS	ADC (zp,x)			STZ zp	ADC zp	ROR zp		PLA	ADC #	ROR A		JMP (ind)	ADC abs	ROR abs	
7x	BVS rel	ADC (zp),y	ADC (zp)		STZ zp,x	ADC zp,x	ROR zp,x		SEI	ADC abs,y	PLY		JMP (abs,x)	ADC abs,x	ROR abs,x	
8x	BRA rel	STA (zp,x)			STY zp	STA zp	STX zp		DEY	BIT #	TXA		STY abs	STA abs	STX abs	
9x	BCC rel	STA (zp),y	STA (zp)		STY zp,x	STA zp,x	STX zp,y		TYA	STA abs,y	TXS		STZ abs	STA abs,x	STZ abs,x	
Ax	LDY #	LDA (zp,x)	LDX #		LDY zp	LDA zp	LDX zp		TAY	LDA #	TAX		LDY abs	LDA abs	LDX abs	
Bx	BCS rel	LDA (zp),y	LDA (zp)		LDY zp,x	LDA zp,x	LDX zp,y		CLV	LDA abs,y	TSX		LDY abs,x	LDA abs,x	LDX abs,y	
Cx	CPY #	CMP (zp,x)			CPY zp	CMP zp	DEC zp		INY	CMP #	DEX		CPY abs	CMP abs	DEC abs	
Dx	BNE rel	CMP (zp),y	CMP (zp)		CMP zp,x	DEC zp,x			CLD	CMP abs,y	PHX			CMP abs,x	DEC abs,x	
Ex	CPX #	SBC (zp,x)			CPX zp	SBC zp	INC zp		INX	SBC #	NOP		CPX abs	SBC abs	INC abs	
Fx	BEQ rel	SBC (zp),y	SBC (zp)		SBC zp,x	INC zp,x			SED	SBC abs,y	PLX			SBC abs,x	INC abs,x	

Abbreviations

= immediate
A = Accumulator
abs = absolute
rel = relative
zp = zero page
x = X-Register
y = Y-Register

Addressing Modes

abs,x = indexed by X
abs,y = indexed by Y
(abs) = indirect
(abs,x) = indexed absolute indirect
zp,x = indexed by X
zp,y = indexed by Y
(zp) = indirect
(zp,x) = indexed indirect (pre-indexed)
(zp),y = indirect indexed (post-indexed)

Appendix D: Monitor Subroutines

Here is a list of some useful subroutines in the Apple's Monitor and Autostart ROMs. To use these subroutines from assembly-language programs, load the proper memory locations or 6502 registers as required by the subroutine and execute a JSR to the subroutine's starting address. It will perform the function and return with the 6502's registers set as described.

Output Subroutines

\$FDED COUT Output a character

COUT is the standard character output subroutine. The character to be output should be in the Accumulator. COUT calls the current character output subroutine whose address is stored in CSW (locations \$36 and \$37), usually COUT1 (see below).

\$FDF0 COUT1 Output to screen

COUT1 displays the character in the Accumulator on the Apple's screen at the current output cursor position and advances the output cursor. It handles the control characters, <RETURN>, linefeed, and bell. It returns with all registers intact. Characters in the range of \$00 to \$3F come out inverse; characters from \$40 to \$7F are flashing; characters from \$80 to \$FF are normal.

\$FE80 SETINV Set Inverse mode

Sets Inverse video mode for COUT1. All output characters will be displayed as black dots on a white background. The Y-Register is set to \$3F; all others are unchanged.

\$FE84 SETNORM Set Normal Mode

Sets Normal video mode for COUT1. All output characters will be displayed as white dots on a black background. The Y-Register is set to \$FF; all others are unchanged.

\$FD8E CROUT Generate a <RETURN>

CROUT sends a <RETURN> character to the current output device.

\$FD8B CROUT1 <RETURN> with clear

CROUT1 clears the screen from the current cursor position to the edge of the text window, then calls CROUT.

\$FD8A PRBYTE Print a hexadecimal byte

This subroutine outputs the contents of the Accumulator in hexadecimal on the current output device. The contents of the Accumulator are scrambled.

\$FDE3 PRHEX Print a hexadecimal digit

This subroutine outputs the lower nibble of the Accumulator as a single hexadecimal digit. The contents of the Accumulator are scrambled.

\$F941 PRNTAX Print A and X in hexadecimal

This outputs the contents of the Accumulator and X-Register as a four-digit hexadecimal value. The Accumulator contains the first byte output; the X-Register contains the second. The contents of the Accumulator are usually scrambled.

\$F948 PRBLNK Print 3 spaces

Outputs three space characters to the standard output device. Upon exit, the Accumulator usually contains \$A0, the X-Register contains 0.

\$F94A PRBL2 Print many spaces

Outputs from 1 to 256 space characters to the standard output device. Upon entry, the X-Register should contain the number of spaces to be output. If the X-Register is \$00, then PRBL2 will output 256 blanks.

\$FF3A BELL Output a “bell” character

Sends a bell (<CTRL>G) character to the current output device. It leaves the Accumulator holding \$87.

\$FBDD BELL 1 Beep the Apple’s speaker

Beeps the Apple’s speaker for 0.1 second at 1KHz. It scrambles the Accumulator and Y-Register.

Input Subroutines**\$FD0C RDKEY Get an input character**

This is the standard character input subroutine. It places a flashing input cursor on the screen at the current cursor position and jumps to the input subroutine whose address is stored in KSW (\$38, \$39), usually KEYIN (see below).

\$FD35 RDCHAR Get an input character or escape code

RDCHAR is an alternate input subroutine which gets characters from the standard input but also is capable of interpreting the eleven escape codes.

\$FD1B KEYIN Read the Apple’s keyboard

This is the keyboard input subroutine. It reads the Apple’s keyboard, waits for a keypress, and randomizes the random-number seed. When it gets a keypress, it removes the flashing cursor and returns with the key code in the Accumulator.

\$FD6A GETLN Get an input line with prompt

GETLN is the subroutine which gathers input lines. Your programs can call GETLN with the proper prompt character in location \$33; GETLN will return with the

input line in the input buffer (beginning at location \$200) and the X-Register holding the length of the input line.

\$FD67 GETLNZ **Get an input line**

GETLNZ is an alternate entry point for GETLN which issues a <RETURN> to the standard output before falling into GETLN (see above).

\$FD6F GETLN1 **Get an input line, no prompt**

GETLN1 is an alternate entry point for GETLN which does not issue a prompt before it gathers the input line. If, however, the user cancels the input line (either with too many backspaces or with a <CTRL>X), then GETLN1 will issue the contents of location \$33 as a prompt when it gets another line.

Low-Res Graphics Subroutines

\$F864 SETCOL **Set low-res graphics color**

This subroutine sets the color used for plotting on the low-res screen to the color passed in the Accumulator.

\$F85F NEXTCOL **Increment color by 3**

This adds 3 to the current color used for low-res graphics.

\$F800 PLOT **Plot a block on the Low-Res Screen**

This subroutine plots a single block on the low-res screen of the pre-specified color. The block's vertical position is passed in the Accumulator and its horizontal position in the Y-Register. PLOT returns with the Accumulator scrambled, but the X-Register and Y-Register are unmolested.

\$F819 HLINE **Draw a horizontal line of blocks**

This subroutine draws a horizontal line of blocks of the pre-specified color on the low-res screen. You should call HLINE with the vertical coordinate of the line in the Accumulator, the leftmost horizontal coordinate in the Y-Register, and the rightmost horizontal coordinate in location \$2C. HLINE returns with the Accumulator and Y-Register scrambled, but with the X-Register intact.

\$F828 VLINE **Draw a vertical line of blocks**

This subroutine draws a vertical line of blocks of the pre-specified color on the low-res screen. You should call VLINE with the horizontal coordinate of the line in the Y-Register, the top vertical coordinate in the Accumulator, and the bottom vertical coordinate in location \$2D. VLINE returns with the Accumulator scrambled.

\$F832 CLRSCR Clear the entire low-res screen
 CLRSCR clears the entire low-res graphics screen. If you call CLRSCR while the video display is in Text mode, it will fill the screen with inverse-mode “@” characters. CLRSCR destroys the contents of the Accumulator and Y-Register.

\$F836 CLRTOP Clear the top of the low-res Screen
 CLRTOP is the same as CLRSCR (above), except that it clears only the top 40 rows of the screen.

\$F871 SCRN Read the low-res screen
 This subroutine returns the color of a single block on the low-res screen. Call it as you would call PLOT (above). The block’s color value will be returned in the Accumulator. No other registers are changed.

Hi-Res Graphics Subroutines

\$F3E2 HGR Hi-res page 1
 This is the entry point for the HGR command. It initializes hi-res page 1, then clears and displays the screen.

\$F3D8 HGR2 Hi-res page 2
 This is the entry point for the HGR2 command. It initializes hi-res page 2, then clears and displays the screen.

\$F3F2 HCLR Clear to black
 Clears the current screen to black1.

\$F3F6 BKGND Clear to color
 Clears the current screen to the last plotted HCOLOR.

\$F6F0 HCOLOR Set color
 Sets the current HCOLOR to the contents of the X-Register (0–7).

\$F411 HPOSN Position the cursor
 Positions the hi-res “cursor” without plotting. Enter with X, Y (low, high) equal to the horizontal position, and the Accumulator equal to the vertical position.

\$F457 HPLOT Plot at cursor
 Identical to HPOSN, but plots current HCOLOR at coordinates given.

\$F5CB HFIND Return the cursor position
 Returns the current “cursor” position. This is useful after a DRAW to find where you’ve been left. The coordinates are returned in: \$E0, \$E1 = horizontal (low, high), \$E2 = vertical.

\$F53A HLIN Draw a line

This subroutine draws a line from the previous plot to the point given. On input, set A, X (low, high) to the horizontal position, and Y equal to the vertical position.

\$F730 SHNUM Load shape number

This routine puts the address of the shape number indicated by X-Register into \$1A, \$1B. SHNUM returns with X, Y (low, high) also set to address of that shape table entry.

\$F601 DRAW Draw a shape

Draw the shape pointed to by X, Y (low, high) in the current HCOLOR. Note: X, Y point to the specific entry, not the beginning of the table. Be sure to call SHNUM first.

\$F65D XDRAW Erase a shape (draw XOR)

Erases a shape that was just drawn (if there) by doing an exclusive OR with the screen data. On input, load X, Y (low, high) with the address of the shape to XDRAW or call SHNUM first with the X-Register equal to the shape number.

Floating Point Accumulator**\$EBAF ABS Absolute value**

This subroutine takes the absolute value of the Floating Point Accumulator (FAC = \$9D-\$A2).

\$EC23 INT INT function

The INT function uses QINT (\$EBF2) to convert the FAC to integer form and then back to a floating-point number in FAC.

\$EFAE RND Random number

This is the same as the RND command. Produces a (poor quality) pseudo-random number in the FAC.

\$EB82 SIGN Sign of FAC (in Accumulator)

Sets the Accumulator to \$01, \$00, or \$FF if the FAC is positive, zero, or negative.

\$EB90 SGN Sign of FAC (in FAC)

Calls SIGN first, then sets FAC based upon the Accumulator value.

\$EE8D SQR Square root

This is the SQR command. It computes the square root of FAC using a slow exponentiation method: $X^{0.5}$.

\$EF09 EXP Exponentiation

This routine raises e to the FAC power and leaves the result in FAC.

- \$E941** **LOG** **Logarithm base e**
 This computes the logarithm (base e) of FAC.
- \$EE97** **FPWRT** **Raise ARG to the FAC power (base e)**
 This computes ARG to the FAC power using the formula $\text{EXP}(\text{LOG}(\text{ARG}) * \text{FAC})$. Before calling, you should load the Accumulator with FACEXP (\$9D).
- \$EBB2** **FCOMP** **Compare FAC to memory**
 Before calling, load the memory location in the Y-Register and Accumulator. On exit, A = \$01 if the value at the memory location is less than FAC; A = \$00 if the memory equals FAC; A = \$FF if the memory is greater than FAC.
- \$EED0** **NEGOP** **Multiply by -1**
 This routine toggles the sign of FAC.
- \$E7A0** **FADDH** **Add 0.5**
 This routine adds 0.5 to FAC.
- \$EA55** **DIV10** **Divide by 10**
 This routine divides FAC by 10. It returns positive values only.
- \$EA39** **MUL10** **Multiply by 10**
 This routine multiplies FAC by 10. It works on both positive and negative numbers.
- \$EFEA** **COS** **Cosine**
 The cosine function of FAC.
- \$EFFA** **SIN** **Sine**
 The sine function of FAC.
- \$EFF1** **TAN** **Tangent**
 The tangent function of FAC.
- \$F09E** **ATN** **Arctangent**
 The arctangent of FAC.
- \$ED34** **FOUT** **Create a string**
 Create a string at the start of the stack (\$100-\$110) equivalent to the FAC value. On exit the Y-Register and Accumulator point to the string. The string is terminated by a \$00.

Other Subroutines

\$FCA8 WAIT **Delay**

This subroutine delays for a specific amount of time, then returns to the program which called it. The amount of delay is specified by the contents of the Accumulator A. The delay is given by $0.5102 \times (26 + 27A + 5A^2)$ microseconds. WAIT returns with the Accumulator zeroed and the X- and Y-Registers undisturbed.

\$FB1E PREAD **Read a game controller**

PREAD returns a number representing the position of a game controller. You should first pass the number of the game controller (0 to 3) in the X-Register. If this number is not valid, strange things may happen. PREAD returns with a number from \$00 to \$FF in the Y-Register. The Accumulator is scrambled.

\$FF2D PRERR **Print "ERR"**

Sends the word "ERR", followed by a bell character, to the standard output device. The Accumulator is scrambled.

\$FF4A IOSAVE **Save all registers**

The contents of the 6502's internal registers are saved in locations \$45 through \$49 in the order A-X-Y-P-S. The contents of the Accumulator and the X-Register are changed; the decimal mode is cleared.

\$FF3F IOREST **Restore all registers**

The contents of the 6502's internal registers are loaded from locations \$45 through \$49.

Appendix E: ASCII and Screen Charts

You Get What You ASCII For...

This chart shows many of the possible interpretations 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 column shows what key on an Apple II keyboard generates that character, if any.

Although the standard Apple II does not have a lowercase keyboard, lowercase keys are shown to allow for machines with special adapters, external keyboards, etc.

The screen column shows what character is to be expected if that value is stored in the screen memory area, \$400-\$7FF. Inverse characters are surrounded by square brackets [A], while flashing characters are surrounded by angle brackets >A<.

The Applesoft column indicates how Applesoft BASIC interprets that byte when tokenizing programs.

Note that for control characters, the “^” symbol is used. Thus a Control-A would be indicated ^A.

Hex	Dec	Binary	Key	Screen	Applesoft
\$00	0	0000 0000		[@]	^@
\$01	1	0000 0001		[A]	^A
\$02	2	0000 0010		[B]	^B
\$03	3	0000 0011		[C]	^C
\$04	4	0000 0100		[D]	^D
\$05	5	0000 0101		[E]	^E
\$06	6	0000 0110		[F]	^F
\$07	7	0000 0111		[G]	^G
\$08	8	0000 1000		[H]	^H
\$09	9	0000 1001		[I]	^I
\$0A	10	0000 1010		[J]	^J
\$0B	11	0000 1011		[K]	^K
\$0C	12	0000 1100		[L]	^L
\$0D	13	0000 1101		[M]	^M
\$0E	14	0000 1110		[N]	^N
\$0F	15	0000 1111		[O]	^O
\$10	16	0001 0000		[P]	^P
\$11	17	0001 0001		[Q]	^Q
\$12	18	0001 0010		[R]	^R
\$13	19	0001 0011		[S]	^S
\$14	20	0001 0100		[T]	^T
\$15	21	0001 0101		[U]	^U
\$16	22	0001 0110		[V]	^V
\$17	23	0001 0111		[W]	^W
\$18	24	0001 1000		[X]	^X
\$19	25	0001 1001		[Y]	^Y
\$1A	26	0001 1010		[Z]	^Z
\$1B	27	0001 1011		[[]	^[
\$1C	28	0001 1100		[\]	^\
\$1D	29	0001 1101		[]]	^]

Assembly Lines

Hex	Dec	Binary	Key	Screen	Applesoft
\$1E	30	0001 1110		[^]	^^
\$1F	31	0001 1111		[_]	^_
\$20	32	0010 0000		[]	Space
\$21	33	0010 0001		[!]	!
\$22	34	0010 0010		["]	"
\$23	35	0010 0011		[#]	#
\$24	36	0010 0100		[\$]	\$
\$25	37	0010 0101		[%]	%
\$26	38	0010 0110		[&]	&
\$27	39	0010 0111		[']	'
\$28	40	0010 1000		[(]	(
\$29	41	0010 1001		[)])
\$2A	42	0010 1010		[*]	*
\$2B	43	0010 1011		[+]	+
\$2C	44	0010 1100		[,]	,
\$2D	45	0010 1101		[-]	-
\$2E	46	0010 1110		[.]	.
\$2F	47	0010 1111		[/]	/
\$30	48	0011 0000		[0]	0
\$31	49	0011 0001		[1]	1
\$32	50	0011 0010		[2]	2
\$33	51	0011 0011		[3]	3
\$34	52	0011 0100		[4]	4
\$35	53	0011 0101		[5]	5
\$36	54	0011 0110		[6]	6
\$37	55	0011 0111		[7]	7
\$38	56	0011 1000		[8]	8
\$39	57	0011 1001		[9]	9
\$3A	58	0011 1010		[:]	:
\$3B	59	0011 1011		[;]	;
\$3C	60	0011 1100		[<]	<
\$3D	61	0011 1101		[=]	=
\$3E	62	0011 1110		[>]	>
\$3F	63	0011 1111		[?]	?
\$40	64	0100 0000		> @ <	@
\$41	65	0100 0001		> A <	A
\$42	66	0100 0010		> B <	B
\$43	67	0100 0011		> C <	C
\$44	68	0100 0100		> D <	D
\$45	69	0100 0101		> E <	E
\$46	70	0100 0110		> F <	F
\$47	71	0100 0111		> G <	G
\$48	72	0100 1000		> H <	H
\$49	73	0100 1001		> I <	I
\$4A	74	0100 1010		> J <	J
\$4B	75	0100 1011		> K <	K
\$4C	76	0100 1100		> L <	L
\$4D	77	0100 1101		> M <	M
\$4E	78	0100 1110		> N <	N
\$4F	79	0100 1111		> O <	O
\$50	80	0101 0000		> P <	P
\$51	81	0101 0001		> Q <	Q
\$52	82	0101 0010		> R <	R
\$53	83	0101 0011		> S <	S
\$54	84	0101 0100		> T <	T
\$55	85	0101 0101		> U <	U
\$56	86	0101 0110		> V <	V
\$57	87	0101 0111		> W <	W

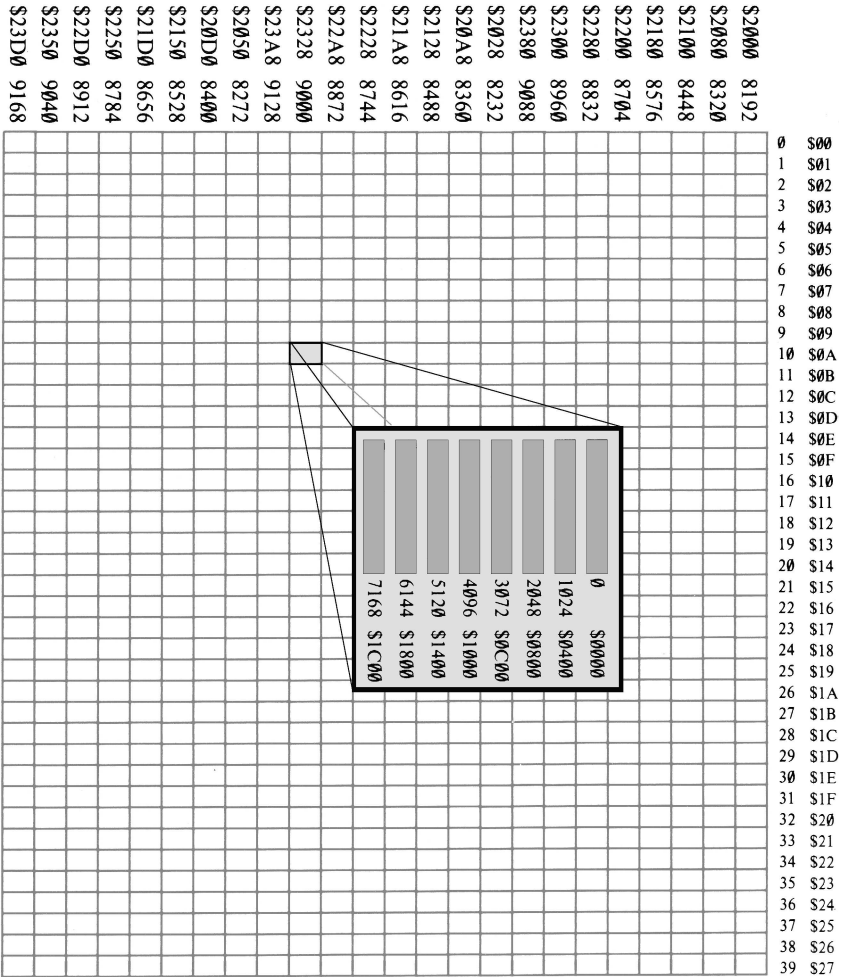
Hex	Dec	Binary	Key	Screen	Applesoft
\$58	88	0101 1000		> X <	X
\$59	89	0101 1001		> Y <	Y
\$5A	90	0101 1010		> Z <	Z
\$5B	91	0101 1011		> [<	[
\$5C	92	0101 1100		> \ <	\
\$5D	93	0101 1101		>] <]
\$5E	94	0101 1110		> ^ <	^
\$5F	95	0101 1111		> _ <	_
\$60	96	0110 0000		> <	`
\$61	97	0110 0001		> ! <	a
\$62	98	0110 0010		> " <	b
\$63	99	0110 0011		> # <	c
\$64	100	0110 0100		> \$ <	d
\$65	101	0110 0101		> % <	e
\$66	102	0110 0110		> & <	f
\$67	103	0110 0111		> ' <	g
\$68	104	0110 1000		> (<	h
\$69	105	0110 1001		>) <	i
\$6A	106	0110 1010		> * <	j
\$6B	107	0110 1011		> + <	k
\$6C	108	0110 1100		> , <	l
\$6D	109	0110 1101		> - <	m
\$6E	110	0110 1110		> . <	n
\$6F	111	0110 1111		> / <	o
\$70	112	0111 0000		> 0 <	p
\$71	113	0111 0001		> 1 <	q
\$72	114	0111 0010		> 2 <	r
\$73	115	0111 0011		> 3 <	s
\$74	116	0111 0100		> 4 <	t
\$75	117	0111 0101		> 5 <	u
\$76	118	0111 0110		> 6 <	v
\$77	119	0111 0111		> 7 <	w
\$78	120	0111 1000		> 8 <	x
\$79	121	0111 1001		> 9 <	y
\$7A	122	0111 1010		> : <	z
\$7B	123	0111 1011		> ; <	{
\$7C	124	0111 1100		> < <	
\$7D	125	0111 1101		> = <	}
\$7E	126	0111 1110		> > <	~
\$7F	127	0111 1111		> ? <	Rubout
\$80	128	1000 0000	^@	@	END
\$81	129	1000 0001	^A	A	FOR
\$82	130	1000 0010	^B	B	NEXT
\$83	131	1000 0011	^C	C	DATA
\$84	132	1000 0100	^D	D	INPUT
\$85	133	1000 0101	^E	E	DEL
\$86	134	1000 0110	^F	F	DIM
\$87	135	1000 0111	^G	G	READ
\$88	136	1000 1000	^H	H	GR
\$89	137	1000 1001	^I	I	TEXT
\$8A	138	1000 1010	^J	J	PR #
\$8B	139	1000 1011	^K	K	IN #
\$8C	140	1000 1100	^L	L	CALL
\$8D	141	1000 1101	^M	M	PLOT
\$8E	142	1000 1110	^N	N	HLIN
\$8F	143	1000 1111	^O	O	VLIN
\$90	144	1001 0000	^P	P	HGR2
\$91	145	1001 0001	^Q	Q	HGR

Assembly Lines

Hex	Dec	Binary	Key	Screen	Applesoft
\$92	146	1001 0010	^R	R	HCOLOR=
\$93	147	1001 0011	^S	S	HPLLOT
\$94	148	1001 0100	^T	T	DRAW
\$95	149	1001 0101	^U	U	XDRAW
\$96	150	1001 0110	^V	V	HTAB
\$97	151	1001 0111	^W	W	HOME
\$98	152	1001 1000	^X	X	ROT=
\$99	153	1001 1001	^Y	Y	SCALE=
\$9A	154	1001 1010	^Z	Z	SHLOAD
\$9B	155	1001 1011	^[[TRACE
\$9C	156	1001 1100	^\	\	NOTRACE
\$9D	157	1001 1101	^]]	NORMAL
\$9E	158	1001 1110	^^	^	INVERSE
\$9F	159	1001 1111	^_	_	FLASH
\$A0	160	1010 0000	Space	Space	COLOR=
\$A1	161	1010 0001	!	!	POP
\$A2	162	1010 0010	"	"	VTAB
\$A3	163	1010 0011	#	#	HIMEM:
\$A4	164	1010 0100	\$	\$	LOMEM:
\$A5	165	1010 0101	%	%	ONERR
\$A6	166	1010 0110	&	&	RESUME
\$A7	167	1010 0111	'	'	RECALL
\$A8	168	1010 1000	((STORE
\$A9	169	1010 1001))	SPEED=
\$AA	170	1010 1010	*	*	LET
\$AB	171	1010 1011	+	+	GOTO
\$AC	172	1010 1100	,	,	RUN
\$AD	173	1010 1101	-	-	IF
\$AE	174	1010 1110	.	.	RESTORE
\$AF	175	1010 1111	/	/	&
\$B0	176	1011 0000	0	0	GOSUB
\$B1	177	1011 0001	1	1	RETURN
\$B2	178	1011 0010	2	2	REM
\$B3	179	1011 0011	3	3	STOP
\$B4	180	1011 0100	4	4	ON
\$B5	181	1011 0101	5	5	WAIT
\$B6	182	1011 0110	6	6	LOAD
\$B7	183	1011 0111	7	7	SAVE
\$B8	184	1011 1000	8	8	DEF FN
\$B9	185	1011 1001	9	9	POKE
\$BA	186	1011 1010	:	:	PRINT
\$BB	187	1011 1011	;	;	CONT
\$BC	188	1011 1100	<	<	LIST
\$BD	189	1011 1101	=	=	CLEAR
\$BE	190	1011 1110	>	>	GET
\$BF	191	1011 1111	?	?	NEW
\$C0	192	1100 0000	@	@	TAB
\$C1	193	1100 0001	A	A	TO
\$C2	194	1100 0010	B	B	FN
\$C3	195	1100 0011	C	C	SPC(
\$C4	196	1100 0100	D	D	THEN
\$C5	197	1100 0101	E	E	AT
\$C6	198	1100 0110	F	F	NOT
\$C7	199	1100 0111	G	G	STEP
\$C8	200	1100 1000	H	H	+
\$C9	201	1100 1001	I	I	-
\$CA	202	1100 1010	J	J	*
\$CB	203	1100 1011	K	K	/

Hex	Dec	Binary	Key	Screen	Applesoft
\$CC	204	1100 1100	L	L	;
\$CD	205	1100 1101	M	M	AND
\$CE	206	1100 1110	N	N	OR
\$CF	207	1100 1111	O	O	>
\$D0	208	1101 0000	P	P	=
\$D1	209	1101 0001	Q	Q	<
\$D2	210	1101 0010	R	R	SGN
\$D3	211	1101 0011	S	S	INT
\$D4	212	1101 0100	T	T	ABS
\$D5	213	1101 0101	U	U	USR
\$D6	214	1101 0110	V	V	FRE
\$D7	215	1101 0111	W	W	SCRN(
\$D8	216	1101 1000	X	X	PDL
\$D9	217	1101 1001	Y	Y	POS
\$DA	218	1101 1010	Z	Z	SQR
\$DB	219	1101 1011	[[RND
\$DC	220	1101 1100	\	\	LOG
\$DD	221	1101 1101]]	EXP
\$DE	222	1101 1110	^	^	COS
\$DF	223	1101 1111	_	_	SIN
\$E0	224	1110 0000	`	`	TAN
\$E1	225	1110 0001	a	a	ATN
\$E2	226	1110 0010	b	b	PEEK
\$E3	227	1110 0011	c	c	LEN
\$E4	228	1110 0100	d	d	STR\$
\$E5	229	1110 0101	e	e	VAL
\$E6	230	1110 0110	f	f	ASC
\$E7	231	1110 0111	g	g	CHR\$
\$E8	232	1110 1000	h	h	LEFT\$
\$E9	233	1110 1001	i	i	RIGHT\$
\$EA	234	1110 1010	j	j	MID\$
\$EB	235	1110 1011	k	k	
\$EC	236	1110 1100	l	l	
\$ED	237	1110 1101	m	m	
\$EE	238	1110 1110	n	n	
\$EF	239	1110 1111	o	o	
\$F0	240	1111 0000	p	p	
\$F1	241	1111 0001	q	q	
\$F2	242	1111 0010	r	r	
\$F3	243	1111 0011	s	s	
\$F4	244	1111 0100	t	t	
\$F5	245	1111 0101	u	u	
\$F6	246	1111 0110	v	v	
\$F7	247	1111 0111	w	w	
\$F8	248	1111 1000	x	x	
\$F9	249	1111 1001	y	y	
\$FA	250	1111 1010	z	z	
\$FB	251	1111 1011	{	{	
\$FC	252	1111 1100			
\$FD	253	1111 1101	}	}	
\$FE	254	1111 1110	~	~	
\$FF	255	1111 1111	Rubout	Rubout	

Hi-Res Memory Map



Appendix F: Zero-Page Memory Usage

Special Locations

This table was adapted from Jon Bettencourt's *Apple II Info Archives*, the *Applesoft II BASIC Programming Reference Manual*, and *Beneath Apple DOS* (Worth and Lechner).

\$0A–\$0C JMP to USR routine	\$79, \$7A BASIC address of line number to be executed next
\$1A, \$1B BASIC hi-res shape pointer	\$7B, \$7C BASIC current line number of DATA
\$1C BASIC last COLOR used	\$7D, \$7E BASIC next address of DATA
\$1D BASIC line-plotting counter	\$7F, \$80 BASIC address of INPUT/DATA
\$20 Left margin (0–39/79, default 0)	\$81, \$82 BASIC last-used variable name
\$21 Width (1–40 or 80, default 40, 0 crashes Applesoft)	\$83, \$84 BASIC address of last used variable's value
\$22 Top margin (0–23, default 0, or 20 for graphics)	\$9B, \$9C Pointer for FNDLIN (\$D61A) and GETARYPT (\$F7D9)
\$23 Bottom margin (0–24, default 24)	\$9D–\$A3 Floating-point accumulator FAC
\$24 Horizontal cursor position (0–39/79)	\$A5–\$AB Floating-point argument ARG
\$25 Vertical cursor position (0–23)	\$AF, \$B0 BASIC program end address
\$26, \$27 GBASL/H BASIC address of leftmost byte of current plot line	\$B1–\$C8 CHRGET subroutine; BASIC calls here for the next character
\$28, \$29 BASL/H cursor position address	\$B8–\$B9 Pointer to last character from CHRGET
\$2B BOOT slot × 16	\$C9–\$CD BASIC RND random number
\$2C Lo-res HLIN/VLIN endpoint	\$D6 BASIC protection flag, default is \$00, set to \$FF for run-only mode
\$30 COLOR × 17	\$D8–\$DE ONERR pointers/scratch
\$32 Text mask (\$FF = Normal, \$3F = Inverse, \$7F = Flashing)	\$DF ERRSTK stack pointer
\$33 Prompt character	\$E0, \$E1 Horizontal HPLOT coordinate
\$36, \$37 CASL/H output routine address	\$E2 Vertical HPLOT coordinate
\$38, \$39 KASL/H input routine address	\$E4 HCOLOR (0=0, 1=42, 2=85, 3=127, 4=128, 5=170, 6=213, 7=255)
\$48, \$49 RWTS IOB address	\$E6 HGR page: HGR=\$20, HGR2=\$40
\$50, \$51 Result of the conversion of the FAC to a 16-bit integer	\$E7 SCALE value (0 = 256)
\$67, \$68 BASIC program start address (default is \$0801)	\$E8, \$E9 Address of start of shape table
\$69, \$6A BASIC variables start address	\$EA DRAW/XDRAW collision counter
\$6B, \$6C BASIC array start address	\$F1 SPEED value (subtracted from 256)
\$6D, \$6E BASIC variables end address	\$F3 Text OR mask for flashing text
\$6F, \$70 BASIC string data start address	\$F4–\$F8 ONERR pointers
\$73, \$74 BASIC HIMEM address + 1	\$F9 ROT value
\$75, \$76 BASIC current line number	\$FF Used by STR\$ function
\$77, \$78 BASIC line number where END or STOP or BREAK occurred	

Memory Usage Table

This table comes from the `comp.sys.apple2` FAQ. The information is drawn from the Apple II technical manuals, *Beneath Apple DOS* (Don Worth and Pieter Lechner), and *Exploring Apple GS/OS and ProDOS 8* (Gary B. Little).

Hi	Low Nibble of Address															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	AP	AP	A	A	A	A	-	-	-	-	A	A	A	A	A	A
1	A	A	A	A	A	A	A	A	A	-	A	A	A	A	-	*
2	M	M	M	M	M	M	MA3	MA3	M	M	M3	M3	MA3	MA3	M3	MA3
3	M	M	M	M	M	M3	M3B	M3B	M3B	M3B	MP	MP	MP	MP	M3P	M3P
4	M3P	M3P	M3P	M3P	M3P	M3P	M3P	M3P	M3P	MP	I3P	I3P	I3P	I3P	MP	M
5	MA	MA	MA	MA	MA	MAI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI
6	AI	AI	AI	AI	AI	AI	AI	AI3	AI3	AI3	AI3	AI	AI	AI	AI	AI3
7	AI3	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI
8	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI
9	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI
A	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI3
B	AI3	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI
C	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI3	AI3	AI3	AI3	I	I
D	AI	AI	AI	AI	AI	AI	I	I	AI3	AI	AI	AI	AI	AI	AI	AI
E	A	A	A	-	A	A	A	A	A	A	A	-	-	-	-	-
F	A	A	A	A	A	A	A	A	A	A	-	-	-	-	-	-

M Monitor;

* used in early Apple //e ROMs, now free

A Applesoft BASIC

I Integer BASIC

3 DOS 3.3

P ProDOS (\$40-\$4E is saved before and restored after use)

B ProDOS BASIC . SYSTEM (also uses all Applesoft locations)

- Free; not used

Appendix G: Beginner's Guide to Merlin

This section is adapted from T. Petersen's "Beginner's Guide to Using Merlin," in the *Merlin Pro User's Manual*, Roger Wagner Publishing, 1984. The instructions should work on both the original *Merlin Macro Assembler* and the *Merlin Pro Macro Assembler*.

The purpose of an assembler is to translate human-readable code into machine instructions which then can be executed by the computer. For 6502 assembly language the code consists of a series of three-letter commands (the "opcodes") along with their associated data (the "operands"). With an assembler such as *Merlin* you can also use optional labels and macros to make your code easier to read and debug.

Control Modes

Merlin has two main modes of operation: Executive Control Mode and Editor Control Mode.

The Executive Control Mode is the main menu which appears when you start the program (see the image below). The prompt is indicated by the "%" character. The Executive Control Mode lets you perform disk actions such as loading and saving source code or object code, quitting to BASIC, or switching to the Editor/Assembler.

The Editor Control Mode consists of the Editor, the Assembler, and the Linker. The prompt is indicated by the ":" character. The Editor Control Mode lets you enter and modify code, define macros, assemble your code, and link in external files.

```
MERLIN-PRO  2.43
           By Glen Bredon

C :Catalog
L :Load source
S :Save source
A :Append file
R :Read text file
W :Write text file
D :Drive change
E :Enter ED/ASM
O :Save object code
Q :Quit

           Source: A#0901,L#0000
Drive: 2
%
```

Getting Started

As discussed in chapter three, assembly code typically has the following form:

```
7 START JSR BELL ; RING THE BELL
```

Each line consists of several fields: the line number, an optional label (START), the command (JSR), the operand (using a label such as BELL or a number such as \$FBDD), and an optional comment. In *Merlin*, the line numbers are added automatically and cannot be edited. When listing or assembling the program, all of those fields are separated by tab characters to produce nicely formatted output. When inputting code, you need type only a single <SPACE> to advance from one field to the next—you do not need to insert tabs yourself.

To get started, try creating a short program that will make your Apple beep once:

1. Boot your *Merlin* or *Merlin Pro* disk.
2. After the main Executive Control Mode menu appears, type “E” at the “%” prompt to enter the Editor Control Mode.
3. To enter a new program, at the “:” prompt type “A” (for Add) and press <RETURN>. You should see a “1” appear and the cursor should be placed one space to the right of that line number. As you enter code the line numbers will advance automatically. These line numbers are used only while editing code in the Editor and are not part of your actual program.
4. On line 1, hit <CTRL>P. A line of asterisks should appear. An asterisk as the first character indicates a comment line. Anything after the first asterisk will be ignored by the assembler. Hit <RETURN> to accept the line and advance to line 2.
5. On line 2, type a single <SPACE> and then hit <CTRL>P. You should now see an asterisk at either end of the line. Space over a few characters and then type “DEMO PROGRAM 1”. Hit <RETURN> to accept the line. You do not need to have the cursor at the end of the line when you hit <RETURN>—the entire line will be accepted, regardless of where the cursor is located.
6. On line 3, again hit <CTRL>P and then <RETURN> to finish making a nicely formatted box of asterisks containing your program name.
7. On line 4, type a single asterisk and hit <RETURN>. From this point on, it will be assumed that you hit <RETURN> to complete each line.
8. On line 5, hit the <SPACE> bar once to advance to the command field, type ORG, hit <SPACE> again to advance to the operand field, then type \$8000. So far your program should look like this:


```

1 *****
2 *      DEMO PROGRAM 1      *
3 *****
4 *
5          ORG  $8000

```

The ORG defines the origin, the memory location from which the program is designed to run.

Quick tip: If you make a mistake, don't panic. Hit <RETURN> on a blank line to exit from Add mode. Type "L" to list your program. Type "D n" to delete line n. Type "A" to re-enter Editor Control Mode and add to your current program.

9. Now we will use our first label. Type:

```
BELL<SPACE>EQU<SPACE>$FBDD
```

This defines the label BELL to be equal to the hex value FBDD. Wherever you use the label BELL in an expression, the assembler will automatically replace it with \$FBDD. Why not simply use the address \$FBDD everywhere? Well, using a label makes the code easier to read and also makes it easier to change the location in the future.

10. Now we need to ring our bell. On line 7, type:

```
START<SPACE>JSR<SPACE>BELL<SPACE>; RING THE BELL
```

Notice that we started our comment with a semicolon. Any characters within the comment field will be ignored; using the semicolon just makes it clear that this is a comment.

11. We're almost done! On line 8, type:

```
DONE<SPACE>RTS
```

12. On line 9 press <RETURN> to exit from Add mode. Because line 9 was empty, it will not be added to your program.

Type "L" to get a listing of your program. It should look like this:

```

1 *****
2 *      DEMO PROGRAM 1      *
3 *****
4 *
5          ORG  $8000
6 BELL    EQU  $FBDD
7 START   JSR  BELL          ; RING THE BELL
8 DONE    RTS

```

Note that each string of characters has been moved to the correct field: labels, commands, operands, and comments. In summary, when adding code, space once to advance to the next field.

Deleting Lines

If you make a mistake or no longer need certain lines, you can delete lines while in the Editor Control Mode. For example:

1. While you are at the “:” prompt, type `D6<RETURN>`. Nothing changes on the screen.
2. Type “L” to list your program. Notice that the original line 6 (with the BELL) is now gone and the remaining lines have moved up.
3. Type `D5,6<RETURN>` to delete the range of lines from 5–6.
4. Typing “L” reveals that our poor program now has only one line of code left, just the RTS.

Caution: Notice that the automatic renumbering caused the line numbers to shift upward. If you intend to delete several lines in succession, be sure to start by deleting the highest desired line number and working backwards to the lowest.

Inserting Lines

We now need to restore our deleted lines.

1. At the “:” prompt, type `I5<RETURN>` to insert new lines starting just before line 5.
2. Type our missing three lines, making sure to use spaces to separate the fields:

```

                ORG  $8000
BELL           EQU  $FBDD
START         JSR  BELL      ; RING THE BELL

```

3. Again, hit `<RETURN>` on the next empty line to return to Editor Control Mode.
4. Type “L” to confirm that the code has been restored.

Editing Lines

While editing a line you can use certain keyboard shortcuts to insert or delete characters. Try this:

1. At the “:” prompt, type “E8” to edit line 8. Line 8 should appear with the cursor over the D in DONE.
2. Press <CTRL>D to delete the character under the cursor. Press <CTRL>D three more times.
3. Hit <RETURN> to accept the changes and finish editing the line. Type “L” to list your program and confirm that the last line now has just the RTS command but no label.
4. Type “E8” to re-edit line 8. Now, press <CTRL>I to go into insert mode. Type the word DONE and press <RETURN>. What do you think happens if you forget to press <CTRL>I? If you’d like, go back and repeat steps 1–4 but skip the <CTRL>I.

Notice that when we did “E8” and finished editing our line, we returned to Editor Control Mode. You can also type a range of lines, such as “E3,6”. This will call up each line from 3–6 in succession. Pressing <RETURN> will take you to the next line to edit, until you’ve reached the end of your range.

Tip: If you have completely botched your line, you can press <CTRL>C to cancel the changes for the current line and return to Editor Control Mode.

Assembling the Code

The next step is to assemble and run our code. At the “:” prompt, type ASM<RETURN>. On your screen should appear the following:

```
UPDATE SOURCE (Y/N)?
```

Type “N” and you should then see:

```

1 *****
2 *          DEMO PROGRAM 1          *
3 *****
4 *
5             ORG  $8000
6 BELL       EQU  $FBDD
8000: 20 DD FB 7  START   JSR  BELL
8003: 60      8  DONE    RTS

--End assembly, 4 bytes, Errors: 0

Symbol table - alphabetical order:
  BELL  = $FBDD  ? DONE  = $8003  ? START  = $8000
Symbol table - numerical order:
? START = $8000  ? DONE  = $8003      BELL  = $FBDD
```

If the system beeps and displays an error message, remember the line number that was referenced and press <RETURN> until the assembly completes. Then go back through your program and compare it with the listing above. Use your new-found editing skills to correct the line, then re-assemble by typing ASM.

To the left of the line numbers we now see the assembled machine code. For example, the JSR BELL has been converted to 20 DD FB, where the 20 is the hexadecimal code for JSR and DD FB is the BELL address in reverse byte order. The next line contains a single opcode, the 60 for the RTS, to return from our subroutine. Notice that none of the labels or comments are within the machine-language code on the left-hand side. Finally, we see that the code has been assembled at address \$8000, as we instructed with the ORG command.

Saving and Running Your Program

Assuming that your code assembled with no errors, you can now save and run your program.

1. At the ":" prompt, type "Q" to return to the Executive Control Mode. Your source code and object code are safe in memory. If you wish, you could return to the Editor and continue editing your code.
2. At the "%" prompt, hit "S" to save your source code. Type a filename such as DEMO1. The file will be saved with ".S" appended to indicate that it is a source file.
3. Now hit "O" to save your object code. *Merlin* will display the same filename with a "?" at the end. Hit "Y" to accept DEMO1 as the object file name. Because the source code file had the ".S" appended to it, the two files will not conflict.
4. Now hit "Q" to quit *Merlin*. Type CATALOG to verify that your program was saved. Then type BRUN DEMO1 to run your program. You should hear a BEEP!

Congratulations! You've just written your first 6502 assembly-language program!

List of Programs

AL03-SAMPLE PROGRAM.....	15	AL16-POINTER SETUP ROUTINE.....	149
AL03-TEST PROGRAM 1.....	19	AL17-INTEGGER VARIABLE READER.....	156
AL04-LOOP PROGRAM 1.....	25	AL17-REAL VARIABLE READER.....	158
AL05-LOOP PROGRAM 2.....	27	AL17-STRING VARIABLE READER.....	159
AL05-LOOP PROGRAM 2A.....	28	AL17-INTEGGER VARIABLE SENDER.....	162
AL05-LOOP PROGRAM 2B.....	30	AL17-REAL VARIABLE SENDER.....	163
AL05-LOOP PROGRAM 3.....	31	AL17-STRS VARIABLE SENDER.....	164
AL05-PADDLE PROGRAM 1.....	32	AL18-HIRES DEMO 1.....	169
AL05-PADDLE PROGRAM 1A.....	34	AL18-BALL.....	175
AL06-PADDLE PROGRAM 2A.....	39	AL19-HIRES DOT.....	177
AL06-KEYBOARD PROGRAM 1A.....	42	AL19-HIRES ONE DOT PROGRAM.....	179
AL06-KEYBOARD PROGRAM 1B.....	43	AL19-HIRES LOTS DOTS.....	184
AL07-SAMPLE DATA PROGRAM.....	48	AL20-HIRES BASE ADDRESS.....	190
AL07-SCREEN CLEAR PROGRAM 1A.....	50	AL21-HIRES PLOT.140.....	201
AL07-SCREEN CLEAR PROGRAM 1B.....	51	AL21-HIRES PLOT.560.....	204
AL08-SOUND ROUTINE 2.....	56	AL21-PLOT LINES.....	206
AL08-SOUND ROUTINE 3.....	56	AL22-HIRES PLOT.140+.....	210
AL08-SOUND ROUTINE 4.....	58	AL22-HIRES PLOT.560+.....	212
AL08-SOUND ROUTINE 5.....	59	AL22-HIRES PLOT.560W.....	214
AL09-BYTE DISPLAY PROGRAM 1.....	61	AL22-PLOT LINES.....	217
AL09-BYTE DISPLAY PROGRAM 2.....	63	AL23-HI-RES SCR N FNCTN.....	219
AL10-ADC SAMPLE PROGRAM 1.....	67	AL24-SCANNER-XDRAW,XDRAW.....	229
AL10-ADC SAMPLE PROGRAM 2.....	67	AL24-SCANNER-DRAW,XDRAW.....	232
AL10-ADC SAMPLE PROGRAM 3.....	68	AL25-SIMPLE NOISE ROUTINE.....	235
AL10-ADC SAMPLE PROGRAM 4.....	68	AL25-SIMPLE NOISE ROUTINE 2.....	237
AL10-ADC SAMPLE PROGRAM 5A.....	69	AL25-SIMPLE RAMP NOISE ROUTINE.....	240
AL10-ADC SAMPLE PROGRAM 5B.....	70	AL25-SIMPLE EXPLOSION ROUTINE.....	241
AL10-ADC SAMPLE PROGRAM 5C.....	71	AL25-SHOOTER PROGRAM.....	245
AL10-ADC SAMPLE PROGRAM 5D.....	71	AL26-BASIC TO FAC.....	256
AL10-SBC SAMPLE PROGRAM 6.....	72	AL26-FAC TO MEMORY.....	257
AL10-BPL KEYTEST PROGRAM 1.....	75	AL26-MEMORY TO FAC.....	258
AL10-BPL KEYTEST PROGRAM 2.....	75	AL26-FAC TO BASIC.....	259
AL10-BPL BUTTON TEST.....	76	AL26-BASIC.FAC.MEM.FAC.BAS.....	260
AL11-GENERAL PURPOSE RWTS.....	83	AL26-BASIC.FAC.MEM.FAC.BAS USR.....	261
AL12-BINARY FUNCTION DISPLAY.....	99	AL27-M.L. ADDITION SUBR 1.....	266
AL13-DATA-TYPE PRINT 1.....	105	AL27-M.L. ADDITION SUBR 2.....	268
AL13-SPECIAL PRINT 2.....	107	AL27-M.L. ADDITION SUBR 3.....	269
AL13-INPUT ROUTINE FOR BINARY.....	108	AL28-BCD DEMO ROUTINE 1.....	272
AL13-INPUT ROUTINE FP BASIC.....	110	AL28-BCD DEMO ROUTINE 2.....	273
AL14-NAME FILE DEMO PROGRAM.....	113	AL28-BCD DEMO 'INC' ROUTINE.....	274
AL14-NAME FILE DEMO PROGRAM 2.....	120	AL28-BCD DEMO 'DEC' ROUTINE.....	274
AL15-NON-RELOCATABLE PRINT DEMO.....	127	AL28-BCD ADDITION ROUTINE.....	275
AL15-NON-RELOCATABLE JMP DEMO.....	129	AL28-BCD SUBTRACT ROUTINE.....	275
AL15-RELOCATABLE JMP 1.....	129	AL28-BCD PRINT ROUTINE 1.....	276
AL15-RELOCATABLE JMP 2.....	130	AL28-BCD PRINT ROUTINE 2.....	276
AL15-LOCATOR 1.....	131	AL28-BCD PRINT ROUTINE 3.....	278
AL15-LOCATOR 2.....	132	AL29-CONTROL CHARACTER DISPLAY.....	286
AL15-RELOCATABLE PRINT 1.....	133	AL29-SPECIAL DISPLAY ROUTINE.....	287
AL15-NON-RELOCATABLE JSR DEMO.....	134	AL30-SIMPLE CASE CONVERTER.....	295
AL15-RELOCATABLE JSR SIMULATION.....	135	AL30-LOWERCASE INPUT ROUTINE.....	297
AL15-RELOCATABLE PRINT 2.....	136	AL31-CHARACTER GENERATOR.....	305
AL15-RELOCATABLE PRINT 3.....	138	AL31-ASCII CHARACTER SET.....	310
AL15-RELOCATABLE JMP 3.....	139	AL32-CHARACTER EDITOR.....	314
AL16-SOUND ROUTINE 3A.....	144	ASSEMBLY LINES CONTEST WINNER.....	340
AL16-SOUND ROUTINE 3B.....	148		

Directory Listing for Program Disks

The programs are contained on two floppy disks. For ProDOS the files are contained in a folder named CODE. The .S suffix indicates a Merlin source file. The .A suffix indicates an Applesoft BASIC file. Names without a suffix are compiled object files.

/ALDISK1/CODE		/ALDISK2/CODE	
AL03.SAMPLE.S	AL11.RWTS.S	AL18.BALL.A	AL27.MLADD.A
AL03.TEST1	AL12.OPERATOR	AL18.HIRES1	AL27.MLADD1.S
AL03.TEST1.S	AL12.OPERATOR.A	AL18.HIRES1.S	AL27.MLADD2.S
AL04.LOOP1.S	AL12.OPERATOR.S	AL19.HIRESDOT	AL27.MLADD3
AL05.LOOP2.S	AL13.INPUTBIN.S	AL19.HIRESDOT.A	AL27.MLADD3.S
AL05.LOOP2A.S	AL13.INPUTFP	AL19.HIRESDOT.S	AL28.BCDADD.S
AL05.LOOP2B.S	AL13.INPUTFP.A	AL19.LOTSDOTS	AL28.BCDDEC.S
AL05.LOOP3.S	AL13.INPUTFP.S	AL19.LOTSDOTS.S	AL28.BCDDEMO1.S
AL05.PADDLE1.S	AL13.PRINT1.S	AL20.HGRADDR	AL28.BCDDEMO2.S
AL05.PADDLE1A.S	AL13.PRINT2.S	AL20.HGRADDR.S	AL28.BCDINC.S
AL06.KBRD1A	AL14.FILE1	AL21.PLOT140	AL28.BCDPRNT1.S
AL06.KBRD1A.S	AL14.FILE1.S	AL21.PLOT140.S	AL28.BCDPRNT2.S
AL06.KBRD1B	AL14.FILE2	AL21.PLOT560	AL28.BCDPRNT3.S
AL06.KBRD1B.S	AL14.FILE2.S	AL21.PLOT560.S	AL28.BCDSUB.S
AL06.PADDLE2A	AL15.LOCATE1.S	AL21.PLOTLINE.A	AL29.CTRLCHAR
AL06.PADDLE2A.S	AL15.LOCATE2.S	AL22.PLOT140	AL29.CTRLCHAR.S
AL07.HGR	AL15.NRJMP.S	AL22.PLOT140.S	AL29.DISPLAY
AL07.HGR.S	AL15.NRJSR.S	AL22.PLOT560	AL29.DISPLAY.S
AL07.SAMPLE	AL15.NRPRINT.S	AL22.PLOT560.S	AL30.CASECVRT
AL07.SAMPLE.S	AL15.PRINT1	AL22.PLOT560W	AL30.CASECVRT.A
AL07.SCREEN1A	AL15.PRINT1.S	AL22.PLOT560W.S	AL30.CASECVRT.S
AL07.SCREEN1A.S	AL15.PRINT2	AL22.PLOTLINE.A	AL30.LCINPUT
AL07.SCREEN1B	AL15.PRINT2.S	AL23.HGRSCRN	AL30.LCINPUT.A
AL07.SCREEN1B.S	AL15.PRINT3	AL23.HGRSCRN.A	AL30.LCINPUT.S
AL08.SOUND2	AL15.PRINT3.S	AL23.HGRSCRN.S	AL31.ASCII
AL08.SOUND2.S	AL15.RELJMP1.S	AL24.SCAN1	AL31.ASCII.S
AL08.SOUND3	AL15.RELJMP2.S	AL24.SCAN1.S	AL31.CHARGEN
AL08.SOUND3.A	AL15.RELJMP3.S	AL24.SCAN2	AL31.CHARGEN.A
AL08.SOUND3.S	AL15.RELJSR.S	AL24.SCAN2.S	AL31.CHARGEN.S
AL08.SOUND4	AL16.POINTER	AL25.EXPLODE	AL32.CHAREDIT
AL08.SOUND4.S	AL16.POINTER.S	AL25.EXPLODE.S	AL32.CHAREDIT.A
AL08.SOUND5	AL16.SOUND3A.S	AL25.NOISE	AL32.CHAREDIT.S
AL08.SOUND5.S	AL16.SOUND3B	AL25.NOISE.A	AL32.PIGFONT
AL09.BYTE1	AL16.SOUND3B.A	AL25.NOISE.S	ALAPP.CONTEST
AL09.BYTE1.S	AL16.SOUND3B.S	AL25.NOISE2	ALAPP.CONTEST.S
AL09.BYTE2	AL17.READINT	AL25.NOISE2.A	
AL09.BYTE2.S	AL17.READINT.A	AL25.NOISE2.S	
AL10.ADC1.S	AL17.READINT.S	AL25.RAMP	
AL10.ADC2.S	AL17.READREAL	AL25.RAMP.A	
AL10.ADC3.S	AL17.READREAL.S	AL25.RAMP.S	
AL10.ADC4.S	AL17.READSTR	AL25.SHOOTER	
AL10.ADC5A.S	AL17.READSTR.S	AL25.SHOOTER.S	
AL10.ADC5B.S	AL17.SENDINT	AL26.BASICFAC.S	
AL10.ADC5C.S	AL17.SENDINT.A	AL26.BFMFB	
AL10.ADC5D.S	AL17.SENDINT.S	AL26.BFMFB.S	
AL10.BPLKEY1.S	AL17.SENDREAL	AL26.BFMFBUSR	
AL10.BPLKEY2.S	AL17.SENDREAL.S	AL26.BFMFBUSR.S	
AL10.BPLPB.S	AL17.SENDSTR	AL26.FACBASIC.S	
AL10.SBC6.S	AL17.SENDSTR.S	AL26.FACMEM.S	
AL11.RWTS		AL26.MEMFAC.S	

Index

6

6502 bug..... 140, 335, 370
65C02.....13, 140, 327-336, 345, 356, 370, 377-380,
387-390, 394, 398-403

A

ABS subroutine.....408
absolute addressing.....18, 45, 128, 328
Accumulator.....6, 18
ADC.....66, 274, 344
addition in assembly.....65, 265
addition, two-byte.....69
address.....3, 4
addressing modes.....18, 45, 328
ampersand vector.....264
AND.....92, 278, 322, 345
Apple ///.....336, 337
Apple //e.....13, 94, 291, 296, 320, 335-337
Apple II.....13, 53, 94, 296, 336, 411
Apple II Plus.....5, 13, 94, 291, 296, 336
Applesoft array variables.....154
Applesoft BASIC.....5, 10, 11, 110, 128
Applesoft variables.....151
ARG register.....265
ASC directive.....106
ASCII.....19, 40, 94, 106, 305, 313, 411
ASL.....89, 186, 348
ASM command.....16
assembler.....2, 9, 13, 15
assembly language.....2, 9
asterisks for comments.....17, 421
ATN subroutine.....409

B

base 16 numbers.....4, 22
base 2 numbers.....21, 65
base address.....17
BASL location.....292, 302
BBR.....331
BBS.....331
BCC.....38, 70, 349
BCS.....38, 350
BELL subroutine.....17, 135, 321, 405, 422
BELL1 subroutine.....405
BEQ.....27, 40, 280, 351
Binary Coded Decimal... 264, 271, 344, 359, 383, 384
binary numbers.....22, 65
BIT.....96, 130, 211, 216, 293, 299, 330, 351
bits.....22, 65
BKGND subroutine.....169, 407
BLOAD command.....35, 113
BMI.....75, 353
BNE.....24, 25, 40, 280, 354
borrow, for subtraction.....72
BPL.....75, 355
BRA.....331
branch commands.....38
branch instruction.....25, 27

branch offsets.....28
branch tests for ranges.....38
branch, reverse.....28
break message.....3, 5, 356
BREAK subroutine.....370
BRK.....256, 356
BRK vector location.....356, 370
BRUN command.....35
BSAVE command.....113
buffer, memory.....118
BVC.....96, 357
BVS.....96, 358

C

CALL command.....6, 15, 143
carry flag.....38, 66, 90, 273
CATALOG command.....106
CH location.....305
CHK directive.....xiii, 16
CHKCOM subroutine....150, 157, 160, 219, 257, 267
CHKNUM subroutine.....158, 163, 219
CHKSTR subroutine.....159, 165
CHRGET subroutine.....147
CLC.....67, 358
CLD.....272, 359
CLL.....359
CLRSCR subroutine.....407
CLRTOP subroutine.....407
CLV.....360
CMP.....38, 360
code location, determining.....131
COLBYTE location.....198, 211, 213
COMBYTE subroutine.....148, 219, 236
command field in assembly.....17, 421
comment field in assembly.....17, 421
compare commands.....38
complements, number.....73
CONUPK subroutine.....269
COS subroutine.....409
counters.....21
COUT subroutine.....29, 120, 277, 281, 323, 370, 404
COUT1 subroutine.....281, 304, 404
CPX.....40, 362
CPY.....40, 362
CROUT subroutine.....404
CROUT1 subroutine.....404
CSW location.....281, 291, 322, 370
CURLIN location.....126
CV location.....302

D

debugging.....357, 374
DEC.....23, 330, 363
decimal number formula.....22
decrementing.....23
delays in execution.....54
delimiters.....16
DEX.....23, 364

DEY.....23, 365
 DFB directive.....48
 directives, assembler.....17
 disassembly.....10
 diskette hard-sectoring.....80
 diskette organization.....78
 diskette sector interleaving.....79
 diskette soft-sectoring.....80
 DIV10 subroutine.....409
 DOS.....16, 35, 77, 79, 281, 290, 294
 DOS bell modification.....88
 DOS buffer pointer.....85
 DOS catalog key modification.....87
 DOS disk-volume modification.....86
 DOS input/output vector.....284, 294, 322
 DOS IOB table.....81, 83, 84
 DRAW command.....225
 DRAW subroutine.....169, 172, 229, 408
 dummy return address.....131

E

EOR.....97, 365
 EQU directive.....17, 422
 exclusive OR.....97
 EXP subroutine.....408
 exponent, real number.....254

F

FAC register.....254, 265
 FACEXP location.....267
 FADD subroutine.....270
 FADDH subroutine.....409
 FADDDT subroutine.....267, 269
 FCOMP subroutine.....409
 FDIV subroutine.....270
 flags, Status Register.....22
 FLASH command.....285
 floating-point accumulator (FAC).....150, 236, 254
 flow of control, machine language.....94
 FMULT subroutine.....270
 forced branch statement.....129, 134
 FOUT subroutine.....409
 FPWRT subroutine.....409
 FRMEVL subroutine.....160, 268
 FRMNUM subroutine.....150, 159, 219, 256, 257, 267
 FSUB subroutine.....270

G

GBAS location.....197, 211
 GETADR subroutine.....150, 159, 219, 256
 GETLN subroutine.....109, 118, 296, 405
 GETLN1 subroutine.....406
 GETLNZ subroutine.....406
 GIVAYF subroutine.....163, 219
 GOSUB command.....11, 17, 136, 371, 382
 graphics, table driven.....183

H

HCLR subroutine.....169, 407
 HCOLOR subroutine.....169, 407
 HCOLOR1 location.....198

HEX directive.....48
 hexadecimal.....4, 22, 65, 271
 HFIND subroutine.....169, 407
 HGR subroutine.....169, 407
 HGR2 subroutine.....169, 407
 hi-res 140-point mode.....201, 207
 hi-res 560-point mode.....203, 207
 hi-res collision counter.....225
 hi-res color mask.....198, 211
 hi-res color shift.....192, 203, 208
 hi-res color table.....168, 193
 hi-res entry points.....168
 hi-res fill effect.....189
 hi-res memory map.....189, 303, 417
 hi-res object velocity.....173
 hi-res screen coordinates.....168
 hi-res screen locations.....173
 hi-res screen motion.....173
 hi-res white color problem.....195, 201
 high bit.....29
 high-order byte.....11
 HIMEM command.....85, 128, 153
 HLIN subroutine.....169, 322, 408
 HLINE subroutine.....406
 HMASK location.....199, 211, 213
 HNDX location.....197
 HOME subroutine.....19
 HPAG location.....197
 HPLOT subroutine.....169, 182, 221, 407
 HPOSN subroutine.....169, 172, 197, 219, 407
 HTAB subroutine.....119

I

immediate addressing.....18, 45, 328
 implicit addressing.....45
 implied addressing.....45
 INC.....23, 330, 367
 inclusive OR.....97
 incrementing.....23
 indexed absolute indirect addressing.....329
 indexed addressing.....46, 328
 indexed indirect addressing.....47, 329
 indirect addressing.....328
 indirect indexed addressing.....46, 328
 indirect jump.....139, 281
 input routines.....108
 input vector.....291, 370
 INT subroutine.....408
 Integer BASIC.....5, 13
 integer variables.....156, 161
 interrupt vector.....370
 interrupts.....356, 359, 370, 381, 385
 INVERSE command.....285
 INVFLG location.....95, 285
 INX.....23, 368
 INY.....23, 369
 IOREST subroutine.....410
 IOSAVE subroutine.....410
 IRQ maskable interrupt.....359, 385
 IRQ subroutine.....370

IRQ vector location.....370

J

JMP.....28, 139, 281, 370
 JMP simulation.....376, 382
 JSR.....11, 13, 63, 371
 JSR simulation.....134
 JSR to JMP trick.....164

K

keyboard buffer.....42
 keyboard input.....42
 keyboard input switch.....291
 keyboard strobe.....42, 293
 KEYIN subroutine.....293, 405
 KSW location.....291, 370, 405

L

label field in assembly.....17, 421
 LANG location.....126
 LDA.....18, 371
 LDX.....18, 372
 LDY.....18, 373
 LIFO (Last-In First-Out).....61, 131
 LINNUM location.....219, 222, 256
 LOG subroutine.....409
 logarithmic form.....153
 logical operators.....92
 LOMEM command.....153
 low-order byte.....11
 lowercase text.....291
 LSR.....89, 373

M

machine language.....9
 MAKSTR subroutine.....165
 mantissa, real number.....254
 mask, AND.....93, 96, 292, 346
 mask, EOR.....366
 mask, inverse flag.....285
 mask, ORA.....277, 292, 375
 math subroutines.....263
 MAXFILES command.....78
 memory map.....3, 152
 memory page.....4
 Merlin Assembler.....13, 16, 337, 420
 Mini-Assembler.....13, 19
 mnemonics.....2, 13
 Monitor.....5, 9, 13
 Monitor subroutines.....41
 MOVAF subroutine.....267
 move command.....54
 MOVFM subroutine.....157, 258
 MOVMF subroutine.....163, 219, 258, 267
 MUL10 subroutine.....409
 multiplication and division.....90, 265, 381
 Munch-A-Bug.....13, 357

N

negative numbers.....72, 178
 NEGOP subroutine.....409

NEXTCOL subroutine.....406
 NMI (non-maskable interrupt).....385
 non-relocatable code.....128
 NOP.....54, 374
 numeric registers, temporary.....267

O

OBJ directive.....xiii, 17
 object code.....15
 one's complement.....73
 opcodes.....2, 10, 17
 OPEN command.....125
 operand field in assembly.....17, 421
 operands.....10, 421
 operational mode, machine language.....93
 ORA.....97, 277, 375
 ORG directive.....17, 422
 output routines.....105
 output vector.....94, 281, 370
 overflow flag.....96, 130

P

paddle input.....32, 39, 59, 62, 103, 171, 182, 230, 249, 342
 paddle interactions.....103, 172, 249
 paddle pushbutton.....76, 171, 249, 342
 parity, number.....91
 PHA.....62, 136, 376
 PHP.....376
 PHX.....331, 377
 PHY.....331, 378
 PLA.....62, 136, 378
 PLOT subroutine.....406
 PLP.....379
 PLX.....331, 379
 PLY.....331, 380
 pointers (vectors).....47
 POKE command.....144
 POP command.....136, 378
 POSN location.....323
 post-indexed addressing.....47, 329, 344
 PRBL2 subroutine.....405
 PRBLNK subroutine.....405
 PRBYTE subroutine.....276, 404
 pre-indexed addressing.....48, 329, 344
 PREAD subroutine.....32, 171, 410
 PRERR subroutine.....410
 PRHEX subroutine.....405
 PRNTAX subroutine.....276, 405
 processing mode, machine language.....93
 ProDOS.....100, 284, 286, 294, 312, 325
 Program Counter.....356, 381, 382, 396
 pseudo opcodes.....17
 pseudo-jump.....376, 382
 PTRGET subroutine.....157, 161, 163, 219, 260, 267

R

RAM (random access memory).....4
 random number generator.....293
 RDCHAR subroutine.....405
 RDKEY subroutine.....118, 291, 324, 370, 405

READ command.....125
 reading/writing data files.....113
 reading/writing text files.....120
 real variables.....158, 162, 252
 registers, 6502.....6, 396
 relative addressing.....46, 328
 relocatable code.....127, 371
 REM command.....17
 RESET.....385
 RMB.....331
 RND subroutine.....408
 ROL.....91, 380
 ROM (read-only memory).....4
 ROR.....91, 381
 rotate commands.....91
 RTL.....381
 RTS.....11, 13, 382
 RWTS error codes.....85
 RWTS subroutines.....78, 81, 83, 359, 384, 385

S

S-C Assembler.....xviii, 337
 SAVD subroutine.....165
 SBC.....72, 274, 382
 screen output.....30
 SCRN subroutine.....219, 407
 SEC.....72, 384
 SED.....272, 384
 SEI.....385
 self-modifying code.....137
 SEND subroutine.....223
 SETCOL subroutine.....406
 SETINV subroutine.....404
 SETNORM subroutine.....404
 SGN subroutine.....408
 shape tables.....171, 225
 shift operators.....89
 SHNUM subroutine.....169, 172, 408
 sign bit.....73
 sign flag.....75, 90
 SIGN subroutine.....408
 SIN subroutine.....409
 SMB.....331
 sneaker, wet.....4
 soft-switch hardware location.....41, 343, 353
 sound duration.....56
 sound from paddle input.....59
 sound generation.....53
 sound pitch.....56
 sound routines.....53, 235
 source code.....15
 Sourceror.....29, 337
 speaker soft-switch.....53
 SQR subroutine.....408

STA.....18, 385
 stack.....61, 62, 131
 Stack Pointer.....61, 131, 396
 Status Register.....21, 22, 37, 130, 396
 string variables.....159, 164, 386
 STX.....18, 386
 STY.....18, 387
 STZ.....331, 387
 subtraction in assembly.....65, 265
 super hi-res graphics.....195

T

TAN subroutine.....409
 TAX.....34, 388
 TAY.....34, 388
 text screen memory map.....19, 302, 416
 transfer commands.....34
 TRB.....331, 335, 389
 TSB.....331, 335, 390
 TSX.....131, 132, 390
 two's complement.....74, 178, 367
 TXA.....34, 391
 TXS.....392
 TXTPTR location.....147, 257
 TYA.....34, 393

U

UCMD location.....85
 USLOT location.....84, 90
 USR command.....255

V

vector.....35, 82, 94, 281, 291, 370
 VLINE subroutine.....406
 VTAB subroutine.....119, 302
 VTOC (Volume Table of Contents).....80

W

WAIT subroutine.....172, 183, 410
 warm-reentry vector.....35
 wrap around of numbers.....24, 363-365, 367-369
 WRITE command.....125

X

X-Register.....6, 18, 396
 XDRAW command.....225
 XDRAW subroutine.....169, 229, 408

Y

Y-Register.....6, 18, 396

Z

zero flag.....22-24, 90
 zero page addressing.....45, 328

Quick Reference

Merlin Assembler

Editor Control Mode

A – Add mode, <RETURN> to exit
E – Edit all lines
E *m* or E *m,n* – Edit line or line range
E “*string*” – Edit lines containing string
I *m* – Insert lines starting at *m*
D *m* or D *m,n* – Delete line or line range
R *m* or R *m,n* – Replace line or line range
L – List source
L *m* or L *m,n* – List specific line or range
. (period) – List from previous range
P, P *n*, P *m,n* – List without line numbers
F “*string*” – Find the given string
C “*str1*” “*str2*” – Change *str1* to *str2*
FW “*word*” – Find the given *word*
CW “*str1*” “*str2*” – Change *str1* to *str2*
EW “*word*” – Edit lines containing *word*
COPY *m* TO *n* – Copy line
COPY *l,m* TO *n* – Copy lines
MOVE *m* TO *n* – Move line
MOVE *l,m* TO *n* – Move lines
123 or \$123 – Hex/Decimal conversion

ASM – Assemble the source code
LEN – Source length and bytes remaining
MON – Exit to the Monitor
NEW – Clear the current source code
PR#1 – Send output to printer in slot 1
VAL “*expression*” – Compute expression
VID 3 or VID 0 – Turn 80-columns on/off
Q – Quit Editor, return to Executive

Expressions

2*LABEL1-LABEL2+\$231
1234+%10111
"K"- "A"+1
"0"!LABEL (“0” EOR LABEL)
LABEL&\$7F (LABEL AND \$7F)
LABEL . \$FFFF (LABEL OR \$FFFF)
*-2 (current address minus 2)

Editing Commands

<CTRL>B – Beginning of line
<CTRL>C or X – Abort Edit mode
<CTRL>D – Delete character
<CTRL>F *c* – Find character
<CTRL>I – Insert; <RETURN> to exit
<CTRL>L – Toggle lowercase/uppercase
<CTRL>N – End of line
<CTRL>O – Insert special characters
<CTRL>P – Fill line with *****
<SPACE><CTRL>P – Border with * *
<CTRL>Q – Delete rest of the line
<CTRL>R – Restore line to original
<RETURN> – Next line or exit

Merlin Pro Full Screen Editor

ĈB/ĈN – Go to beginning/end of source
ĈD/ĈR – Delete or replace the current line
ĈE – Exchange (find and replace)
ĈF – Find text
ĈI – Insert a blank line
ĈL – Locate label, marker, or line number
ĈQ – Return to Editor Control Mode
ĈX/ĈV – Clipboard cut and paste
ĈY – Select all text to the end
Ĉ8 – Create a line of asterisks
Ĉ9 – Create a box of asterisks

Pseudo Opcodes

label EQU *expression*
label KBD – Define label during assembly
ASC “*string*” – Define ASCII text
CHK – Add a checksum byte
ERR *expression* – Force error if nonzero
HEX *data* – Define hex data
LST ON or OFF – Enable/disable listing
LUP ... --^ – Loop and repeat opcodes
OBJ *expression* – Assembly address
ORG *expression* – Run address
PUT *filename* – Insert T.*filename*
SAV *filename* – Save current code
DUM/DEND – Dummy section of code
DO *expression* ... ELSE ... FIN
IF *char*,]var ... ELSE ... FIN

Monitor Commands¹

Command	Syntax	Description
Enter	CALL -151	Enter the Monitor from BASIC.
Display	300 300.320	Display the byte at \$300. Display the bytes from \$300 to \$320.
Store	300:00 01 02. . . :03 04 05	Store byte values starting at \$300. Type ":" to continue adding values.
Move	2001<2000.2FFFF	Copy memory from \$2000 to \$2FFF into location starting at \$2001.
Verify	800<400.7FFV	Display differences in memory from \$400 to \$7FF with bytes starting at \$800.
Examine	<CTRL>E :01 02 03 04 05	Display the 6502 registers. Type ":" and the new values to modify.
Go	300G	Run the program at \$300.
List	300L L	Disassemble 20 lines, starting at \$300. Type L to continue the list.
Add bytes	2F+3B	Add two bytes, display the result.
Subtract	3B-2F	Subtract two bytes, display the result.
Normal	N	Set normal video mode.
Inverse	I	Set inverse video mode.
User	<CTRL>Y	Jump to the user routine at \$3F8-3FA.
Keyboard	2<CTRL>K	Cause slot 2 to become the input source.
Printer	1<CTRL>P	Cause slot 1 to become the output device.
Exit	<CTRL>C	Exit Monitor and enter BASIC.

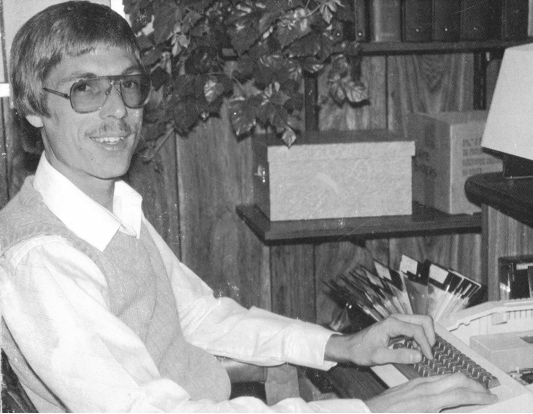
Addressing Modes

Mode	Example	Bytes	Time (µs)
Implied	RTS	1	2-7
Immediate	LDA #\$FF	2	2
Zero Page	LDA \$06	2	3-5
Zero Page Indexed, X	LDA \$06, X	2	4-6
Absolute	LDA \$C000	3	3-6
Absolute Indexed, X	LDA \$2000, X	3	4-7
Absolute Indexed, Y	LDA \$2000, Y	3	4-5
Indirect Indexed (post-indexed)	LDA (\$06), Y	2	5-6
Indexed Indirect (pre-indexed)	LDA (\$06, X)	2	6
Relative	BCC \$300	2	2-4
Indirect Jump	JMP (\$0036)	3	5-6
Zero Page Indirect [65C02]	LDA (\$06)	2	5
Indexed Absolute Indirect [65C02]	JMP (\$1234, X)	3	6

¹ [CT] Adapted from Table 3-1 in *Inside the Apple IIe*, by Gary B. Little.

About the Author

Roger Wagner bought one of the early Apple II computers in 1978 and over the next few years wrote a monthly tutorial for *Softalk* magazine. *Assembly Lines: The Book*, published in 1982, was the first book specifically about creating assembly-language programs on the Apple II.



Roger also wrote *Apple IIGS Machine Language for Beginners* and numerous programs including *Roger's Easel*, *The Programmer's Utility Pack*, and *The Write Choice*. He was the designer and creator of *HyperStudio for the Apple IIGS*, a multimedia package that soon became the most-used software in K-12 classrooms.

Technology & Learning magazine named Roger one of the top 5 “Most Important Educational Technology Gurus of the Past Two Decades” along with Seymour Papert, Bill Gates, Steve Wozniak, and Steve Jobs.

Roger coined the phrase “Copyright Friendly,” a predecessor to the Creative Commons license, and he worked with the Creative Commons organization to develop automatic attribution systems.

Roger serves on the Board of Directors of California Computer-Using Educators (CUE) Inc., and is an energetic advocate for enabling students to effectively use technology, as well as for issues relating to digital citizenship and student privacy. Finally, Roger is the inventor of the Hyper-Duino, a platform that enables students to easily create interactive maker projects that combine physical hardware with digital content.



For more information, visit <http://rogerwagner.com>.

5
6
7
8 “Roger Wagner didn’t just read the first book on programming the Apple
9 computer—he wrote it.”

10 —Steve Wozniak

11
12 “I cannot express how truly lucky I was to have gotten Roger’s *Assembly*
13 *Lines: The Book*. I am not exaggerating when I say that I merely read the first
14 few pages in chapter one and, all of a sudden, assembly language totally made
15 sense to me. Very few times in my life have I had such an important event
16 happen to me in lightning bolt-like form as when I read the first chapter and
17 *understood without error* exactly how it worked.”

18
19 —John Romero, Co-founder of id Software, designer of
20 *Wolfenstein 3D*, *Dangerous Dave*, *Doom*, and *Quake*
21



37
38 *Roger Wagner, Steve Wozniak, Bob Clardy*
39 *Hang gliding, Baja California, March 1981*

40
41 Roger Wagner’s *Assembly Lines* articles originally appeared in *Softalk*
42 magazine from October 1980 to June 1983. Now, for the first time, all thirty-
43 three articles are available in one complete volume. This edition also contains
44 new appendices on the 65C02, zero-page memory usage, and a guide to using
45 the *Merlin Assembler*. The book is designed for students of all ages: the
46 nostalgic programmer enjoying the retro revolution, the newcomer interested
47 in learning assembly coding, or the embedded-systems developer using the
48 latest 65C02 chips from Western Design Center.
49
50
51



56
57
58

ISBN 978-1-312-08940-2

